eVOGUE

Liam TARDIEU

www.evogue.fr

SUPPORT DE COURS / PHP – ORIENTE OBJET

Sommaire

➢ Présentation

HISTORIQUE

La notion d'objet a été introduite avec le langage de programmation Simula, créé à Oslo entre 1962 et 1967 dans le but de faciliter la programmation de logiciels de simulation.
Avec ce langage de programmation, les caractéristiques et les comportements des objets à simuler sont décrits dans le code source.
Le langage de programmation orientée objet Smalltalk a été créé par le centre de recherche Xerox en 1972.
La programmation orientée objet est devenue populaire en 1983 avec la sortie du langage de programmation C++, un langage orienté objet, dont l'utilisation ressemble volontairement au populaire langage C.

UNE AUTRE TECHNIQUE DE DEVELOPPEMENT, POURQUOI ?

Au cours des 35 dernières années, les concepteurs de matériel informatique sont passés des machines de la taille d'un hangar à des ordinateurs portables légers basés sur de minuscules microprocesseurs.
Au cours des mêmes années, les développeurs de logiciels sont passés de l'écriture de programmes en assembleur et en COBOL à l'écriture de programmes encore plus grands en C et C++. Nous pourrons parler de progrès (bien que cela soit discutable), mais il est clair que le monde du logiciel ne progresse pas aussi vite que celui du matériel. Qu'ont donc les développeurs de matériel que les développeurs de logiciels n'ont pas ?
La réponse est donnée par les composants. Si les ingénieurs en matériel électronique devaient partir d'un tas de sable à chaque fois qu'ils conçoivent un nouveau dispositif, si leur première étape devait toujours consister à extraire le silicium pour fabriquer des circuits intégrés, ils ne progresseraient pas bien vite. Or, un concepteur de matériel construit toujours un système à partir de composants préparés, chacun chargé d'une fonction particulière et fournissant un ensemble de services à travers des interfaces définies. La tâche des concepteurs de matériel est considérablement simplifiée par le travail de leurs prédécesseurs.
La réutilisation est aussi une voie vers la création de meilleurs logiciels. Aujourd'hui encore, les développeurs de logiciels en sont toujours à partir d'une certaine forme de sable et à suivre les mêmes étapes que les centaines de programmeurs qui les ont précédés. Le résultat est souvent excellent, mais il pourrait être amélioré. La création de nouvelles applications à partir de composants existants, déjà testés, a toutes chances de produire un code plus fiable. De plus, elle peut se révéler nettement plus rapide et plus économique, ce qui n'est pas moins important.

A QUELS BESOINS REPOND LA PROGRAMMATION ORIENTEE OBJET ?

Le problème qui se pose quand nous avons besoin d'élaborer des programmes complexes, c'est que nous sommes très vite confrontés aux problèmes suivants :

- comment comprendre et réutiliser les programmes faits par d'autres ?
- comment réutiliser les programmes que vous avez écrits il y a plusieurs mois et dont vous avez oublié le fonctionnement ?
- comment "cloner" rapidement des programmes déjà faits pour des applications légèrement différentes ?
- comment programmer simplement des actions simples sur des éléments variés et complexes ?
- comment ajouter des fonctions sans tout réécrire et tout retester ?

L'approche traditionnelle du développement, dite fonctionnelle, marche bien tant que le développeur sait où il va : le client qui a fourni un cahier des charges parfait et complet, ne changera pas d'avis, et une fois que le logiciel sera fini, nous n'y toucherons plus...

Dans ce cas, nous pouvons décomposer les programmes en "fonctions" : un processus global, et des sous processus ou fonctions qui créent un arbre décomposant le problème en une série d'actions (c'est une décomposition "algorithmique")

Sauf que ces programmes idéaux n'existent pas : le client change d'avis, les logiciels évoluent, et le développeur doit vraiment savoir où il va, surtout en cas de travail en équipe, car il ne s'agit pas de "découvrir" d'autres éléments en cours de route.

Or, avec une approche fonctionnelle, tout changement de spécification peut avoir des conséquences catastrophiques... Il faut parfois tout refaire de A à Z...

Pour résoudre ces problèmes, il faut développer trois stratégies :

- modéliser différemment
- modulariser
- encapsuler

Un bon modèle doit réunir deux qualités :

- faciliter la compréhension du programme étudié, en réduisant la complexité
- permettre de simuler le comportement du programme

AVANTAGES DE LA PROGRAMMATION ORIENTEE OBJET

La POO (programmation orientée objet) est une forme particulière de programmation destinée à faciliter la maintenance et la réutilisation / adaptation de vos scripts PHP.

L'idéal c'est qu'un développeur (de classe) puisse transmettre son code avec une documentation et qu'un autre développeur (utilisateur de ces classes) puisse les utiliser sans avoir à replonger de manière approfondie dans le code de chacune d'entre elles. Ainsi le développeur qui le suit ne réinvente pas la roue. Il utilise l'existant et développe seulement ce qu'il a besoin et qui n'existe pas.

Si une fonctionnalité approche son besoin sans vraiment y répondre, il peut interagir avec cette dernière afin de modifier légèrement son comportement pour répondre aux besoins du moment.

Le principal apport de la programmation objet est certainement de faciliter la réutilisation de modules.
Ecrire une classe spécialisée à partir d'une classe générale est le meilleur moyen de réutiliser ce code déjà produit.

Masquer la complexité du code est un autre fondement de la POO, lire « *panier->ajout(…)* » est quand même plus compréhensible qu'une série de conditions *if* et *else* dans une *boucle*.

La force de la programmation objet, c'est qu'elle s'appuie sur un modèle calqué sur la réalité physique du monde. Les objets se comportent comme des entités indépendantes, autosuffisantes qui collaborent par échange de messages.

Nous pouvons aussi raisonner du particulier au général (généraliser) grâce à la notion de classes d'objets, qui permet de partager entre les objets de la même classe des propriétés et des comportements...

Un objet, dans ce modèle, est défini comme une "entité" qui a des propriétés, et un comportement.

- La POO encourage le travail collaboratif.
- La POO simplifie la maintenance.
- La POO assouplit le code.

INCONVENIENTS DE LA PROGRAMMATION ORIENTEE OBJET

L'approche objet est moins intuitive que l'approche fonctionnelle. Malgré les apparences, il est plus naturel pour l'esprit humain de décomposer un problème informatique sous forme d'une hiérarchie de fonctions et de données, qu'en termes d'objets et d'interaction entre ces objets.

La POO oblige à réfléchir et à modéliser avant de programmer.

UNIFIED MODELING LANGUAGE

UML (en anglais Unified Modeling Language ou « langage de modélisation unifié ») est un langage de modélisation graphique à base de pictogrammes. Il est apparu dans le monde du génie logiciel, dans le cadre de la « conception orientée objet ». Couramment utilisé dans les projets logiciels, il peut être appliqué à toutes sortes de systèmes ne se limitant pas au domaine informatique.

Dans les concepts de base de l'approche objet nous dictons comment modéliser la structure objet d'un système de manière pertinente. Quels moyens devons-nous alors utiliser pour mener une analyse qui respecte les concepts objets ? Sans un cadre méthodologique approprié, la dérive fonctionnelle de la conception est inévitable.

Pour remédier à ces inconvénients majeurs de l'approche objet, il nous faut donc :

- un langage (pour s'exprimer clairement à l'aide des concepts objets), qui doit permettre de représenter des concepts abstraits (graphiquement par exemple),
- limiter les ambiguïtés (parler un langage commun, au vocabulaire précis, indépendant des langages orientés objets),
- faciliter l'analyse (simplifier la comparaison et l'évaluation de solutions).
- une démarche d'analyse et de conception objet, pour ne pas effectuer une analyse fonctionnelle et se contenter d'une implémentation objet, mais penser objet dès le départ,
- définir les vues qui permettent de décrire tous les aspects d'un système avec des concepts objets.

En d'autres termes : il faut disposer d'un outil qui donne une dimension méthodologique à l'approche objet et qui permette de mieux maîtriser sa richesse.

LANGAGE PHP ET PROGRAMMATION ORIENTEE OBJET

Versions

Beaucoup de langages implémentent la POO, parfois de manière optionnelle, comme le C ou le PHP.

PHP est un langage impératif disposant depuis la version 5 de fonctionnalités de modèle objet complètes.

La version actuelle de PHP est la version 5, sortie le 13 juillet 2004.

Différence entre php4 et php5

PHP5, la version actuelle du PHP a supprimé des fonctions venant de PHP4, en a implémenté des nouvelles et apporte une nouvelle conception de codage avec la POO (programmation orientée objet) mais il n'est pas obligatoire de développer en POO avec php5, il est naturellement possible de continuer de développer en procédurale.

➢ Classes et Objets

QU'EST-CE QU'UN OBJET ?

Un objet est un conteneur symbolique, qui possède sa propre existence et englobe des caractéristiques, états et comportements et qui, par métaphore, représente quelque chose de tangible du monde réel manipulé par informatique.
En programmation orientée objet, un objet est créé à partir d'un modèle appelé classe, duquel il hérite les comportements et les caractéristiques. Les comportements et les caractéristiques sont typiquement basés sur celles propres aux choses qui ont inspiré l'objet : une personne, un dossier, un produit, une lampe, une chaise, un bureau. Tout peut être objet.

