



Module 4

# Programmation Orientée Objet en PHP

# Contenu du module

- Introduction aux concepts de la programmation orientée objet.
- Les classes et les objets.
  - Définition de classes et instanciation.
- L'héritage de classes et le polymorphisme en PHP.
- Concepts avancés de programmation Objet.
  - Les interfaces.
  - Les espaces de noms (Namespaces).
- Gestion des erreurs avec les exceptions.

A grayscale photograph of a person in a business suit, with their hands cupped together in front of them. Overlaid on the image is a large, stylized white cloud with a thick white outline. A horizontal blue band cuts across the middle of the cloud, containing the text 'Concepts objets' in white.

# Concepts objets

# Historique

- Monde de la simulation
- Grammaires procédurales impropre à la modélisation des problèmes
- Besoin d'exprimer un environnement de façon naturelle
- Décomposition d'un sujet à comprendre analogue au fonctionnement de l'esprit humain :
  - Le "monde" est composé de "choses", "d'objets"
  - Ces objets interagissent entre eux
- Premiers langages objet : SIMULA, Smalltalk, C++, Eiffel

# Conception orientée objet : une approche descriptive

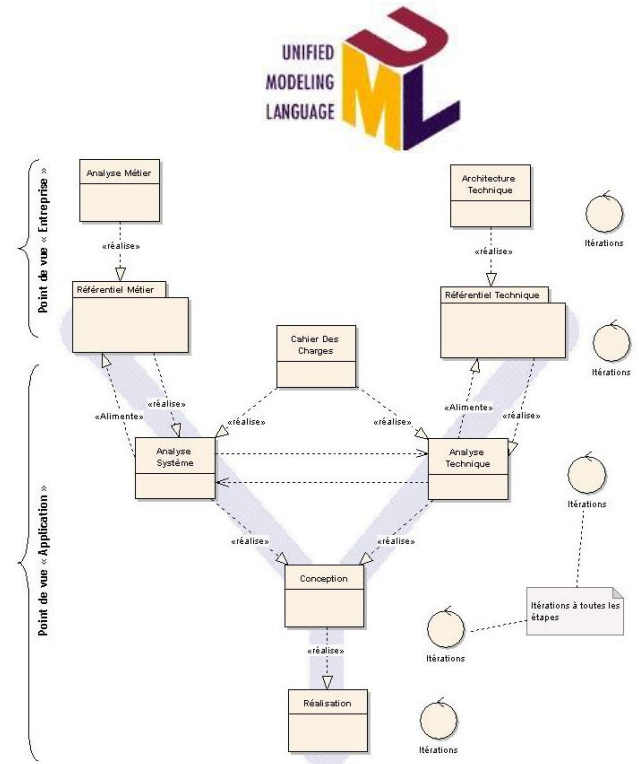
- Le développeur commence par identifier les concepts qui peuplent le domaine observé
  - On en déduit les classes métiers qui traitent des concepts connus des utilisateurs
  - On décrit ces classes en termes de
    - Données
    - Traitements
- Ces concepts métiers sont plongés dans un environnement technique
  - On en déduit des classes d'intégration spécialisées dans l'interaction
    - Avec les bases de données
    - Avec des fichiers
    - Avec des web services
    - Avec des annuaires
    - Avec les systèmes centralisés
    - ...
  - On en déduit des classes d'interface utilisateurs spécialisées dans la présentation
    - HTML
    - PDF
    - ...

# Les 3 concepts fondateurs

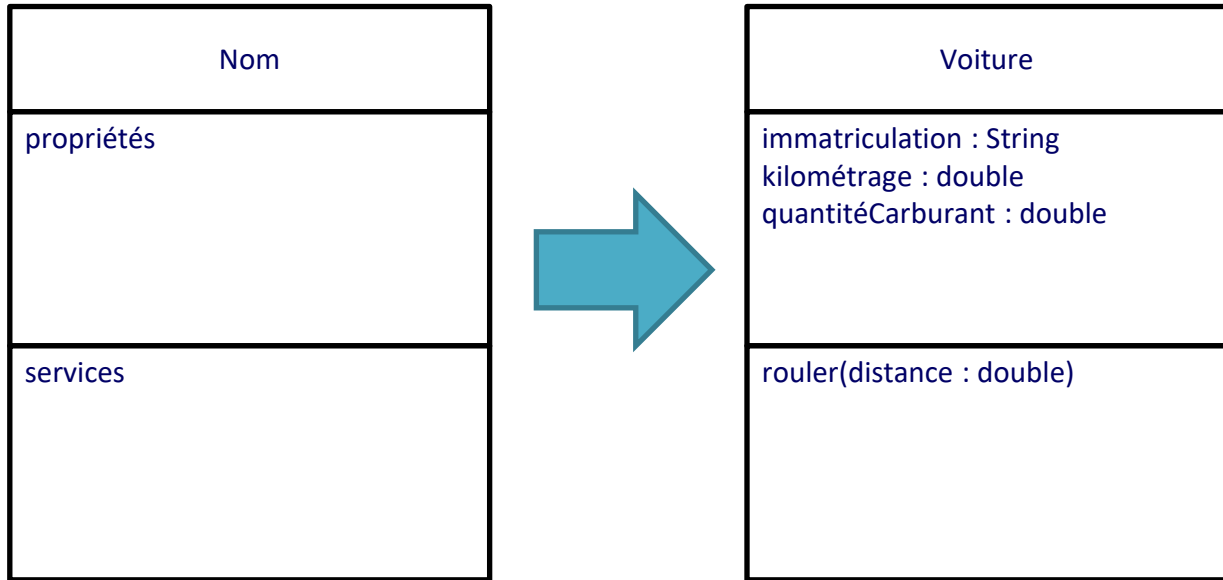
- Encapsulation
  - L'objet forme un tout
  - L'objet appartient à une nature (type) qui ne peut changer
  - L'objet est garant de son état
- Héritage
  - L'objet peut être une évolution d'un autre, plus général
- Polymorphisme
  - Des objets de natures différentes peuvent réagir au même message

