Devweb 105 : Programmation Orientée Objet (POO)

Introduction

En général, vous n'avez jusqu'à maintenant utilisé une **représentation procédurale** pour programmer, càd que vous aviez d'un côté vos données (listes, BDD...) et de l'autre côté une liste de fonctions à appliquer à ces données.

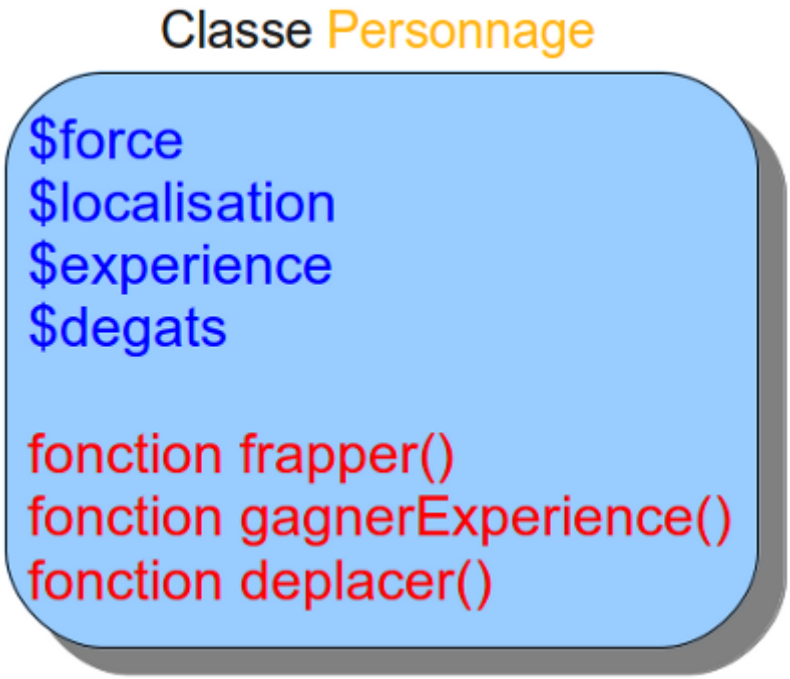
Il existe une autre représentation très courante pour programmer appelée la **POO** (ou Programmation Orientée Objet). Lorsque l'on programme en utilisant la POO, tout notre code devient un ensemble d'objets qui interagissent entre eux.

Chaque objet est défini selon des caractéristiques et un plan bien précis. En POO, ces informations sont contenues dans ce qu'on appelle des **classes**.

Les classes

Une classe est un ensemble d'attributs (~de caractéristiques) et de méthodes (~de fonctions). On peut donner l'exemple d'un **Personnage** de jeu vidéo qui aurait certains attributs (sa vie, son emplacement, son expérience, sa force...) et certaines méthodes (frapper, gagner de l'expérience, se déplacer...).

Notre classe serait donc ainsi :



L'un des avantages de la POO est que l'on peut masquer le code à l'utilisateur. En général les

attributs sont **privés** et les **méthodes** sont **publiques**. Il existe des méthodes permettant à l'utilisateur de modifier (**setter**) ou de consulter (**getter**) les attributs mais il ne peut qu'exécuter ces méthodes sans y avoir un accès direct. (Nous y reviendrons don't worry :p)

Créer un classe en PHP

Syntaxe de Base

La syntaxe de base est très simple :

```php
<?php
class Personnage // Présence du mot-clé class suivi du nom de la classe.
{
  // Déclaration des attributs et méthodes ici.
}
```

Visibilité d'un attribut ou d'une méthode

A cela on va ajouter la déclaration des attributs et des méthodes, en les précédant de leur visibilité. Comme je viens de vous l'expliquer avant, il y a 2 types de visibilité : **public** et **private**.

Si un attribut/une méthode est ***public***, *alors on pourra y avoir accès depuis n'importe où depuis l'extérieur ou l'intérieur de l'objet*.

Si un attribut/une méthode est ***private***, *alors on pourra y avoir accès uniquement depuis l'intérieur de l'objet afin de nourrir nos méthodes*.

Ceci s'appelle le principe d'encapsulation, c'est de cette manière qu'on peut interdire l'accès à nos attributs.