QU'EST-CE QU'UNE CLASSE

Les classes sont présentes pour « fabriquer » des objets.
En programmation orientée objet, un objet est créé sur le modèle de la classe à laquelle il appartient.

Exemple

Prenons l'exemple le plus simple du monde : les gâteaux et leur moule. Le moule, il est unique. Il peut produire une quantité infinie de gâteaux. Dans ces cas-là, les gâteaux sont les objets et le moule est la classe. La classe est présente pour produire des objets. Elle contient le plan de fabrication d'un objet et nous pouvons nous en servir autant que nous le souhaitons afin d'obtenir une infinité d'objets.

Autre Exemple

Etant donné qu'une classe est le modèle de quelque chose que nous voudrions construire.
Le plan de construction d'une maison qui réunit les instructions destinées à la construction est donc une classe. Cependant le plan n'est pas une maison.
La maison est un objet qui a été « fabriqué » à partir de la classe (le plan).
A partir du plan (la classe) nous pouvons construire une ou plusieurs maisons (l'objet).

Le vocabulaire diffère légèrement en Programmation Orientée Objet, nous ne parlerons plus de fonctions mais de méthodes, nous ne parlerons plus de variables mais d'attributs (ou propriétés).

Une classe est une entité regroupant un certain nombre d'attributs et de méthodes que tout objet issu de cette classe possèdera.

DIFFERENCE ENTRE UNE CLASSE ET UN OBJET

- La classe est un plan, une description de l'objet. Sur le plan de construction d'une voiture nous y retrouverons le moteur ou encore la couleur de la carrosserie.
- L'objet est une application concrète du plan. L'objet est la voiture. Nous pouvons créer plusieurs voitures basées sur un plan de construction.

Nous pouvons donc créer plusieurs objets à partir d'une classe.

APPLICATION CONCRETE DE LA CLASSE SUR L'OBJET

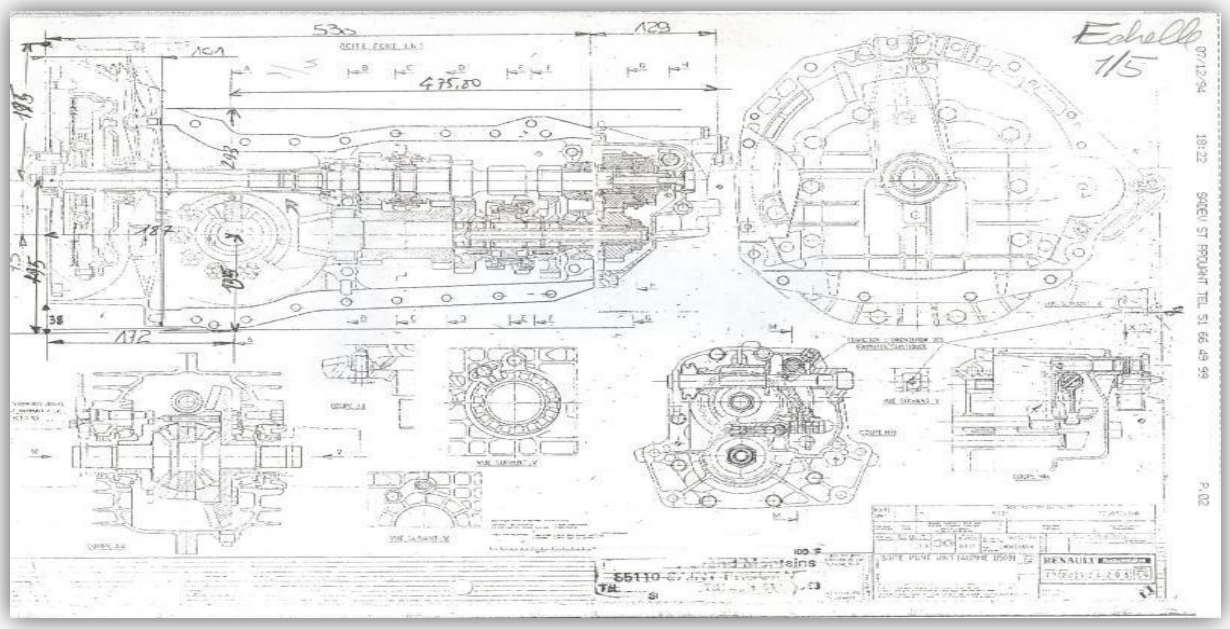
Définition

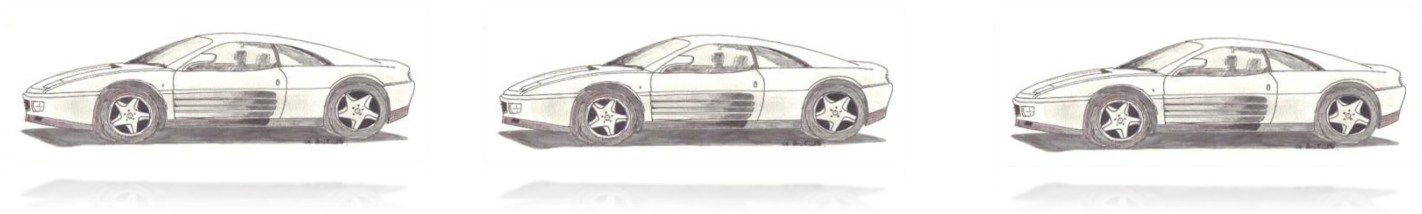
Une instance de classe est le fait de créer un objet de cette classe.

Une fois l'objet créé, nous pourrons appeler les méthodes qui composent notre classe.

Exemple

Classe : Plan de construction d'une voiture





Objet (instance de la classe) | Objet (instance de la classe) | Objet (instance de la classe)

Représentation

Nous pouvons donc imaginer les méthodes et les attributs de nos objets.

Les méthodes pourraient être : démarrer(), rouler(), s'arrêter(), se_ravitailler(), etc.

Les attributs pourraient être : couleur, poids, taille, etc.

Le plan de construction est unique, il peut produire une quantité infinie de voitures.

➢ Création d'un objet

INTRODUCTION

La création d'un objet permet plusieurs choses :

- La création d'un espace mémoire est réservée pour recevoir toutes les propriétés durant son utilisation.
- L'attribution d'un nom à l'objet.
- Si l'objet possède un constructeur, cet objet sera initialisé (avec des valeurs par défaut ou des valeurs passées en paramètres).

INSTANCIATION

« new » peut être traduit comme « fabrique-moi un nouvel objet émanant de cette classe ».

Quand vous « instanciez » une classe, la variable stockant l'objet ne stocke en fait pas l'objet lui-même, mais un identifiant qui représente cet objet. C'est-à-dire en faisant :

Exemple

```
$objet = new MaClasse;
```

$objet ne contient pas l'objet lui-même, mais son identifiant unique.
Notez que si le constructeur de la classe ne comporte pas des arguments, il n'est pas nécessaire d'utiliser des parenthèses. Cependant, il peut être préférable de toujours les mettre afin de n'avoir jamais d'erreurs.

INFERENCE

Une autre manière de générer l'objet est par inférence, c'est-à-dire par déduction à partir du contexte. Vu le fait que PHP permet de créer des variables $obj sans une déclaration préalable nous pouvons créer l'objet par le simple fait d'utiliser une variable dans un contexte d'objet (en appliquant l'opérateur -> à une variable suffit que cette variable contienne un objet même si jamais la classe correspondante n'a été définie (ceci n'est pas valable pour ajouter des méthodes !).

Exemple

```
$obj-> age = 25;
print_r($obj);
```

$obj représente un objet.

TRANSFORMATION

Il est tout à fait possible de générer l'objet par la transformation (casting) d'un tableau associatif. Lorsqu'un tableau devient un objet, $obj, tous les éléments du tableau indexés par une chaîne de caractères deviennent des propriétés (en cas d'indexés par chaîne vide ils sont cachés pour l'objet).

Exemple

```
$tab['nom'] = "julien";
$tab['pays'] = "Suisse";
$obj = (object) $tab;
print_r($obj);
```

$obj représente un objet.

CLASSE PAR DEFAUT

Pour les objets générés par inférence et transformation (casting) aucune classe n'a été définie.

Ils appartiennent à la classe stdClass du PHP. Elle est la classe mère racine de PHP.

Les objets qui sont « orphelins » de classe se retrouvent dans cette classe par défaut de Php servant à les accueillir.

➢ Encapsulation

DEFINITION

L'encapsulation en général est la notion de mettre une chose dans une autre. En imageant, nous pouvons voir que cette chose est mise dans une capsule.
En programmation, l'encapsulation de données est l'idée de cacher l'information.

POURQUOI ?

Sur un site web, si nous devions créer un espace membre, un blog, un forum ou autres.
Il ne serait pas forcément judicieux de tout créer de A à Z, pour au moins deux raisons :

- Il faut du temps pour créer ces fonctionnalités, les tester et les rendre fiables.
- Cela a déjà été fait par d'autres développeurs à plusieurs reprises.

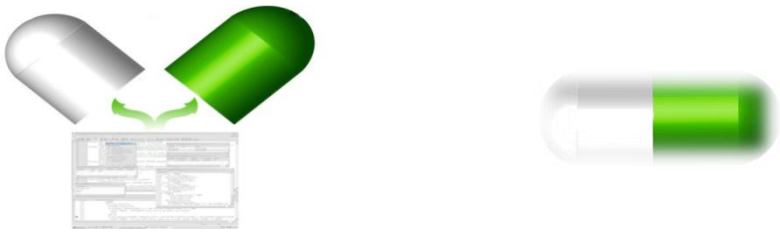
CONCEVOIR POUR REUTILISER...COMMENT FAIRE ?