# UML

- Unified Modelling Language
- Langage graphique ≠ méthode
- Boîte à outils
  - Vues
  - Diagrammes
  - Modèles d'éléments
- Reprend, étend plusieurs autres travaux de modélisation : Booch, Rumbaugh et Ivar Jacobson
- Association avec une méthode
  - OMT
  - RUP
  - 2TUP



# UML : l'élément classe





A black and white photograph of a person in a suit, holding a large, stylized cloud-shaped object. The cloud is white with a thick white outline and is positioned in the center of the frame. A blue horizontal band is overlaid across the middle of the image, containing the title text.

# Objets et classes

# Notion d'objet

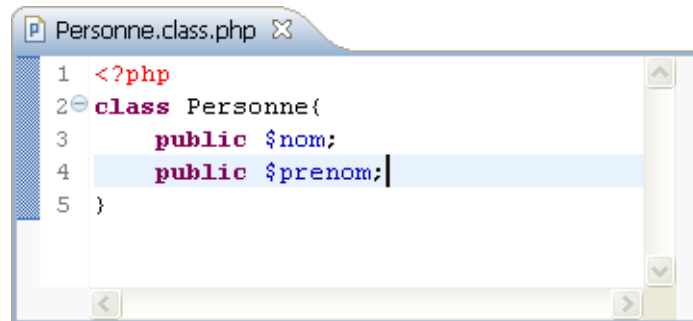
- Approche descriptive
  - Énumération des propriétés/attributs pertinents pour l'étude  
ex : une voiture a comme caractéristiques un kilométrage, une quantité de carburant et une immatriculation
  - ⇒ L'ensemble des propriétés de l'objet à un instant donnée constitue son état
  - ⇒ Modifier la valeur d'un des attributs revient à modifier l'état de l'objet
  - Énumération des capacités de l'objet étudié : les services qu'il peut réaliser  
ex : une voiture peut rouler une certaine distance
  - Traitements implémentés sous forme procédurale : les méthodes

# Description d'un objet

- Attributs + méthodes = Objet
  - Un nom permet de résumer les attributs et les méthodes : la classe  
ex : Voiture
  - Émergence d'un concept
    - Réflexion sur les propriétés communes à toutes les voitures  
⇒ Différence entre le concept et l'exemplaire
  - Principe d'abstraction :
    - Un nom résume un tout complexe
    - Permet d'appréhender la réalité sans s'attacher au superflu
    - Cela explique que l'approche est naturelle  
ex : un voyageur peut prendre l'avion sans tout connaître de l'avion

# La classe PHP

- Mot-clé `class`
- Marque la nature des objets issus de cette classe
- Un objet ne peut changer de nature au cours de sa vie
- Il est fréquent de rencontrer des fichiers nommés d'après la classe qu'ils contiennent
- Tout ce qui est dans l'accolade participe à la description du concept
- Il n'y a pas d'ordre dans la description



```
1 <?php
2 class Personne{
3     public $nom;
4     public $prenom;
5 }
```

# Attributs et méthodes

- Membres d'instance
- Pas d'ordre dans les déclarations
- Attributs
  - Structure de donnée
  - Type simple ou complexe
- Méthodes
  - Équivalent d'une fonction ou d'une procédure dans un contexte objet

# Attributs et méthodes

- Attributs
  - Déclaration directement dans la classe
  - Variable dont la durée de vie est l'instance
  - Global à l'instance
- Méthode
  - Fonction ou procédure dans un contexte objet
- `$this`
  - Les variables non-désignées par `$this` sont des variables locales
  - Les méthodes non désignées par `$this` sont des fonctions (hors classe)
  - Mot-clé qui représente l'instance courante
  - Suivi de « `->` », il permet d'accéder aux membres de l'objet

| Personne                        |
|---------------------------------|
| nom : String<br>prénom : String |
| parler(phrase : String)         |