Création d'attributs

Comme je vous l'ai dit précédemment on va rendre les attribut privés et créer des méthodes publiques grâce auxquelles y accéder :

```php
<?php
class Personnage
{
  private $_force;        // La force du personnage
  private $_localisation; // Sa localisation
  private $_experience;   // Son expérience
  private $_degats;       // Ses dégâts
  private $_vie;          // Ses points de vie
}
```

Vous pouvez également initialiser les attributs lorsque vous les déclarez :

```php
<?php
class Personnage
{
  private $_force = 50;            // La force du personnage, par défaut à 50.
  private $_localisation = 'Lyon'; // Sa localisation, par défaut à Lyon.
  private $_experience = 1;        // Son expérience, par défaut à 1.
  private $_degats = 0;            // Ses dégâts, par défaut à 0.
  private $_vie = 100;             // Ses points de vie, par défaut à 100.

}
```

Et son équivalent en **Java**, pour ceux qui y ont déjà touché en cours d'informatique :

```java
package com.devweb105;

class Personnage {
  private int force = 50;
  private String localisation = “Lyon”;
  private int experience = 1;
  private int degat = 0;
}
```

Création des méthodes

Comme je vous l'ai dit précédemment on va rendre les méthodes publiques afin d’accéder aux attributs, mais aussi de donner des actions aux Personnages :

```php
<?php
class Personnage
{
  private $_force;        // La force du personnage
  private $_localisation; // Sa localisation
  private $_experience;   // Son expérience
  private $_degats;       // Ses dégâts
  private $_vie;          // Ses points de vie
  public function deplacer($lieu) // Une méthode qui déplacera le personnage (modifiera sa localisation).
  {
    $this->_localisation = $lieu;
  }
  public function frapper($personnage) // Une méthode qui frappera un personnage (suivant la force qu'il a).
  {
    $personnage->_vie = $personnage->_vie - $this->_degats;
  }
  public function gagnerExperience($xp) // Une méthode augmentant l'attribut $experience du personnage.
  {
```

```
    $this->_experience = $this->_experience + $xp;
  }
   //+ getters et setters (cf plus tard)
}
```

Utiliser la classe

Créer un objet

Syntaxe très simple grâce au mot-clé **new**:

```
<?php
$perso = new Personnage;
```

Soit, en java

```
Personnage perso = new Personnage;
```

Appeler les méthodes de l'objet

Pour appeler une méthode d'un objet, il va falloir utiliser l'opérateur –>. Cela donne :

```
<?php
// Nous créons une classe « Personnage ».
class Personnage
{
  private $_force;
  private $_localisation;
  private $_experience;
  private $_degats;
  // Nous déclarons une méthode dont le seul but est d'afficher un texte.
  public function parler()
  {
    echo 'Je suis un personnage !';
  }
}
```

Ajouter les getters et les setters ainsi que le constructeur

Cela donne donc avec les getters et les setters et le constructeur:

```
<?php
class Personnage
{
  private $_id;
  private $_nom;
```

```
  private $_forcePerso;
  private $_degats;
  private $_niveau;
  private $_experience;
  // Constructeur
  public function __construct($force, $nom)
  // Le constructeur de la méthode. Il initialise les attributs dès la
création de l'objet.
  {
   $this->setForcePerso($force);
   $this->setNom($nom);
   $this->setNiveau(0);
  }
  // Liste des getters
  public function id()
  {
    return $this->_id;
  }
  public function nom()
  {
    return $this->_nom;
  }
  public function forcePerso()
  {
    return $this->_forcePerso;
  }
  public function degats()
  {
    return $this->_degats;
  }
  public function niveau()
  {
    return $this->_niveau;
  }
  public function experience()
  {
    return $this->_experience;
  }
  // Liste des setters
  public function setId($id)
  {
    // On convertit l'argument en nombre entier.
    // Si c'en était déjà un, rien ne changera.
    // Sinon, la conversion donnera le nombre 0 (à quelques exceptions près,
mais rien d'important ici).
    $id = (int) $id;
    // On vérifie ensuite si ce nombre est bien strictement positif.
    if ($id > 0)
    {
      // Si c'est le cas, c'est tout bon, on assigne la valeur à l'attribut
correspondant.
```

```
      $this->_id = $id;
    }
  }
  public function setNom($nom)
  {
    // On vérifie qu'il s'agit bien d'une chaîne de caractères.
    if (is_string($nom))
    {
      $this->_nom = $nom;
    }
  }
  public function setForcePerso($forcePerso)
  {
    $forcePerso = (int) $forcePerso;
    if ($forcePerso >= 1 && $forcePerso <= 100)
    {
      $this->_forcePerso = $forcePerso;
    }
  }
  public function setDegats($degats)
  {
    $degats = (int) $degats;
    if ($degats >= 0 && $degats <= 100)
    {
      $this->_degats = $degats;
    }
  }
  public function setNiveau($niveau)
  {
    $niveau = (int) $niveau;
    if ($niveau >= 1 && $niveau <= 100)
    {
      $this->_niveau = $niveau;
    }
  }
  public function setExperience($experience)
  {
    $experience = (int) $experience;
    if ($experience >= 1 && $experience <= 100)
    {
      $this->_experience = $experience;
    }
  }
}
?>
```