Voici le code « imagé » permettant de créer une boutique en ligne :

Si nous voulons que notre code soit réutilisable, cela force les autres développeurs à lire et à comprendre entièrement notre code avant qu'ils parviennent à le modifier.
Ils risquent d'être réticents à l'utiliser car il est souvent plus difficile de comprendre un code écrit par autrui que créer son propre code.

Notre tentative de diffuser un « code type » réutilisable par d'autres pour créer une e-boutique sera un échec inévitable.

C'est précisément au moment de la conception que nous allons décider d'encapsuler le code « compliqué » dans des capsules (qui sont en réalité informatiques : des objets) :



Notre capsule est bien évidemment une métaphore, en POO nous dirons qu'il s'agit d'un objet.
L'un des avantages de la POO est que l'on peut masquer la complexité du code à celui qui l'utilise.

Dans cet exemple nous distinguons deux sortes de développeurs :
Groupe « 1 » Les développeurs de classes/objets permettant de mettre au point la boutique en ligne.
Groupe « 2 » Les développeurs utilisateurs de cette classe profitant de cette boutique en ligne et pouvant ajouter d'autres fonctionnalités éventuelles.

INTERET DE L'ENCAPSULATION

Les développeurs du groupe « 1 » (de la classe) ont englobé dans celle-ci un code qui peut être assez complexe et il est donc inutile voire risqué de laisser les développeurs du groupe « 2 » manipuler ces objets sans aucune restriction. Il est important d'interdire aux développeurs du groupe « 2 » de modifier directement les attributs d'un objet.

Sur notre exemple de voiture :
Le conducteur n'a pas à forcer la porte de sa voiture pour l'ouvrir, il lui suffit d'appuyer sur un bouton et elle se déverrouille.
Le conducteur ne sait pas avec précision pourquoi l'appui sur le bouton déclenche le déverrouillage des portes, cela serait inutile. Son rôle est de déverrouiller la porte de la voiture afin de s'y introduire et la conduire.

Les développeurs du groupe « 2 » doivent se contenter d'invoquer les méthodes dont ils ont besoin sans avoir une connaissance portant sur la totalité du code de la classe. Exemple : *panier->ajout_panier(…)*.
Les développeurs peuvent ajouter des produits au panier en utilisant une méthode explicite sans se soucier du code mis en œuvre pour y parvenir.

Cela leur permet de ne pas avoir à lire chaque ligne de code ainsi ils n'auront pas les inconvénients (complexité du code) mais uniquement les avantages (e-boutique fonctionnelle) dont ils pourront se servir.

Pour instaurer cette contrainte, les attributs auront un niveau de visibilité.

NIVEAU DE VISIBILITE

- **Publique** (public) : est la visibilité la plus large (accessible directement en dehors de la classe).
  - Un attribut ou une méthode déclarée comme publique les rendent accessibles de partout, autant depuis l'intérieur de l'objet mais aussi depuis l'extérieur.

- **Privée** (private): est la plus restrictive (accessible uniquement depuis la classe courante).
  - Seule la classe elle-même peut accéder à une méthode ou à un attribut.

- **Protégée** (protected) : est intermédiaire.
  - Les méthodes et attributs seront visibles par la classe courante, toutes les classes filles, mais inaccessibles en dehors des classes mères/filles.

➢ Manipulation d'objets

LA PSEUDO-VARIABLE $THIS

$this est une variable réservée qui désigne l'objet en cours, elle le représente.
Nous l'utilisons exclusivement à l'intérieur de la classe pour appeler une méthode ou un attribut de l'objet.
Afin d'avoir accès à un attribut ou à une méthode d'un objet depuis l'intérieur de la classe, la variable $this est accompagnée de l'opérateur ->.
Ce qui signifie « dans cet objet, je veux tel attribut » ou « dans cet objet, je veux telle méthode ».
L'utilisation de $this-> doit se faire pour accéder à un attribut ou à une méthode depuis l'intérieur de la classe uniquement.

ACCESSEUR (GETTER) / MUTATEURS (SETTER)

Les accesseurs (Get) et les manipulateurs (Set) :

- les accesseurs permettent de récupérer la valeur de données privées sans y accéder directement de l'extérieur,
- les manipulateurs (ou mutateurs) permettent de changer l'état de données en vérifiant si la valeur que nous voulons apporter à la donnée respecte les normes de celle-ci.

Accesseurs

Si l'un de nos attributs possède un type de visibilité « private » nous ne pourrons les afficher de l'extérieur.
Nous allons donc déclarer des méthodes avec la visibilité « public » qui afficheront la valeur de la propriété ayant la visibilité « private ».

Mutateurs

Si l'un de nos attributs possède un type de visibilité « private » nous ne pourrons les modifier de l'extérieur.
Nous allons donc déclarer des méthodes avec la visibilité « public » qui réaliseront quelques tests (si nécessaire) et qui, en cas de succès, attribueront la valeur à la propriété avec la visibilité « private ».
Il est bien plus logique que les tests qui permettront de mener à l'affectation éventuelle d'une valeur à cette propriété en cas de succès se réalisent dans la méthode et non directement sur la page web.
Cela nous donnera l'avantage de contrôler l'assignation d'une valeur à un attribut et ainsi éviter d'y retrouver n'importe quoi, c'est d'autant plus intéressant avec php qui ne type pas ses variables.

Cela ne casse pas l'encapsulation, cela l'enrichit dans le sens où il ne s'agit plus d'un "tout accès" ou d’un "sans accès", mais d'une multitude de possibilités intermédiaires qui peuvent être définies dans l'accesseur.
Un accès public à un attribut permet au développeur d'y affecter n'importe quelle valeur, tandis qu'un accès restreint à l’attribut forcera le développeur à passer par un set pour lui affecter une valeur et cela permettra entre-temps de contrôler et d'assurer le type, le format, la taille, etc.

Convention

Ces méthodes s’appellent des « accesseurs » ou des « mutateurs » et commencent par convention par le mot get ou set.

➢ Manipulation de classes

CONSTANTES

Définition

Une constante déclarée à l'intérieur d'une classe est un attribut appartenant à la classe dont la valeur ne change jamais. Nous n'aurons donc pas à préciser « static » devant les constantes.

Déclaration

```
const nom ;
```

Le mot-clé const est un mot réservé pour déclarer une constante à l'intérieur d'une classe.
Une constante appartient à la classe, et non à un quelconque objet.

Différence entre const et define

Const est le mot-clé permettant de définir des constantes à l'intérieur des classes alors que define permet de définir des constantes globales (en dehors des classes).

ELEMENTS STATIC

Définition

Elément appartenant à la classe et non à l'objet.
Nous pouvons par exemple définir des méthodes ou des attributs en static, ils appartiendront donc à la classe.

Déclaration

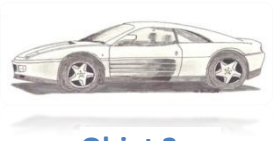
```
Public static function nom() { }
Public static $var ;
```

APPARTENANCE A LA CLASSE OU A L'OBJET ?

Appartenance à l'objet

```
Class voiture{    private $couleur ;  Public set_couleur($arg){    // …   } }
$objet1 = new voiture() ;
$objet2 = new voiture() ;
```

Objet 1



Objet 2

Nous créons deux objets « voiture ».

Nous allons changer la valeur de l'attribut couleur d'un objet « voiture » (en rouge).

```
$objet2->set_couleur(« rouge ») ;
```

Objet 1

Objet 2

Appartenance à la classe

Reprenons le même exemple en faisant appartenir la propriété couleur à la classe (en static) :

Objet 1

Objet 2

Nous créons deux objets « voiture ».
Nous allons changer la valeur de l'attribut couleur de la classe (plan) « voiture » (en bleu)
Toutes les prochaines voitures seront bleues.

Objet 3

Objet 4

Conclusion

Quand une propriété appartient à l'objet et que nous la modifions, cette modification s'applique sur l'objet uniquement.

Quand une propriété appartient à la classe et que nous la modifions, cela sera répercuté sur tous les prochains objets créés émanant de cette classe.

OPERATEUR DE RESOLUTION DE PORTEE

L'opérateur de résolution de portée :: sera utilisé pour appeler des éléments appartenant à telle classe et non à tel objet.

Exemple

```
Echo Voiture::couleur // si la propriété est « public ».
```

DIFFERENCE ENTRE SELF:: ET $THIS->

- self représente la classe à l'intérieur de la classe
- $this représente l'objet « courant » à l'intérieur de la classe

Par conséquent nous ne pourrons pas accéder à un attribut appartenant à la classe (statique) avec $this mais avec self.
Nous ne pourrons pas non plus accéder à un attribut appartenant à l'objet (non statique) avec self mais nous le ferons avec $this.