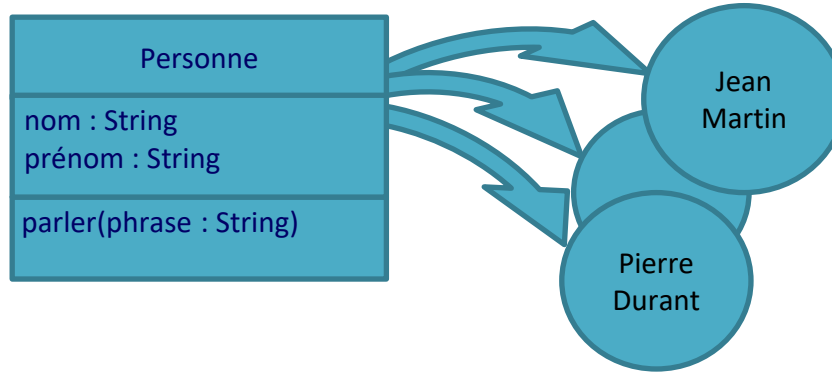
```
Personne.class.php
1 <?php
2 class Personne{
3     public $nom;
4     public $prenom;
5
6     public function parler($phrase){
7         printf('%s %s dit :"%s"', $this->nom,$this->prenom, $phrase );
8     }
9 }
```

# Classe et instance

- La classe
  - Concept
  - Vision statique
  - Ne "fait" rien
  - ⇒ Nécessité de rendre actif tout ce qui est décrit
  - ⇒ Utiliser le point d'entrée du programme qui réalise des traitements
- L'instance
  - Instance = objet = exemplaire  
ex : la voiture immatriculée "123 ABC 45"
  - Vision dynamique
  - Indépendance des "vies" des instances

# Constructeur

- Mécanisme qui utilise la classe comme une fabrique à objets



- Procédure d'initialisation
  - Ne renvoie rien
  - Construit et initialise des objets (instances)
  - Pont entre le concept et la réalité
  - Opérateur new
  - On peut définir un constructeur explicitement comme une méthode ayant pour nom `__construct()` ou le même identifiant que la classe



# Constructeur

```
script.php
1 <?php
2 require_once 'Personne.class.php';
3 $p = new Personne();
4 $p->parler('Bonjour');
```

Appel sur la classe :  
c'est à la classe qu'on demande un nouvel  
exemplaire

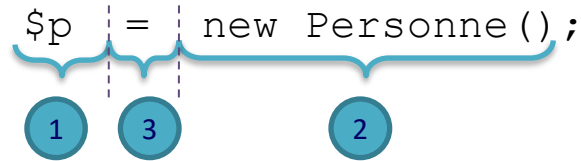
```
Personne.class.php
1 <?php
2 class Personne{
3     public $nom;
4     public $prenom;
5
6     function construct() {
7         $this->prenom = "Pierre";
8         $this->nom = "Dupont";
9     }
10
11     public function parler($phrase) {
12         printf('%s %s dit :"%s"', $this->nom,$this->prenom, $phrase );
13     }
14 }
```

Initialisation sur l'instance

# Instanciation

- Examen de l'instruction de construction

`$p = new Personne();`



1

Création d'une "variable de manipulation d'une Personne"

2

Pas de création de Personne. Valorisée à `null`

3

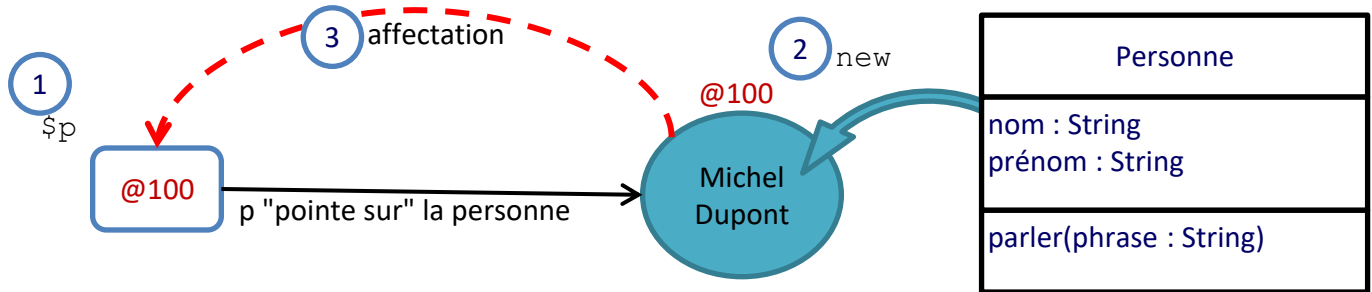
Création d'une nouvelle instance de Personne. Cela se traduit par une réservation de mémoire à une adresse (par ex : @100)