Le même code, en Java donnerait :

Un Getter

```
public String[] getNom() {
```

```
        return this.nom;
    }
```

Un Setter

```
public void setNom(String[] nom) {
        this.nom = nom;
    }
```

Créer une classe qui hérite d'une autre classe

Dans cet exemple, on crée une classe Archer qui hérite de la classe personnnage. Ici, les archers ont leurs dégats initialisés à 10. Et leur nom contient le préfixe 'Archer '.

```
class Archer extends Personnage{

    public function __construct($force, $nom){
        parent::__construct($force, $nom);
        $this->setDegats(10);
    }

    public function setNom($cible){
        parent::setNom($cible);
        $this->_nom = 'Archer '.$this->nom();
    }

}
```

Exercices

Exercice 1 - Calculette peu performante

Cet exercice a pour but de voir si vous avez compris les bases avant de passer à la suite

On vous demande ici de créer une classe MyCalculator toute simple qui prend 2 nombres en arguments. Vous devrez créer une méthode afficher la somme de ces deux nombres, une autre pour afficher la multiplication de ces deux nombres.

On doit donc obtenir quelque chose qui resemble à :

```
<?php
class MyCalculator{
// Le code de ma classe
}

$mycalc = new MyCalculator(12, 6);
echo $mycalc->add()."<br/>"; // Affiche 18
```

```
echo $mycalc->multiply()."<br/>"; // Affiche 72
?>
```

Exercice 2 - Comptes Bancaires

Introduction

Le but de cet exercice est de modéliser le système bancaire de manière assez simple. Ce système bancaire fait par vos soins devra comprendre plusieurs banques, permettre à des gens d'ouvrir un compte dans une banque de leur choix, de pouvoir faire des transactions bancaires ainsi que de pouvoir ajouter ou retirer de l'argent de leur compte.

Exigences

Votre code devra comprendre au début 3 classes :

- La classe Banque qui contient au moins les attributs nom, nombre de clients;
- La classe Client qui contient au moins les attributs nom, prenom, compte;
- La classe Compte qui contient au moins les attributs client, banque, montant;

On veut que:

1. Un client puisse créer un compte dans la banque de son choix
2. Un client puisse ajouter ou retirer de l'argent de son compte bancaire
3. Un client puisse faire un virement à un autre client (Pour simplifier chaque client possède un seul compte à une seule banque)
4. Un client ne puisse pas être en négatif
5. Lorsqu'une transaction a lieu entre deux clients de banques différentes, il y ait 10% de frais supplémentaires pour celui qui fait le virement

Nous vous suggérons d'implémenter ces fonctionnalités une par une et de les tester avant de passer à la suivante. Vous mettrez tout les attibuts en privé et utiliserez des getters et setters pour y accéder et les modifier depuis les autres classes. Les méthodes et attributs seront nommées en CamelCase (ex: NomDeLaMéthode).

Bonus: Vous pouvez ajouter une classe Devise et modifier les autres classes pour permettre a des clients d'ouvrir des comptes dans plusieurs devises et faire des transaction entre comptes avec des devises différentes (Attention aux conversions !)