- L'opérateur -> : cet opérateur permet d'accéder à un élément de tel objet
- L'opérateur :: : cet opérateur permet d'accéder à un élément de telle classe

Nous verrons donc :

- $this-> et ~~non $this::~~
- Self:: et ~~non self ->~~

➢ Héritage

CONCEPT

L'héritage est un concept puissant de la programmation orientée objet, permettant entre autres la réutilisation (décomposition du système en composants) et l'adaptabilité des objets grâce au polymorphisme. Elle se nomme ainsi car le principe est en quelque sorte le même que celui d'un arbre généalogique. Ce principe est basé sur des classes dont les « filles » héritent des caractéristiques de leur(s) « mère(s) ».

Chaque classe possède des caractéristiques (attributs et méthodes) qui lui sont propres. Lorsqu'une classe fille hérite d'une classe mère, elle peut alors utiliser ses caractéristiques.

- Si la classe mère est abstraite il y a de fortes chances qu'elle présente des caractéristiques abstraites. Pour être effective (« instanciable »), la classe fille doit alors les définir, sinon elle sera elle-même abstraite.
- Si la classe mère est effective (toutes les caractéristiques sont définies) alors la classe fille est également effective. Il est possible d'y ajouter des caractéristiques, d'utiliser les caractéristiques héritées (appartenant aux parents) et de redéfinir les méthodes héritées. Généralement cette redéfinition se fait par surcharge sémantique (on déclare de nouveau la méthode avec le même nom et la même signature).

FONCTIONNEMENT

Quand nous parlons d'héritage, l'exemple type est le suivant :
Lorsqu'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.
La classe B hérite de tous les attributs et méthodes de la classe A. Si nous déclarons des méthodes dans la classe A, et que nous créons une instance de la classe B, alors nous pourrons appeler n'importe quelle méthode déclarée dans la classe A, du moment qu'elle est publique (public ou protégée (protected).

RELATION ET NOTION D'HERITAGE

- Transitivité : si B hérite de A et si C hérite de B alors C hérite de A ;
- Non réflexif : une classe ne peut hériter d'elle même ;
- Non symétrique : si A hérite de B, B n'hérite pas de A ;
- Sans cycle : Il n'est pas possible que B hérite de A, C hérite de B et que A hérite de C.

EFFECTUER UN HERITAGE, DANS QUELS CAS ?

Prenons deux classes A et B.

Pour qu'un héritage soit possible, il faut que vous puissiez dire que A est un B.

Autrement dit :

- une voiture est un véhicule (la classe voiture hérite de la classe véhicule)
- une moto est un véhicule (la classe moto hérite de la classe véhicule)
- un éléphant est un animal (la classe éléphant hérite de la classe animal)

- Une classe mère peut avoir plusieurs classes filles
- Une classe fille ne peut avoir plusieurs classes mères.

Déclaration

Pour faire hériter une classe d'une autre, vous aurez besoin du mot-clé « extends »

```
Class animal
{
	Public function deplacement()
	{
		return "je me deplace";
	 }
}
Class elephant extends animal
{
	Public function manger()
	{
		return "je mange de l'herbe";
	}
}
```

Ainsi la classe éléphant obtient toutes les méthodes et tous les attributs déclarés dans la classe animal.

Il est donc possible de faire appel à la méthode « deplacement() » de animal via un objet éléphant :

```
$eleph = new Elephant();
echo $eleph->deplacement(); // affiche :  je me déplace
```

SURCHARGER UNE METHODE

Appelé aussi « override ».

Lors d'un héritage, surcharger une méthode permet d'utiliser une fonction existante de la classe mère dans la classe fille tout en modifiant le comportement initialement prévu de celle-ci.

Exemple

```
class A
{
	protected function calcul()
	{
		return 10;
	}
}
class B extends A
{
	public function calcul()
	{
		$retour = parent::calcul();
		if($retour > 100) return "$retour est supérieur à 100";
		else return "$retour est inférieur à 100";
	}
}
$objet_b = new B();
echo $objet_b->calcul();
```

Lors d'une surcharge, dans la méthode calcul() de la classe B vous ne pourrez utiliser $this->calcul() ; en pensant appeler la méthode calcul() de la classe A car c'est comme si la méthode s'appelait elle-même et cela aboutirait à un échec.

Le mot-clé parent fonctionne pour appeler une méthode d'une classe mère lors d'une surcharge dans la classe fille.

Lorsqu'une méthode appartient à la classe (méthode static par exemple), la classe fille qui en hérite peut également se servir de parents pour accéder à la méthode de la classe mère.

➢ Abstraction

DEFINITION

Le concept d'abstraction identifie et regroupe des caractéristiques et traitements communs applicables à des entités ou concepts variés ; une représentation abstraite commune de tels objets permet d'en simplifier la manipulation.

PRINCIPE ET UTILISATION

Une classe abstraite n'est pas « instanciable » et ne pourra donc pas « fabriquer » d'objets.
Nous pouvons considérer que lorsqu'une classe hérite d'une classe abstraite, il s'agirait du plan d'un autre plan.

CLASSES ABSTRAITES

Imaginons que nous ayons une classe Animal et des classes : Chien, Chat, Loup, Eléphant, Chameau.
Tous sont des animaux et héritent naturellement de la classe « Animal ».
Cependant aucun d'entre eux n'est « juste animal », un chien est un chien, et un chat est un chat.
Par conséquent la classe Animal ne sera jamais « instanciée », la classe Animale sera abstraite.
Elle servira de classe modèle pour celles qui en hériteront. En effet chaque animal dort, chaque animal mange…
Ces méthodes « dormir() » ou encore « manger() » seront relatives à la classe Animal.
En revanche chaque animal a un cri différent : le chien aboie, le chat miaule.
Ces méthodes cri() seront dans les classes filles.

Exemple

```
abstract class Animal
{
	public function manger()
	{
		return "je mange";
	}
	abstract public function cri();
}
class Chien extends Animal
{
	public function cri()
	{
		return "moi, j'abboie";
	}
}
```

Mot-clé : abstract.

METHODES ABSTRAITES

Nous avons tout de même ajouté la méthode cri() dans la classe Animal pour forcer les classes héritières à écrire cette méthode, la méthode cri() est abstraite.

INFORMATIONS

- Pour définir une méthode comme étant abstraite, il faut que la classe elle-même soit abstraite.
- Une classe abstraite n'est pas forcément uniquement composée de méthodes abstraites
- Le fait de déclarer une classe abstraite permet d'empêcher son « instanciation ».
- Le fait de déclarer une méthode abstraite permet de forcer les classes filles à les réécrire.
- Les méthodes abstraites n'ont pas de corps

➢ Finalisation

DEFINITION

Le mot-clé final a des sens légèrement différents suivant le contexte, mais en général il signifie « Cela ne peut pas changer ». Vous pourriez vouloir empêcher les changements pour deux raisons : conception ou efficacité.

CLASSES FINALES

Quand nous disons qu'une classe entière est finale (en faisant précéder sa définition par le mot-clé final) on stipule que nous ne voulons pas hériter de cette classe ou permettre à qui que ce soit de le faire. En d'autres mots, soit la conception de cette classe est telle que nous n'aurons jamais besoin de la modifier, soit pour des raisons de sûreté ou de sécurité nous ne voulons pas qu'elle soit sous-classée (héritée). Ou alors, nous pouvons avoir à faire à un problème d'efficacité, et nous voulons nous assurer que toute activité impliquant des objets de cette classe sera aussi efficace que possible. Une classe finale est donc une classe qui ne peut pas être héritée.

METHODES FINALES

Les méthodes finales ont deux raisons d'être. La première est de mettre un « verrou » sur la méthode pour empêcher toute sous-classe de la redéfinir. Ceci est fait pour des raisons de conception quand nous voulons être sûrs que le comportement d'une méthode est préservé durant l'héritage et ne peut pas être redéfini.
Toutes les méthodes privées sont implicitement finales. Parce que nous ne pouvons pas accéder à une méthode privée, nous ne pouvons pas la surcharger. Nous pouvons ajouter le mot-clé final à une méthode privée, mais cela n'apporte rien de plus.
Une méthode finale est donc une méthode qui ne peut pas être surchargée.

Mot-clé : final.

➢ Méthodes magiques

INTRODUCTION

Les méthodes magiques sont des outils très pratiques pour simplifier le code et automatiser certaines tâches.
Elles sont appelées automatiquement lorsque certaines actions sont effectuées et permettent d'éviter certaines erreurs.

__CONSTRUCT

Définition

Le rôle du constructeur est d'accompagner la construction de la classe.
Le constructeur va précisément s'exécuter au moment ou la classe va être « instanciée » et l'objet créé.
On pourra plus tard y retrouver l'initialisation d'attributs ou une connexion à la base de données car le constructeur est exécuté dès la création de l'objet.

Déclaration

Le constructeur d'une classe prend le nom suivant : __construct
Une méthode ayant le même nom que la classe est considérée comme étant le constructeur (précédente convention de PHP 4). Toutefois il est préférable d'utiliser la méthode magique __construct.

Information

Quand vous « instanciéz » une classe et que vous lui donnez un argument :

```
$obj = new Maclass("Liam");
```

Cet argument va directement au constructeur de la classe qui est en train d'être « instanciée ».
Il implique donc que le constructeur de cette classe ait été déclaré :

Exemple

```
Class Maclass{
public function __construct($arg) {    echo "Bonjour $arg";   }
}
```

__DESTRUCT

Information

A l'inverse, le destructeur est appelé à la fin d'exécution de votre script.

Il peut, par exemple, fermer la connexion à la base de données.