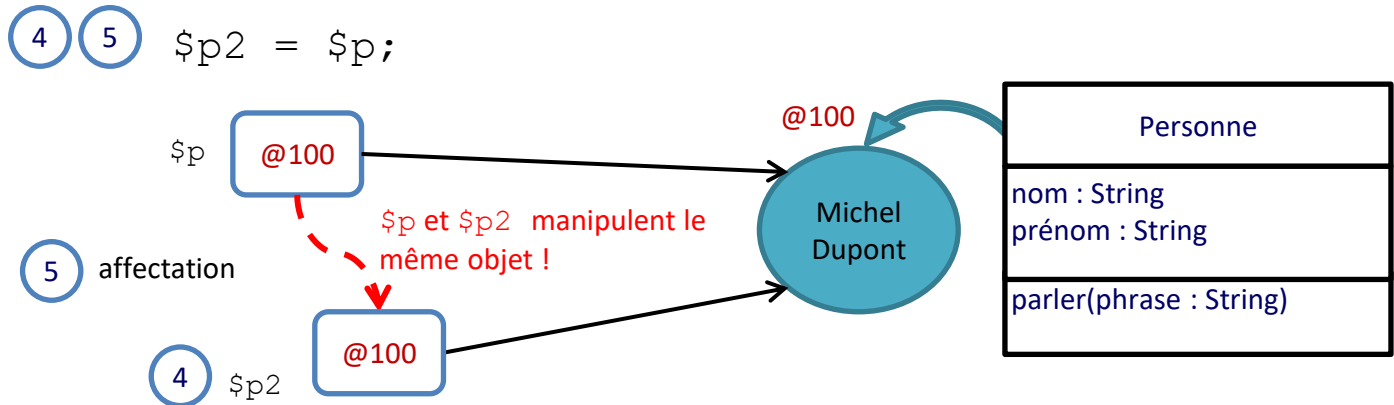
Affectation de la Personne nouvellement créée à la variable `p` par copie de l'adresse

"`$p`" n'est pas l'objet, mais un moyen de le manipuler

# Références aux objets



Si on affecte une référence à une autre :



# Constructeur

- Le constructeur par défaut
  - Défini tant qu'un autre constructeur n'est pas exprimé
  - ⇒ Écrire un autre constructeur "efface" le constructeur par défaut
  - C'est un constructeur sans paramètre
  - Il ne fait rien de plus que réserver de la mémoire

```
public function __construct(){}  

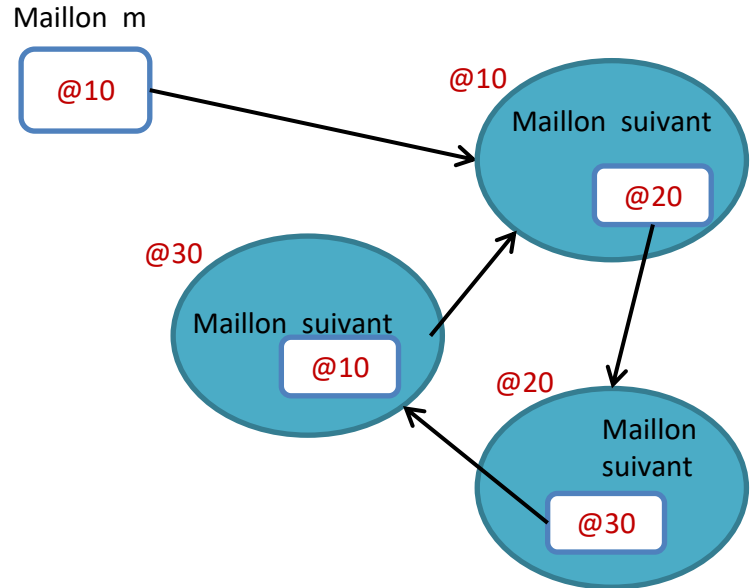
```

# Destructeur

- Construction donc destruction
- Préoccupation d'usage mémoire
- Processus de fond : le garbage collector (ramasse-miettes)
- En PHP: le destructeur est une méthode `__destruct()` appelée automatiquement par le garbage collector
- Système de comptage des références : si le nombre est à 0, l'objet est orphelin donc suppression
- Permet (presque) de s'affranchir de la réflexion : qui est responsable de la destruction de l'objet

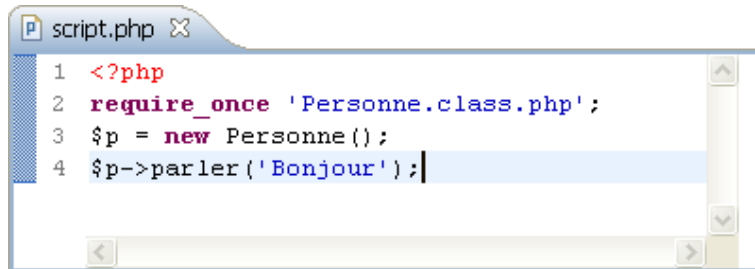
# Destructeur et gargabe collector

- Références circulaires
    - Si "m" disparaît :
      - La chaine de maillons est orpheline (grappe d'objets)
      - Pris séparément, chaque maillon n'est pas orphelin
  - ⇒ Le garbage collector ne nettoie pas la mémoire des objets d'adresses 10, 20 et 30
  - ⇒ Il faut "casser" au moins un lien : valeur `null` dans un des "suivant"
- De toutes façons, en fin de script, toutes les variables allouées sont effacées