Format attendu

Pour pouvoir tester facilement votre code, il va falloir respecter certaines notations pour utiliser ce fichier:

test2.php

```
<?php
```

```php
include("Le nom du fichier avec les classes"); // Pour que le fichier test ait accès à vos classes

// Question 1
$client = new Client("Jean", "Loutre");
$banque = new Banque("KSI");
echo $banque->getNom()." a ".$banque->getNombreClients()." clients<br>";  // KSI a 0 clients
$banque->createCompte($client);
echo $banque->getNom()." a ".$banque->getNombreClients()." clients<br>";  // KSI a 1 clients
$compte = $client->getCompte();
echo $client->getNom()." ".$client->getPrenom()." possède un compte chez ".$compte->getBanque()->getNom()."<br>";  // Jean Loutre possède un compte chez KSI
echo "Ce client a ".$compte->getMontant()."€<br>";  // Ce client a 0€

// Question 2
$compte->ajouter(100);
echo "Ce client a ".$compte->getMontant()."€<br>";  // Ce client a 100€
$compte->retirer(25);
echo "Ce client a ".$compte->getMontant()."€<br>";  // Ce client a 75€

// Question 3
$client2 = new Client("Jeannot", "Loutre Junior");
$banque->createCompte($client2);
$client->virer($client2, 25);
echo $client->getNom()." ".$client->getPrenom()." possède ".$compte->getMontant()."€<br>";  // Jean Loutre possède 50€
echo $client2->getNom()." ".$client2->getPrenom()." possède ".$client2->getCompte()->getMontant()."€<br>";  // Jeannot Loutre Junior possède 25€

// Question 4
$compte->retirer(100); // Cette ligne doit renvoyer une alerte et ne pas réaliser la transaction
$client->virer($client2, 100);  // Cette ligne doit renvoyer une alerte et ne pas réaliser la transaction
echo $client->getNom()." ".$client->getPrenom()." possède ".$compte->getMontant()."€<br>";  // Jean Loutre possède 50€
echo $client2->getNom()." ".$client2->getPrenom()." possède ".$client2->getCompte()->getMontant()."€<br>";  // Jeannot Loutre Junior possède 25€

// Question 5
$banque2 = new Banque("FOCEEN");
$client3 = new Client("Jeanne", "Loutréa");
$banque2->createCompte($client3);
$client->virer($client3, 10);
echo $client->getNom()." ".$client->getPrenom()." possède ".$compte->getMontant()."€<br>";  // Jean Loutre possède 39€
```

```
echo $client3->getNom()." ".$client3->getPrenom()." possède
".$client3->getCompte()->getMontant()."€<br>";  // Jeanne Loutréa
possède 10€
```

Exercice 3 - Banque en ligne

Il est nécessaire d'avoir fait l'exercice 2 pour réaliser celui-ci.

L'objectif de cet exercice est de vous familiariser avec la notion d'héritage. Pour cela, il vous est demandé d'implémenter une classe BanqueEnLigne qui hérite de la classe Banque qui permettra de modéliser une banque en ligne.

On veut que :

1. La banque en ligne respecte le cahier des charges de la banque classique sans la partie concernant les transactions (cf question 3)
2. La banque en ligne doit en plus avoir un attribut 'url' donnant l'url de son site web.
3. Les transactions réalisées par les clients des banques en ligne n'engendrent aucun frais

Dans cet exercice, il est **interdit de modifier le code de l'exercice 2**. Les consignes et conseils de l'exercice 2 restent tous valides. Voici un fichier à utiliser pour tester votre code.