__SET

Information

Cette méthode prend deux arguments et se lance uniquement si nous tentons d'accéder une valeur à un attribut et que cela n'est pas possible. Par exemple pour un attribut privé (private) ou protégé (protected) en dehors de la classe ou un attribut qui n'existe pas.

Exemple

```
public function __set($nom, $valeur){
        if(property_exists($this, $nom)) {   $this->$nom = $valeur;   }
        else {  echo "La propriété $nom N'existe PAS, la valeur $valeur ne peut donc pas être
attribué"; }
}
```

__GET

Information

Cette méthode prend un argument et se lance uniquement si nous tentons d'accéder une valeur à un attribut et que cela n'est pas possible.

Par exemple pour un attribut privé (private) ou protégé (protected) en dehors de la classe ou un attribut qui n'existe pas.

Exemple

```
public function __get($nom){
                if(property_exists($this, $nom)) {     echo $this->$nom . "<br />";     }
                else{   echo "La propriété $nom N'existe PAS, Vous ne pouvez donc pas l'afficher";  }
}
```

__CALL

Information

La méthode __call est appelée lorsque nous essayons d'invoquer une méthode privée ou inexistante. Elle prend deux arguments : le premier est le nom de la méthode que nous avons essayé d'appeler et le second est la liste des arguments qui lui ont été passés (sous forme de tableau).

Exemple

```
public function __call($nom, $arguments)
{
        echo "la méthode $nom à été appelé mais elle n'existe pas ! <br />";
        echo "Ses arguments étaient: " . implode($arguments, " - ");
}
```

La méthode magique __callStatic agit exactement comme __call mais lors de l'appel statique d'une méthode inexistante.

__TOSTRING

Information

La méthode magique __toString est appelée lorsque l'objet est amené à être converti en chaîne de caractères.
Après un « cast » en « string » (chaine de caractère), il sera envisageable d'obtenir un affichage :

Exemple

```
class MaClasse{
    private $texte;
    public function __construct ($texte)  {   $this->texte = $texte;   }
    public function __toString()   {   return $this->texte;    }
}

  $obj = new MaClasse ('bonjour');
  $texte = (string) $obj ;
  echo $obj; // Affiche : bonjour
```

__ISSET

Information

Cette méthode, comme son nom l'indique, est lancée en appelant lors d'un test isset() sur un des éléments composant l'objet.

Exemple

```
public function __isset($nom)
{
        echo "Vérification de $nom<br />";
}
```

__SLEEP

Information

La fonction serialize() vérifie si votre classe a une fonction avec le nom magique __sleep. Si c'est le cas, cette fonction sera exécutée avant toute linéarisation. Elle peut nettoyer l'objet et elle est supposée retourner un tableau avec les noms de toutes les variables de l'objet qui doivent être « linéarisées ». Le but avoué de __sleep est de valider des données en attente ou d'effectuer les opérations de nettoyage. De plus, cette fonction est utile si vous avez de très gros objets qui n'ont pas besoin d'être sauvegardés en totalité.

__WAKEUP

Information

Réciproquement, la fonction unserialize() vérifie la présence d'une fonction dont le nom est le nom magique __wakeup. Si elle est présente, cette fonction peut reconstruire toute ressource que l'objet possède.

Le but avoué de __wakeup est de rétablir toute connexion base de données qui aurait été perdue durant la linéarisation et d'effectuer des tâches de réinitialisation

__UNSET

Information

Cette méthode est appelée lorsque unset() est appelé sur des éléments de l'objet.

__INVOKE

Information

La méthode __invoke est appelée lorsque le script tente d'appeler un objet comme une fonction.

__SETSTATE

Information

Cette méthode statique est appelée pour les classes exportées par la fonction var_export().
Le seul paramètre de cette méthode est un tableau contenant les propriétés exportées sous la forme array('propriété' => valeur, ...).

__CLONE

Information

La méthode magique clone se lance lors d'un clonage et s'il y a des instructions, ceux-ci ne s'appliquent que sur l'objet cloné. Elle ne peut pas être appelée directement, elle réagit à l'appel de la fonction *clone $object*.

Exemple

```
class Maclasse1
{
	public $attribut1;
	public $attribut2;
	public function __clone()
	{
		$this->attribut2 = "coucou";
	}
}

$a = new Maclasse1();
$b = $a;
$a->attribut1 = "bonjour";
echo "Objet A: " . $a->attribut1 . "<br />";		// Affiche : Bonjour
echo "Objet B: " .  $b->attribut1 . "<br />";		// Affiche : Bonjour
echo  "<hr />";
$origine = new Maclasse1();
$origine->attribut1 = "bonsoir";
$copie = clone $origine;
echo "Objet Origine: " . $origine->attribut1 . "<br />"; // Affiche : Bonsoir
echo "Objet copie: " . $copie->attribut1 . "<br />"; // Affiche : Bonsoir
echo "Objet Origine: " . $origine->attribut2 . "<br />"; // n'affiche rien
echo "Objet copie: " . $copie->attribut2 . "<br />"; // Affiche : coucou
```

$a et $b sont des références qui pointent vers le même objet. Ils représentent le même objet.

$origine et $copie sont deux objets différents

➢ Comparaison

LES OPERATEURS

Exemple

```
if ($objet1 == $objet2)
   echo '$objet1 et $objet2 sont identiques !';
else
   echo '$objet1 et $objet2 sont différents !';
```

Ce test retourne true si les deux objets sont identiques (mêmes attributs, mêmes valeurs) et surtout qu'ils soient des instances issu de la même classe.
Si l'objet 1 sort de la classe A et que l'objet 2 sort de la classe B, la condition renverra false.

L'opérateur **===** permet de vérifier que deux objets sont strictement identiques.
Avec la présence du triple égale, la condition renverra true si les deux objets font référence à la même instance. Il vérifiera donc que les deux identifiants d'objets comparés sont les mêmes.

Différence == et === sur les objets

- == permet de voir si deux objets sortent de la même classe
- === permet de voir si deux objets ont le même identifiant et pointent vers le même objet.

Instanceof

L'opérateur instanceof nous permettra de vérifier si un objet est une instance d'une classe en particulier.

Exemple

```
class D { }
$objet = new D;
if ($objet instanceof D)     echo '$objet est une instance de D<br />';
else      echo '$objet n'est pas une instance de D<br />';
```

➢ Interfaces

INTRODUCTION

Les interfaces sont des classes particulières qui autorisent d'autres classes à hériter d'un certain comportement, et qui définissent des méthodes à implémenter. Cela vous permet de créer du code qui spécifie quelles méthodes une classe peut implémenter, sans avoir à définir comment ces méthodes seront gérées.

Les interfaces servent à créer des comportements génériques ; si plusieurs classes doivent obéir à un comportement particulier, nous créons une interface décrivant ce comportement, nous la faisons implémenter par les classes qui en ont besoin. Ces classes devront ainsi obéir strictement aux méthodes de l'interface (nombre, type et ordre des paramètres, type des exceptions), sans quoi la compilation ne se fera pas.

Grâce aux interfaces, le code devient donc plus flexible tout en étant plus strict, et le programme suit constamment le schéma de conception défini au début du projet.

PRINCIPE ET UTILISATION

Exemple

```
interface PointCommun
{
    public function deplacement($dest);
}
class Bateau implements PointCommun
  {
        public function deplacement($dest)
        {
                echo "je navigue";
        }
  }
class Avion implements PointCommun
{
        public function deplacement($dest)
         {
                echo "je vol";
         }
  }
```

Un bateau et un avion ne se déplacent pas de la même manière pourtant ils ont ce point commun : ils se déplacent tous les deux.

Les méthodes déclarées dans une interface ne peuvent être de type privé étant donné que nous y définissons les méthodes que nous pourrons appeler lors de « l'instanciation ».

Informations

Une interface est une classe entièrement abstraite et une des caractéristiques principales des interfaces est d'obliger les classes qui les implémentent à créer toutes leurs méthodes. Si ce n'est pas le cas, une erreur sera générée et c'est là toute la puissance d'une interface car elle nous assure que la classe contient bien les méthodes qu'elle doit implémenter.

Les méthodes d'une interface n'ont pas de corps et une interface ne peut être « instanciée ».
Son rôle est de décrire un comportement à notre objet.

L'implémentation se fait de la même manière que l'héritage, sauf que nous utilisons le mot-clé "implements".

Les classes peuvent implémenter plus d'une interface en séparant chaque interface par une virgule.

DIFFERENCE HERITAGE ET IMPLEMENTATION

Classes abstraites et Interfaces : A ne pas confondre.

L'héritage représente un sous-ensemble, par exemple :

- Un chien est un animal (Chien hérite d'Animal)
- Un Airbus est un avion (Airbus hérite d'Avion)

Un avion et un animal n'ont pas de raison d'hériter de la même classe, cependant les deux peuvent se déplacer, il est possible de mettre en place une interface possédant ce point commun.

Héritage ou implémentation ?

- ***Implements*** (en français : met en œuvre) permet d'implémenter une interface dans une classe, toutes les méthodes de l'interface doivent être redéclarées dans la classe.
- ***Extends*** permet de faire hériter deux classes ou deux interfaces. (Contrairement aux classes, l'héritage multiple entres interfaces est possible en les séparant par des virgules).