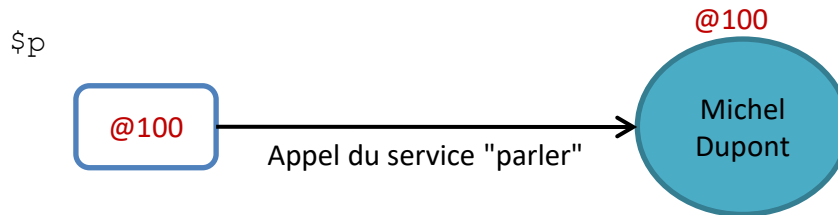


# Accès aux membres

- L'opérateur "->"
  - S'applique à la référence sur l'objet
  - N'est possible que si la référence n'est pas nulle
- Envoi d'un message à un objet destinataire
- En réponse l'objet déclenche un comportement



```
script.php X
1 <?php
2 require_once 'Personne.class.php';
3 $p = new Personne();
4 $p->parler('Bonjour');
```



# Encapsulation : un tout cohérent

- Exemple, que penser de :

```
script.php
1 <?php
2 require_once 'Voiture.class.php';
3 $v = new Voiture(0,35);
4 $v->rouler(100);
5 echo ($v->km);
6 $v->km=-12;
7
```

```
Voiture.class.php
1 <?php
2 class Voiture {
3     public $km;
4     public $qteCarb;
5
6     public function __construct($km, $qteCarb) {
7         $this->km=$km;
8         $this->qteCarb=$qteCarb;
9     }
10
11    public function rouler($distance) {
12        $this->km+=$distance;
13        $this->qteCarb-=$distance*0.07;
14    }
15 }
```

- La voiture doit rouler correctement
  - Modification de l'état de la voiture sans contrôle par celle-ci
- ⇒ Incohérence ⇒ Cas de violation d'encapsulation



# Encapsulation : un tout cohérent

⇒ L'objet doit se protéger

- Modificateurs de visibilité
  - `private` : invisible à l'extérieur de la classe, même en lecture
  - `public` : manipulable de l'extérieur de la classe, même en écriture
- Démarche : sauf bonne raison, on cache tout
- Si on veut autoriser la lecture, il faut passer par un service dédié : le getter (accesseur en lecture)
- Idem pour l'écriture : setter (accesseur en écriture). Objectif : faire des contrôles de validité des valeurs

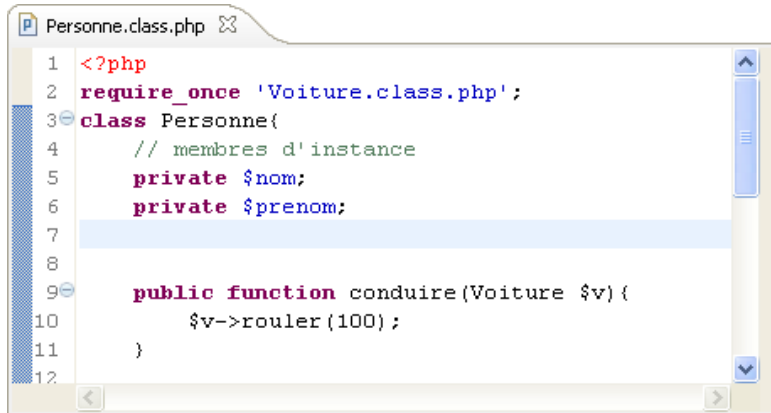
# Encapsulation : un tout cohérent

```
script.php
1 <?php
2 require_once 'Voiture.class.php';
3 $v = new Voiture(0,35);
4 $v->rouler(100);
5 echo($v->getKm());
6 // $v->km=-12; // illégal
7

Voiture.class.php
1 <?php
2 class Voiture {
3     private $km;
4     private $qteCarb;
5
6     public function __construct($km, $qteCarb) {
7         $this->km=$km;
8         $this->qteCarb=$qteCarb;
9     }
10
11     public function rouler($distance) {
12         $this->km+=$distance;
13         $this->qteCarb-=$distance*0.07;
14     }
15
16     public function getKm(){
17         return $this->km;
18     }
19
20     public function getQteCarb(){
21         return $this->qteCarb;
22     }
23
24     public function setQteCarb($qteCarb){
25         if ($qteCarb>0 && $qteCarb<40) {
26             $this->qteCarb=$qteCarb;
27         }
28         else{
29             // TODO : cf gestion des exception
30         }
31     }
32
33 }
```

# Les collaborations

- Association
  - Forme générale de collaboration
  - Forme de collaboration la plus faible
  - Connaissance via `require_once` si la classe est dans un autre script
  - Pas de référence permanente : pas de relation conteneur-contenu



```
Personne.class.php
1 <?php
2 require_once 'Voiture.class.php';
3 class Personne{
4     // membres d'instance
5     private $nom;
6     private $prenom;
7
8
9     public function conduire(Voiture $v){
10         $v->rouler(100);
11     }
12 }
```

# Les collaborations

- Relation conteneur-contenu
  - Construire des objets complexes à partir d'objets simples
  - Différentes sortes de collaborations  
Ex : quelle différence sémantique entre "une voiture contient des passagers"  
et "une voiture contient un moteur"