test3.php

```
<?php
include("Le nom du fichier avec les classes"); // Pour que le fichier
test ait accès à vos classes

// Cette série de test vérifie que l'ancien code fonctionne toujours
avec les banques en ligne
// Question 1.1 (Ancienne question 1)
$client = new Client("Jean", "Loutre");
$banque = new BanqueEnLigne("KSI");
echo $banque->getNom()." a ".$banque->getNombreClients()."
clients<br>";  // KSI a 0 clients
$banque->createCompte($client);
echo $banque->getNom()." a ".$banque->getNombreClients()."
clients<br>";  // KSI a 1 clients
$compte = $client->getCompte();
echo $client->getNom()." ".$client->getPrenom()." possède un compte
chez ".$compte->getBanque()->getNom()."<br>";  // Jean Loutre possède
un compte chez KSI
echo "Ce client a ".$compte->getMontant()."€<br>";  // Ce client a 0€

// Question 1.2 (Ancienne question 2)
$compte->ajouter(100);
echo "Ce client a ".$compte->getMontant()."€<br>";  // Ce client a 100€
$compte->retirer(25);
echo "Ce client a ".$compte->getMontant()."€<br>";  // Ce client a 75€
```

```php
// Question 1.3 (Ancienne question 3)
$client2 = new Client("Jeannot", "Loutre Junior");
$banque->createCompte($client2);
$client->virer($client2, 25);
echo $client->getNom()." ".$client->getPrenom()." possède
".$compte->getMontant()."€<br>";  // Jean Loutre possède 50€
echo $client2->getNom()." ".$client2->getPrenom()." possède
".$client2->getCompte()->getMontant()."€<br>";  // Jeannot Loutre
Junior possède 25€

// Question 1.4 (Ancienne question 4)
$compte->retirer(100); // Cette ligne doit renvoyer une alerte et ne
pas réaliser la transaction
$client->virer($client2, 100);  // Cette ligne doit renvoyer une alerte
et ne pas réaliser la transaction
echo $client->getNom()." ".$client->getPrenom()." possède
".$compte->getMontant()."€<br>";  // Jean Loutre possède 50€
echo $client2->getNom()." ".$client2->getPrenom()." possède
".$client2->getCompte()->getMontant()."€<br>";  // Jeannot Loutre
Junior possède 25€

// Question 2
$banque->setUrl("https://ksi-centrale-marseille.fr/");
echo "L'url de ".$banque->getNom()." est ".$banque->getUrl()."<br>";
// L'url de KSI est https://ksi-centrale-marseille.fr/

// Question 5.1
$banque2 = new Banque("FOCEEN");
$client3 = new Client("Jeanne", "Loutréa");
$banque2->createCompte($client3);
$client->virer($client3, 20);
echo $client->getNom()." ".$client->getPrenom()." possède
".$compte->getMontant()."€<br>";  // Jean Loutre possède 30€
echo $client3->getNom()." ".$client3->getPrenom()." possède
".$client3->getCompte()->getMontant()."€<br>";  // Jeanne Loutréa
possède 20€

// Question 5.2 (Ancienne question 5)
$client3->virer($client2, 10);
echo $client->getNom()." ".$client->getPrenom()." possède
".$compte->getMontant()."€<br>";  // Jean Loutre possède 40€
echo $client3->getNom()." ".$client3->getPrenom()." possède
".$client3->getCompte()->getMontant()."€<br>";  // Jeanne Loutréa
possède 9€
```

Compléments

Les héritages

L'héritage en POO (que ce soit en C++, Java ou autre langage utilisant la POO) est une technique très puissante et extrêmement pratique. Ce chapitre sur l'héritage est le chapitre à connaitre par cœur (ou du moins, le mieux possible). Pour être bien sûr que vous ayez compris le principe, un TP vous attend au prochain chapitre. ;)

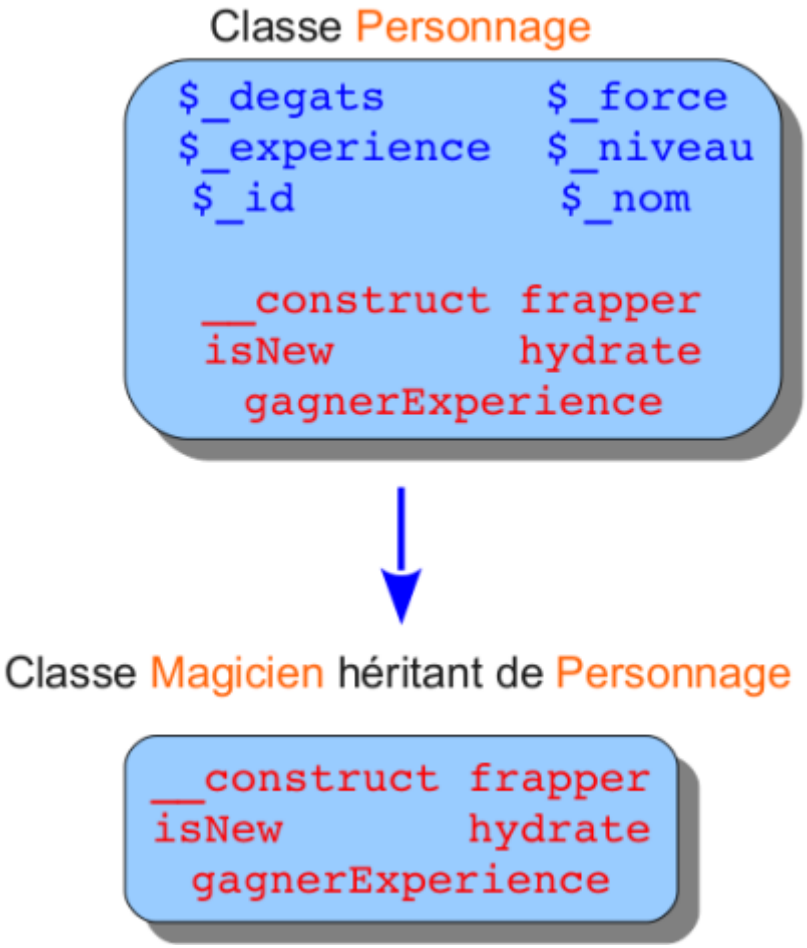
Allez, j'arrête de vous mettre la pression, allons-y ! Notion d'héritage Définition

Quand on parle d'héritage, c'est qu'on dit qu'une classe B hérite d'une classe A. La classe A est donc considérée comme la classe mère et la classe B est considérée comme la classe fille.