Une classe ne peut implémenter deux interfaces qui partagent des noms de fonctions, puisque cela causerait une ambiguïté. Lors d'un héritage entre interfaces (via extends) les méthodes de l'interface « mère » ne doivent pas être redéclarées dans l'interface « fille ».
Une interface ne peut pas posséder de méthodes abstraites ou finales et ne peut pas avoir le même nom qu'une classe, et vice-versa.

Mot-clé : interface.

➢ Traits

INTRODUCTION

Un Trait est un type abstrait, « simple modèle conceptuel pour structurer des programmes orientés objets ».

Cette fonctionnalité permet ainsi de repousser les limites de l'héritage simple puisque une classe ne peut héritée que d'une seule classe à la fois. En revanche elle peut importer (donc hérité) de plusieurs traits.

Les traits ne sont pas non plus comparables aux interfaces dans la mesure où les méthodes d'interface n'ont pas de corps. Plus simplement, un trait est un regroupement de méthodes pouvant être importées dans une classe.

UTILISATION

Exemple

```
trait Pays{
        public function afficher_pays()  {  return 'France';   }
}
trait Capitale{
        public function afficher_capitale()  {  return 'Paris';  }
}
class A{ use Capitale, Pays ; }
class B{ use Capitale; }
$a = new A;
echo $a->afficher_capitale(); // Affiche « Paris »

$b = new b;
echo $b->afficher_capitale(); // Affiche aussi « Paris »
```

FONCTIONNEMENT

« use » est le mot clé permettant d'utiliser un trait dans une classe.

Les traits peuvent eux aussi utiliser des traits.

➢ Design Pattern

INTRODUCTION

En informatique, et plus particulièrement en développement logiciel, un patron de conception « design pattern » est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel. Il s'agit d'un concept destiné à résoudre les problèmes récurrents suivant le paradigme objet décrivant une solution standard, utilisable dans la conception de différents logiciels.

On distingue trois familles de patrons de conception selon leur utilisation :

- de construction : ils définissent comment faire l'instanciation et la configuration des classes et des objets.
- structuraux : ils définissent comment organiser les classes d'un programme dans une structure plus large (séparant l'interface de l'implémentation).
- comportementaux : ils définissent comment organiser les objets pour que ceux-ci collaborent (distribution des responsabilités) et expliquent le fonctionnement des algorithmes impliqués.

TYPE : CREATION

- Fabrique abstraite (Abstract Factory)
- Monteur (Builder)
- Fabrique (Factory Method)
- Prototype (Prototype)
- Singleton (Singleton)

TYPE : STRUCTURE

- Adaptateur (Adapter)
- Pont (Bridge)
- Objet composite (Composite)
- Décorateur (Decorator)
- Façade (Facade)
- Poids-mouche ou poids-plume (Flyweight)
- Proxy (Proxy)

TYPE : COMPORTEMENT

- Chaîne de responsabilité (Chain of responsibility)
- Commande (Command)
- Interpréteur (Interpreter)
- Itérateur (Iterator)
- Médiateur (Mediator)
- Mémento (Memento)
- Observateur (Observer)
- État (State)
- Stratégie (Strategy)
- Patron de méthode (Template Method)
- Visiteur (Visitor)
- Fonction de rappel (Callback)

PRESENTATION DU PATTERN : SINGLETON

Le Singleton, en programmation orientée objet, répond à la problématique de n'avoir qu'une seule et unique instance d'une même classe dans un programme. Par exemple, dans le cadre d'une application web dynamique, la connexion au serveur de bases de données est unique. Afin de préserver cette unicité, il est judicieux d'avoir recours à un objet qui adopte la forme d'un singleton.

Exemple

```
class Singleton
{
	private static $instance = null;
	public $numero;
	private function __construct() {}
	public static function creation()
	{
		if(is_null(self::$instance))
		{
			self::$instance = new Singleton();
		}
		return self::$instance ;
	}
}
$a = Singleton::creation(); // A est la référence #1 qui représente l'objet en RAM
$b = Singleton::creation(); // B est la référence #1 qui représente l'objet en RAM
$c = Singleton::creation(); // C est la référence #1 qui représente l'objet en RAM
$c->numero = 18; // j'affecte une propriété
echo $a->numero . '<hr />'; // 18 - et tout les représentant de l'objet ont cette valeur (preuve qu'il s'agit bien du même objet.
echo $b->numero . '<hr />'; // 18
echo $c->numero . '<hr />'; // 18
```

La classe n'est pas « instanciable » de l'extérieur (*$s = New Singleton; : donne une erreur*) puisque le constructeur est déclaré volontairement comme étant privé.
Nous exécutons la méthode « creation() » de la classe Singleton, la première fois l'instance possède la valeur « null », nous créons donc un objet Singleton à l'intérieur de la classe qui est retourné ensuite.
$a représente un objet issue de la classe Singleton avec la référence #1 en mémoire.
La seconde fois, l'instance n'est pas null, nous ne rentrons pas dans le IF et l'objet est retourné immédiatement.
$a, $b et $c représente un objet issue de la classe Singleton avec la même référence #1 en mémoire.

De cette manière, il est impossible de créer plusieurs objets de type Singleton.

Nous en déduisons donc que ces trois variables référencent (pointent) le même objet en mémoire.

Nous avons donc réussi à créer une instance unique de la classe via Singleton.

Un singleton est composé de 3 caractéristiques :

- Un attribut privé et statique qui conservera l'instance unique de la classe.
- Un constructeur privé afin d'empêcher la création d'objet depuis l'extérieur de la classe
- Une méthode statique qui permet soit d'instancier la classe soit de retourner l'unique instance créée.

➢ Namespace

INTRODUCTION

Dans leur définition la plus large, les espaces de noms « namespace » représentent un moyen d'encapsuler des éléments. Cela peut être conçu comme un concept abstrait, à différents endroits. Par exemple, dans un système de fichiers, les dossiers représentent un groupe de fichiers associés, et sert d'espace de noms pour les fichiers qu'il contient. Un exemple concret est que le fichier cv.doc peut exister dans les deux dossiers /home/julien et /home/other, mais que les deux copies de cv.doc ne peuvent pas coexister dans le même dossier. De plus, pour accéder au fichier cv.doc depuis l'extérieur du dossier /home/ julien, il faut préciser le nom du dossier en utilisant un séparateur de dossier, tel que /home/julien/cv.doc. Le même principe est appliqué pour les espaces de noms dans le monde de la programmation.

AVANTAGES

Les espaces de noms sont conçus pour résoudre deux problèmes que les auteurs de bibliothèques et applications rencontrent lors de la réutilisation d'éléments tels que des classes ou des bibliothèques de fonctions :

- Collisions de noms entre le code que vous créez, les classes, fonctions ou constantes internes de PHP, ou celle de bibliothèques tierces.
- La capacité de faire des alias ou de raccourcir des Noms_Extrememement_Long pour aider à la résolution du premier problème, et améliorer la lisibilité du code.

Les espaces de noms PHP fournissent un moyen pour regrouper des classes, interfaces, fonctions ou constantes. Voici un exemple de syntaxe des espaces de noms PHP.

L'encapsulation d'un ensemble est particulièrement utilisée lors de l'utilisation de classe d'une librairie ou tout simplement pour classer ses fonctions.
Typiquement, si deux bibliothèques définissent toutes deux une classe nommée « MyClass », et que nous utilisons ces deux librairies, nous aurons une erreur du type : Fatal Error.

EXEMPLE ET UTILISATION

Prenons comme exemple un fichier nommé : ***declaration.php***

```
namespace A ;
        function affichage_ville() {   return "Paris";  }

namespace B ;
        function affichage_ville()   {   return "Paris";       }
```

Prenons un second fichier nommé : ***execution.php***

```
require_once("declaration.php");
echo A\ affichage_ville();
echo B\ affichage_ville();
```

Remarques

Nous avons déclaré une fonction « affichage_ville() » dans le namespace A et une autre (précisément du même nom) dans le namespace B.
Lors de l'appel il faut d'abord indiquer le nom du namespace suivi du signe « \ » et du nom de la fonction à exécuter.

Le rôle d'un namespace est de contenir plusieurs fonctions relatives à une catégorie, un sujet, un contexte. Les fonctions déclarées hors d'un namespace font partie de l'espace de nom global.

Dans l'espace de nom global, nous n'aurions pas pu créer deux fonctions du même nom, ni lancer l'exécution de deux fonctions nommées de la même manière.
En les définissants dans des espaces de noms « namespace » cela nous permet de les classer et ceci est très pratique lors d'un travail collaboratif en équipe pour éviter une erreur lorsqu'une fonction est déclarée avec un nom déjà existant sur un projet.

Il est préférable de ne pas avoir deux fonctions du même nom, cependant les espaces de noms « namespace » offrent la possibilité d'avoir deux fonctions du même nom si elles ne sont pas déclarées dans le même espace. Cela peut être utile selon le contexte.

➢ Exceptions

INTRODUCTION

Jusqu'à présent nous avons défini les classes qui permettaient de créer nos objets, il est également possible de se servir de classes prédéfinies, la classe Exception est l'une d'entre-elles, elle est native et par conséquent chargée automatiquement par PHP.
Lorsqu'une exception se produit, l'exécution normale du programme est interrompue et l'exception est traitée.

Tout programme en exécution peut être sujet à des erreurs pour lesquelles des stratégies de détection et de réparation sont possibles. Ces erreurs ne sont pas des bugs mais des conditions particulières (ou conditions exceptionnelles, ou exceptions) dans le déroulement normal d'une partie d'un programme.