# Les collaborations

- Composition
  - Relation "forte"
  - Les cycles de vies des objets sont liés
    - La construction du conteneur implique la construction du contenu
    - La destruction du conteneur provoque la destruction du contenu
    - La destruction du contenu dégrade le conteneur
  - Exemple : Voiture - Moteur
  - Notion de "possession"

# Les collaborations

- Agrégation
  - Relation conteneur-contenu "faible"
  - Les cycles de vies des objets ne sont pas liés
    - La construction du conteneur est indépendante de celle du contenu
    - La destruction du conteneur ne provoque pas la destruction du contenu
    - Le contenu est facultatif dans le conteneur
  - Exemple : Voiture - Passager
- En PHP, à la différence du C++, mêmes déclarations
- Précautions relatives à la composition
  - Charge au programmeur de synchroniser les cycles de vie pour la composition, dans le constructeur
  - Le conteneur ne donne pas accès au contenu ⇒ violation d'encapsulation
  - Le problème est caduque pour l'agrégation car le conteneur ne possède pas le contenu

# Les collaborations

```
index.php
1 <?php
2 require_once 'Voiture.class.php';
3 require_once 'Personne.class.php';
4 $v = new Voiture(50, 'AA-123-BB', 30);
5 $p = new Personne("Dupont", "Pierre");
6 $p->conduire($v);
7
```

```
Moteur.class.php
1 <?php
2 class Moteur{
3     private $puissance;
4
5     public function consommer($distance){
6         return $this->puissance*$distance*0.0007
7     }
8
```

```
Voiture.class.php
1 <?php
2 require_once 'Moteur.class.php';
3
4 class Voiture {
5     private $km;
6     private $immat;
7     private $qteCarb;
8     /**
9      * lien de composition avec le moteur de la voiture
10     * @var Moteur
11     */
12     private $moteur;
13     const CAPACITE_RESERVOIR=40;
14
15     function __construct($km, $immat, $qteCarb, $puissance) {
16         $this->setImmat($immat);
17         $this->setKm($km);
18         $this->setQteCarb($qteCarb);
19         $this->moteur = new Moteur($puissance);
20     }
21
22     function __destruct() {
23         unset($this->moteur);
24     }
25
26     public function deplacer($distance) {
27         $this->km += $distance;
28         $this->qteCarb -= $this->moteur->consommer($distance);
29     }
30
```

A black and white photograph of a person in a suit, with their hands cupped together holding a large, stylized white cloud shape. The cloud has a blue horizontal band across its center.

# Membres d'instance et de classe



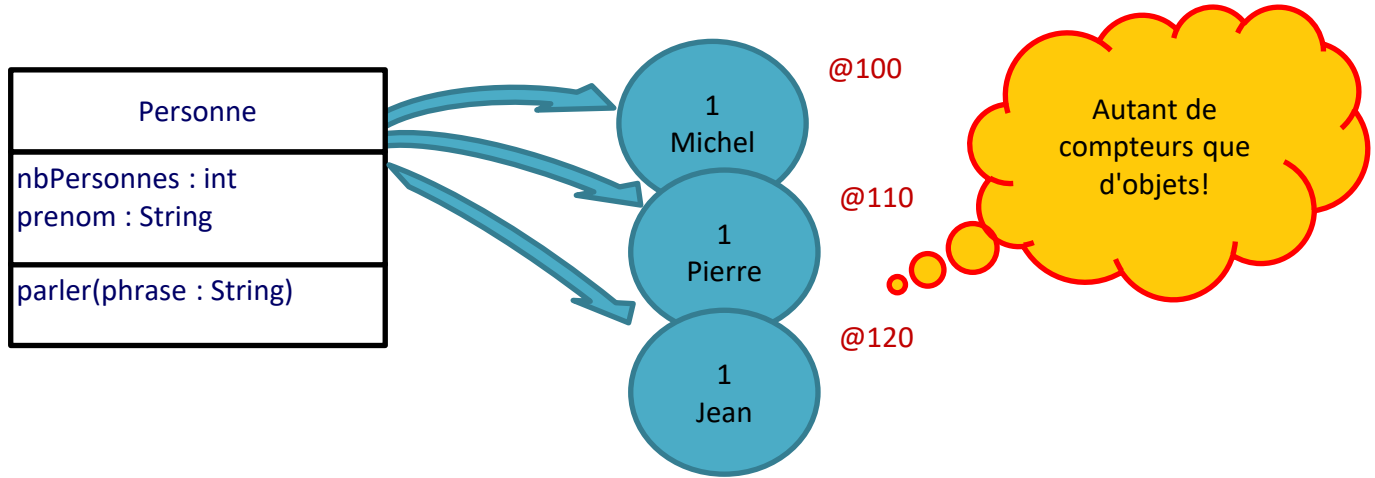
# Membres de classe

- Question : comment mettre en place un système de comptage des personnes?

```
public function __construct($prenom){  
    $this->prenom = prenom;  
    $this->nbPersonnes ++;  
}
```

- Quel problème rencontre-t-on?