Concrètement, l'héritage, c'est quoi ?

Lorsqu'on dit que la classe B hérite de la classe A, c'est que la classe B hérite de tous les attributs et méthodes de la classe A. Si l'on déclare des méthodes dans la classe A, et qu'on crée une instance de la classe B, alors on pourra appeler n'importe quelle méthode déclarée dans la classe A du moment qu'elle est publique.

Schématiquement, une classe B héritant d'une classe A peut être représentée comme ceci (figure suivante).

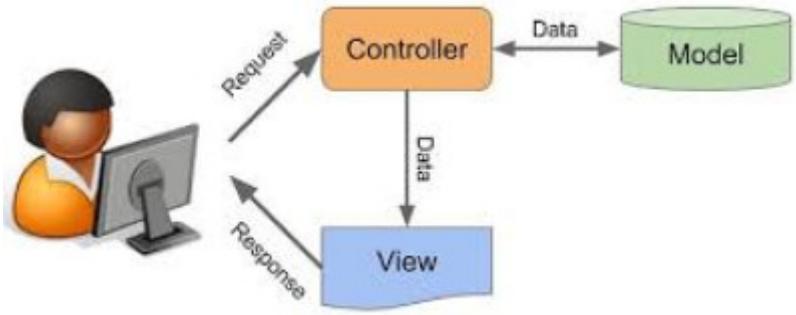


Quand est-ce que je sais si telle classe doit hériter d'une autre ?

Soit deux classes A et B. Pour qu'un héritage soit possible, il faut que vous puissiez dire que A est un B. Par exemple, un magicien est un personnage, donc héritage. Un chien est un animal, donc héritage

aussi. Bref, vous avez compris le principe. ;)

La Modèle-Vue-Contrôleur (MVC)



Un des plus célèbres design patterns s'appelle MVC, qui signifie Modèle - Vue - Contrôleur. C'est celui que nous allons découvrir maintenant.

Le pattern MVC permet de bien organiser son code source. Il va vous aider à savoir quels fichiers créer, mais surtout à définir leur rôle. Le but de MVC est justement de séparer la logique du code en trois parties que l'on retrouve dans des fichiers distincts.

Modèle : cette partie gère les données de votre site. Son rôle est d'aller récupérer les informations « brutes » dans la base de données, de les organiser et de les assembler pour qu'elles puissent ensuite être traitées par le contrôleur. On y trouve donc entre autres les requêtes SQL.

Vue : cette partie se concentre sur l'affichage. Elle ne fait presque aucun calcul et se contente de récupérer des variables pour savoir ce qu'elle doit afficher. On y trouve essentiellement du code HTML mais aussi quelques boucles et conditions PHP très simples, pour afficher par exemple une liste de messages.

Contrôleur : cette partie gère la logique du code qui prend des décisions. C'est en quelque sorte l'intermédiaire entre le modèle et la vue : le contrôleur va demander au modèle les données, les analyser, prendre des décisions et renvoyer le texte à afficher à la vue. Le contrôleur contient exclusivement du PHP. C'est notamment lui qui détermine si le visiteur a le droit de voir la page ou non (gestion des droits d'accès).

Remarque : Cette forma a été fait à la va-vite et est donc temporairement grandement inspirée de la génialissime et exhaustive formation sur la POO d'OpenClassroom : https://openclassrooms.com/fr/courses/1665806-programmez-en-oriente-objet-en-php/1666060-utiliser-la-classe

---

Cette formation a été réalisée par Alexandre Menasria, Romain GRONDIN et Ahmad IKSI.

From:
https://wiki.centrale-med.fr/ginfo/ - **Wiki GInfo**

Permanent link:
https://wiki.centrale-med.fr/ginfo/formations:devweb_5

Last update: **23/11/2021 19:31**