Par exemple, l'absence d'un fichier utile n'est pas un bogue du programme ; par contre, ne pas gérer son absence en provoquerait un.
Les exceptions permettent donc de simplifier, personnaliser et d'organiser la gestion des « erreurs » dans un programme informatique. Ici le mot « erreurs » ne signifie pas « bug », qui est un comportement anormal de l'application développée, mais plutôt « cas exceptionnel » à traiter différemment dans le déroulement du programme.

Disposer dans un langage de programmation d'une syntaxe pour différencier l'exécution normale de l'exécution dans un contexte exceptionnel peut être utile pour le traitement des situations exceptionnelles.

Le traitement d'une situation exceptionnelle peut nécessiter de revenir « dans le passé » de l'exécution du programme, c'est-à-dire remonter brutalement la chaîne d'appels pour annuler une opération fautive, ou encore modifier les valeurs de certaines variables, puis reprendre l'exécution du programme un peu avant le site de l'erreur. D'où le besoin d'associer, à la syntaxe spéciale, des opérateurs spéciaux pour effectuer des sauts et des modifications de variables à des points arbitraires de la chaîne d'appels.

FONCTIONNEMENT

Une exception peut être lancée ("throw") et attrapée ("catch") dans PHP. Le code devra être entouré d'un bloc try pour faciliter la saisie d'une exception potentielle. Chaque try doit avoir au moins un bloc catch correspondant. Plusieurs blocs catch peuvent être utilisés pour attraper différentes classes d'exceptions. L'exécution normale (lorsqu'aucune exception n'est lancée dans le bloc try ou lorsqu'un catch correspondant à l'exception lancée n'est pas présent) continue après le dernier bloc catch défini dans la séquence. Les exceptions peuvent être lancées (ou relancées) dans un bloc catch. Lorsqu'une exception est lancée, le code suivant le traitement ne sera pas exécuté et PHP tentera de trouver le premier bloc catch correspondant

L'objet lancé doit être une instance de la classe Exception.

Exception est la classe de base pour toutes les exceptions.

LA CLASSE EXCEPTION

Propriétés et méthodes de la classe Exception

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

- Exception::__construct — Construit l'exception
- Exception::getMessage — Récupère le message de l'exception
- Exception::getPrevious — Retourne l'exception précédente
- Exception::getCode — Récupère le code de l'exception
- Exception::getFile — Récupère le fichier dans lequel l'exception est survenue
- Exception::getLine — Récupère la ligne dans laquelle l'exception est survenue
- Exception::getTrace — Récupère la trace de la pile
- Exception::getTraceAsString — Récupère la trace de la pile en tant que chaîne
- Exception::__toString — Représente l'exception sous la forme d'une chaîne
- Exception::__clone — Clone l'exception

DEROULEMENT

Le bloc try « essaie » d'exécuter le script entre les deux premières accolades. Si une exception est lancée dans ce bloc, elle est immédiatement « attrapée » dans le bloc catch() et les traitements particuliers sont exécutés à la place.

Exemple

```
function addition ($a, $b)
{
   if (!is_numeric ($a) OR !is_numeric ($b))
      throw new Exception ('Les deux paramètres doivent être des nombres');
   return $a + $b;
}
try
{
   echo addition (12, 3), '<br />';
   echo addition ('azerty', 55), '<br />';
   echo addition (4, 8);
}
catch (Exception $e)
{
   echo 'Une exception a été lancée. Message d\'erreur : ', $e->getMessage() . "<br />";
}
```

Try joue son rôle de « bloc d'essai », Catch quant à lui a le rôle de « bloc de capture ».
Try et Catch permettent d'avoir deux blocs d'instructions distinctes.
Le fait de filtrer les exceptions permet d'éviter une erreur avant qu'elle se produise.

➢ Php Data Objects

INTRODUCTION

Tout comme la classe Exception, la classe PDO est une classe prédéfinie, elle est native et chargée automatiquement par PHP.
L'extension n'est pas toujours activée dans PHP5, par défaut, dans ce cas il faut donc décommenter la ligne de "php_pdo.dll" dans le fichier php.ini.

PDO (PHP Data Objects) est l'extension définissant l'interface pour accéder à une base de données depuis PHP.
Peu importe le SGBD utilisé, le code est unique, ce qui permet un "déplacement" rapide, puisqu'il faut juste changer les arguments envoyés au constructeur.

PDO constitue une couche d'abstraction. Elle intervient entre le serveur d'application et le serveur de base de données. La couche d'abstraction permet de séparer le traitement de la base de données. Ainsi nous pouvons migrer vers un autre SGBD sans pour autant changer le code déjà développé.

Pour récupérer les enregistrements d'une table de la base de données, la méthode classique en PHP consiste à parcourir cette table ligne par ligne en procédant à des allers-retours entre le serveur d'application et le serveur de base de données. Ceci risque d'alourdir le traitement surtout si les deux serveurs sont installés chacun sur une machine différente. PDO remédie à ce problème en permettant de récupérer en une seule reprise tous les enregistrements de la table sous forme d'une variable PHP de type tableau à deux dimensions ce qui réduit visiblement le temps de traitement.

REQUETES

Choix	Type	Description	Valeur de retour
Exec()	INSERT UPDATE DELETE	Exec() est utilisé pour la formulation de requêtes ne retournant pas de résultat. exec() renvoie le nombre de lignes affectées par la requête.	Echec : *False (boolean)* Succès : *1 (int)* En cas de succès la valeur est 1 si cela a affecté 1 enregistrement seulement. Ce nombre peut naturellement être supérieur tout dépend du nombre d'enregistrements affectés par la requête.
Query()	SELECT	Au contraire d'exec(), Query() est utilisé pour la formulation de requêtes retournant un ou plusieurs résultats.	Echec : *False (boolean)* Succès : *PDOStatement (object)*
Prepare() puis Execute()	INSERT UPDATE DELETE SELECT	prepare() permet de préparer la requête mais ne l'exécute pas. Execute() permet d'exécuter une requête préparée. Ceci est à préconiser si vous exécutez plusieurs fois la même requête et ainsi vouloir éviter le cycle : Annalyse / Interpretation / Exécution	Prepare() renvoie toujours un *PDOStatement (object)*. Execute() : Echec : *False (boolean)* Succès : *True (boolean)*

Toutes ces méthodes peuvent prendre des arguments si nécessaire.

REQUETES PREPAREES

La plupart des bases de données supportent le concept des requêtes préparées. Qu'est-ce donc ? Vous pouvez les voir comme une sorte de modèle compilé pour le SQL que vous voulez exécuter, qui peut être personnalisé en utilisant des variables en guise de paramètres. Les requêtes préparées offrent deux fonctionnalités essentielles :

- La requête ne doit être analysée (ou préparée) qu'une seule fois, mais peut être exécutée plusieurs fois avec des paramètres identiques ou différents. Lorsque la requête est préparée, la base de données va analyser, compiler et optimiser son plan pour exécuter la requête. En utilisant les requêtes préparées, vous évitez ainsi de répéter le cycle analyser/compilation/optimisation. Pour résumer, les requêtes préparées utilisent moins de ressources et s'exécutent plus rapidement.

- Les paramètres pour préparer les requêtes n'ont pas besoin d'être entre guillemets ; le driver le gère pour vous. Si votre application utilise exclusivement les requêtes préparées, vous pouvez être sûr qu'aucune injection SQL n'est possible (Cependant, si vous construisez d'autres parties de la requête en vous basant sur des entrées utilisateurs, vous continuez à prendre un risque).

PASSAGE D'ARGUMENTS

Choix	Description
Array()	Reçoit les arguments sous forme d'un tableau de données sans qu'on puisse préciser le type de chaque donnée.
BindValue()	Reçoit le contenu d'une variable ou d'une chaîne. On utilise BindValue() pour transmettre une valeur à un marqueur.
BindParam()	BindParam() reçoit exclusivement le contenu d'une variable. On utilise BindParam pour transmettre une valeur à un marqueur.
BindColumn()	Numéro de la colonne (en commençant à 1) ou nom de la colonne dans le jeu de résultats. Si vous utilisez les noms de colonnes, assurez-vous que le nom corresponde à la casse de la colonne, comme retourné par le pilote.

BindValue() signifie "lier une valeur" tandis que BindParam() signifie "lier un paramètre".

CONSTANTES PRE-DEFINIES

Choix	Description
PDO::PARAM_BOOL	Représente le type de données booléen.
PDO::PARAM_NULL	Représente le type de données NULL SQL.
PDO::PARAM_INT	Représente le type de données INTEGER SQL.
PDO::PARAM_STR	Représente les types de données CHAR, VARCHAR ou les autres types de données sous forme de chaîne de caractères SQL.

Par exemple, PDO::PARAM_STR (pour une chaîne de caractères) permet de préciser le type d'une donnée, dans ce cas le serveur entoure la chaîne avec des quottes.