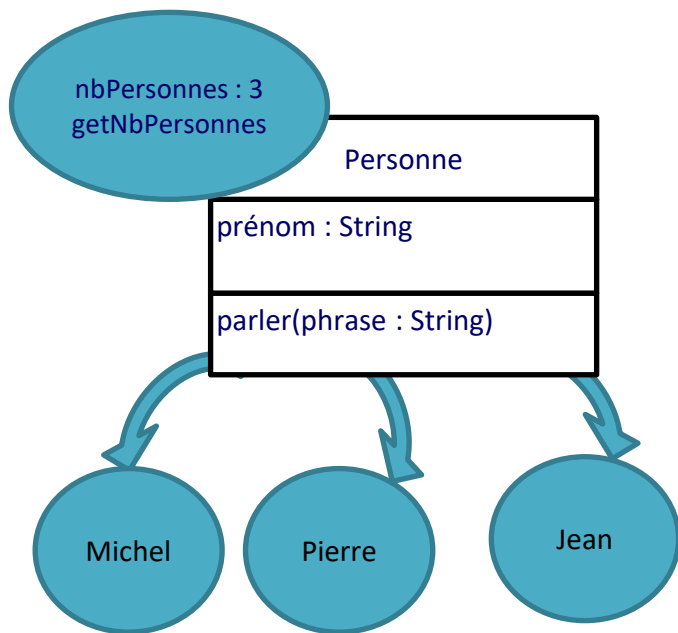
# Membres de classe



# Membres de classe

- ⇒ Parfois des données ne relèvent pas de l'instance mais de la classe
  - Membre de classe  $\neq$  membre d'instance
    - Mot-clé `static` = "de classe"
    - Propriétés de classe
    - Méthodes de classe
  - Les méthodes d'instance peuvent accéder aux membres de classe
  - Les méthodes de classe ne peuvent pas accéder aux membres d'instance

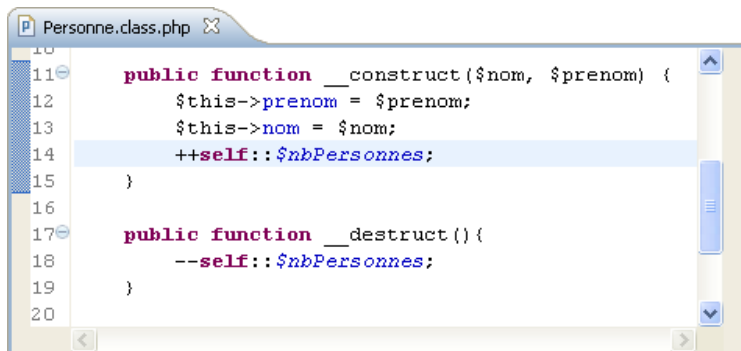
# Membres de classe



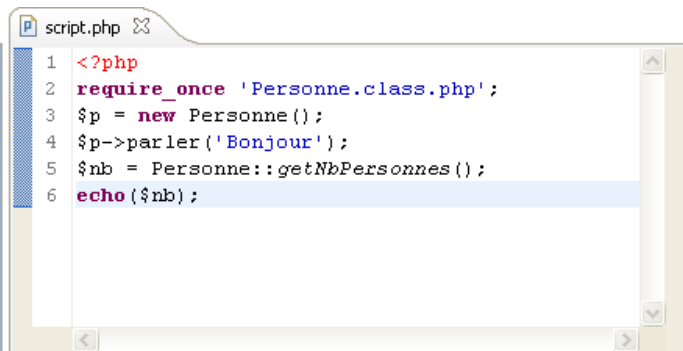
```
Personne.class.php
1  <?php
2  class Personne{
3      private static $nbPersonnes;
4      public static function getNbPersonnes(){
5          return self::$nbPersonnes;
6      }
7
8      private $nom;
9      private $prenom;
10
11     public function __construct($nom, $prenom) {
12         $this->prenom = $prenom;
13         $this->nom = $nom;
14         ++self::$nbPersonnes;
15     }
16
17     public function __destruct(){
18         --self::$nbPersonnes;
19     }
20
21     public function parler($phrase){
22         printf('%s %s dit :"%s"', |
23             $this->nom,$this->prenom, $phrase );
24     }
25 }
```

# Membres de classe

- Accès dans la classe
  - Mot clé `self::`
  - Ou `<classe>::`
- Accès hors de la classe
  - `<classe>::`



```
10
11 public function __construct($nom, $prenom) {
12     $this->prenom = $prenom;
13     $this->nom = $nom;
14     ++self::$nbPersonnes;
15 }
16
17 public function __destruct(){
18     --self::$nbPersonnes;
19 }
20
```



```
1 <?php
2 require_once 'Personne.class.php';
3 $p = new Personne();
4 $p->parler('Bonjour');
5 $nb = Personne::getNbPersonnes();
6 echo($nb);
```