RECUPERATION DES DONNEES

Choix	Description
PDO::FETCH_ASSOC	Spécifie que la méthode de récupération doit retourner chaque ligne dans un tableau indexé par les noms des colonnes comme elles sont retournées dans le jeu de résultats correspondants. Si le jeu de résultats contient de multiples colonnes avec le même nom, PDO::FETCH_ASSOC retourne une seule valeur par nom de colonne.
PDO::FETCH_NAMED	Spécifie que la méthode de récupération doit retourner chaque ligne dans un tableau indexé par les noms des colonnes comme elles sont retournées dans le jeu de résultats correspondants. Si le jeu de résultats contient de multiples colonnes avec le même nom, PDO::FETCH_NAMED retourne un tableau de valeurs par nom de colonne.
PDO::FETCH_NUM	Spécifie que la méthode de récupération doit retourner chaque ligne dans un tableau indexé par le numéro des colonnes comme elles sont retournées dans le jeu de résultats correspondants, en commençant à 0.
PDO::FETCH_BOTH (par défaut)	Spécifie que la méthode de récupération doit retourner chaque ligne dans un tableau indexé par les noms des colonnes ainsi que leurs numéros, comme elles sont retournées dans le jeu de résultats correspondants, en commençant à 0.
PDO::FETCH_OBJ	Spécifie que la méthode de récupération doit retourner chaque ligne dans un objet avec les noms de propriétés correspondant aux noms des colonnes comme elles sont retournées dans le jeu de résultats.

MARQUEUR

Il existe deux types de marqueurs qui sont respectivement ? et les marqueurs nominatifs. Ces marqueurs ne sont pas « mélangeables » : donc pour une même requête, il faut choisir l'une ou l'autre des options.

DIFFERENCE ENTRE PDO ET PDOSTATEMENT

PDO

Représente une connexion entre PHP et un serveur de base de données.
On utilisera donc l'objet PDO pour formuler une requête, démarrer une transaction, obtenir des informations (exemple : obtenir le dernier identifiant inséré, etc.).

PDOSTATEMENT

Représente une requête, une fois exécutée, le jeu de résultats associés.
On utilisera l'objet PDOStatement pour préciser des arguments sur une requête préparée, pour traiter et exploiter les résultats, pour obtenir des informations (exemple : obtenir les métadonnées d'un champ sur une table sql).

LA CLASSE PDO

Représente une connexion entre PHP et un serveur de base de données.

Propriétés et méthodes

- PDO::beginTransaction — Démarre une transaction
- PDO::commit — Valide une transaction
- PDO::__construct — Crée une instance PDO qui représente une connexion à la base
- PDO::errorCode — Retourne le SQLSTATE associé avec la dernière opération sur la base de données
- PDO::errorInfo — Retourne les informations associées à l'erreur lors de la dernière opération sur la base de données
- PDO::exec — Exécute une requête SQL et retourne le nombre de lignes affectées
- PDO::getAttribute — Récupère un attribut d'une connexion à une base de données
- PDO::getAvailableDrivers — Retourne la liste des pilotes PDO disponibles
- PDO::inTransaction — Vérifie si nous sommes dans une transaction
- PDO::lastInsertId — Retourne l'identifiant de la dernière ligne insérée ou la valeur d'une séquence
- PDO::prepare — Prépare une requête à l'exécution et retourne un objet
- PDO::query — Exécute une requête SQL, retourne un jeu de résultats en tant qu'objet PDOStatement
- PDO::quote — Protège une chaîne pour l'utiliser dans une requête SQL PDO
- PDO::rollBack — Annule une transaction
- PDO::setAttribute — Configure un attribut PDO

LA CLASSE PDOSTATEMENT

Représente une requête, une fois exécutée, le jeu de résultats associés.

Propriétés et méthodes

- queryString — chaîne de caractères utilisée pour la requête.
- PDOStatement->bindColumn — Lie une colonne à une variable PHP
- PDOStatement->bindParam — Lie un paramètre à un nom de variable spécifique
- PDOStatement->bindValue — Associe une valeur à un paramètre
- PDOStatement->closeCursor — Ferme le curseur, permettant à la requête d'être de nouveau exécutée
- PDOStatement->columnCount — Retourne le nombre de colonnes dans le jeu de résultats
- PDOStatement->debugDumpParams — Détaille une commande préparée SQL
- PDOStatement->errorCode — Récupère le SQLSTATE associé lors de la dernière opération sur la requête
- PDOStatement->errorInfo — Récupère les informations sur l'erreur associée lors de la dernière opération sur la requête
- PDOStatement->execute — Exécute une requête préparée
- PDOStatement->fetch — Récupère la ligne suivante d'un jeu de résultats PDO
- PDOStatement->fetchAll — Retourne un tableau contenant toutes les lignes du jeu d'enregistrements
- PDOStatement->fetchColumn — Retourne une colonne depuis la ligne suivante d'un jeu de résultats
- PDOStatement->fetchObject — Récupère la prochaine ligne et la retourne en tant qu'objet
- PDOStatement->getAttribute — Récupère un attribut de requête
- PDOStatement->getColumnMeta — Retourne les métadonnées pour une colonne d'un jeu de résultats
- PDOStatement->nextRowset — Avance à la prochaine ligne de résultats d'un gestionnaire de lignes de résultats multiples
- PDOStatement->rowCount — Retourne le nombre de lignes affectées par le dernier appel à la fonction PDOStatement::execute()
- PDOStatement->setAttribute — Définit un attribut de requête
- PDOStatement->setFetchMode — Définit le mode de récupération par défaut pour cette requête

GESTION DES ERREURS

Exécuter une requête via l'objet PDO peut se faire dans un bloc try/catch.

Il est également possible de gérer les erreurs via la méthode setAttribute().

Exemple :

```
$pdo->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING );
```

Plus simplement, vous pouvez avoir accès à l'erreur d'une requête de la manière suivante :

Exec

```
$reponse = $pdo->exec('update employesssssssss set salaire="752" where id_employes="1111"');
print "<pre>";print_r($pdo->errorInfo());print "</pre>";
```

Nous interrogeons la méthode errorInfo() de l'objet PDO.

Query

```
$reponse = $pdo->query('select * frommm employes');
print "<pre>";print_r($pdo->errorInfo());print "</pre>";
```

Nous interrogeons la méthode errorInfo() de l'objet PDO.

Prepare, Execute

```
$reponse  = $pdo->prepare('SELECT ***** FROM employes');
$reponse->execute();
print "<pre>";print_r($reponse->errorInfo());print "</pre>";
```

Nous interrogeons la méthode errorInfo() de l'objet PDOStatement.

CHOIX DU CONNECTEUR DE BASE DE DONNEES

PHP 5 propose plusieurs connecteurs capables de se connecter à une base de données MySQL. Le premier et le plus courant reste le driver MySQL de base. C'est le plus simple à utiliser et donc le plus populaire. Il existe également l'extension MySQLi (MySQL improved) qui n'est autre qu'un driver amélioré de l'extension MySQL de base et qui a la particularité d'être orienté objet. Avec MySQLi, le développeur manipule directement un objet de type MySQLi alors qu'avec le driver MySQL de base il doit fonctionner par appels de procédures et fonctions.
Enfin, le dernier connecteur est l'extension PDO qui signifie PHP Data Objects. Cette extension permet de se connecter à de multiples bases de données à condition que les pilotes pour chaque système d'information soient installés sur le serveur Web. PDO a été intégré avec PHP 5 et a le principal avantage de faire bénéficier le développeur de certaines fonctionnalités de la base de données en fonction de son pilote. Au même titre que MySQLi, PDO est une interface orientée objet, ce qui est beaucoup plus pratique à utiliser lorsque nous sommes à l'aise avec de la programmation orientée objet.
Au fil du temps, les fonctions mysql_* et (ou plus généralement l'extension mysql) pourraient disparaître au profit de PDO et MYSQLI.

➢ Methodes Pratiques

DIVERS

- class_alias — Crée un alias de classe
- class_exists — Vérifie qu'une classe a été définie
- get_called_class — Le nom de la classe en "Late Static Binding"
- get_class_methods — Retourne les noms des méthodes d'une classe
- get_class_vars — Retourne les valeurs par défaut des propriétés d'une classe
- get_class — Retourne la classe d'un objet
- get_declared_classes — Liste toutes les classes définies dans PHP
- get_declared_interfaces — Retourne un tableau avec toutes les interfaces déclarées
- get_object_vars — Retourne les propriétés d'un objet
- get_parent_class — Retourne le nom de la classe d'un objet
- interface_exists — Vérifie si une interface a été définie
- is_a — Vérifie si l'objet fait partie d'une classe ou à cette classe comme parents
- is_subclass_of — Détermine si un objet est une sous-classe
- method_exists — Vérifie que la méthode existe pour une classe
- property_exists — Vérifie si un objet ou une classe possède une propriété

SPL_AUTOLOAD_REGISTER

Une fonction est particulièrement utile pour les classes, il s'agit de l'autoload().

Si cette fonction magique est déclarée dans vos scripts, alors toute classe « instanciée » mais n'ayant pas été chargée jusque-là, est chargée à l'aide de cette fonction.

Exemple

```
function inclus($classname)
{
        require $classname.'.class.php';
}
spl_autoload_register(inclus);
```

Avantages

- Cela évite d'écrire de multiples require/include
- Vous avez moins de cas particuliers à tester : à partir du moment où votre autoloading se fait correctement et que vous utilisez des classes existantes, vous limitez les risques d'inclusions multiples, ou d'oublis d'inclusions.
- Seul le code nécessaire est inclus ; PHP a donc moins de codes à interpréter — ce qui peut n'être que positif pour les performances de votre application, et la charge serveur qu'elle entrainera.