# Membres de classe

- Stratégies d'implémentations
    - Membre de classe = Mutualisation de code
    - Mettre `static` tout ce qui peut l'être
      - Méthodes dont le résultat ne dépend pas de l'état de l'instance
    - Transformer les méthodes liées à l'instance en méthodes `static` quand c'est possible  
ex : calcul de la distance entre l'emplacement actuel de la voiture et la destination : calcul de la distance entre deux points
    - Constantes
      - Il ne faut pas déclarer les constantes comme `static` car le mot clé `const` place automatiquement les constantes au niveau de la classe et non plus dans l'instance
- ```
class Voiture{  
    const NB_ROUES=4;  
}
```
- echo(Voiture::NB\_ROUES);

A black and white photograph of a person in a suit, with their hands cupped together holding a large, stylized white cloud shape. A blue horizontal band with a white cloud-like border is overlaid across the middle of the image.

# Héritage & Polymorphisme

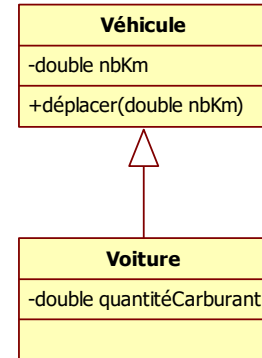
# Héritage

- Concept issu d'observations du réel
    - L'esprit humain classe naturellement les concepts : classification du règne animal, etc
    - Besoin de rattacher un concept précis à un autre plus général  
ex : une voiture est un véhicule, une moto est un véhicule
  - Avantages
    - Gain de temps en ne rappelant pas des qualités déjà admises  
ex : une moto est un véhicule motorisé à 2 roues
    - Abstraction par l'assimilation de plusieurs concepts à un seul  
ex : tous les véhicules peuvent se déplacer
- ⇒ Permet de se focaliser sur l'essentiel, tout en conservant la nature spécifique des objets



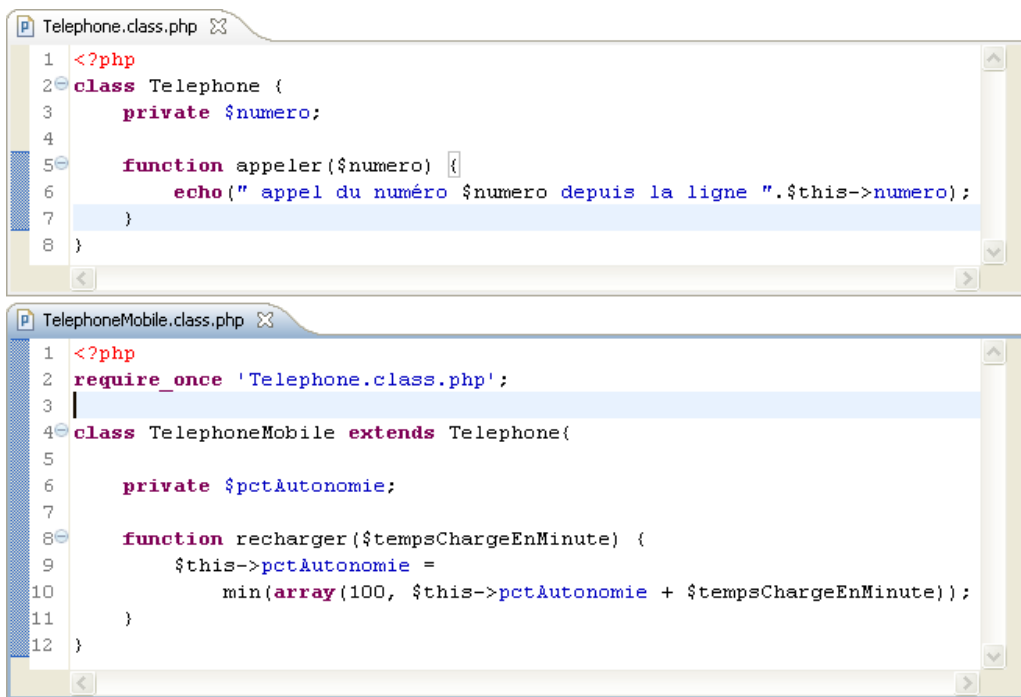
# Héritage

- Expression UML :
  - Flèche : sens de généralisation
  - Voiture spécialise Véhicule
  - Véhicule est une généralisation de Voiture
- Héritage au sens patrimonial
  - Ex : une voiture est un véhicule
  - Une instance de Voiture contient tout d'une instance de Véhicule
  - Les membres statiques ne sont pas hérités (ne sont pas dans l'instance)



# Héritage

- Syntaxe PHP : mot clé `extends`



```
Telephone.class.php
1 <?php
2 class Telephone {
3     private $numero;
4
5     function appeler($numero) {
6         echo(" appel du numéro $numero depuis la ligne ".$this->numero);
7     }
8 }

TelephoneMobile.class.php
1 <?php
2 require_once 'Telephone.class.php';
3
4 class TelephoneMobile extends Telephone{
5
6     private $pctAutonomie;
7
8     function recharger($tempsChargeEnMinute) {
9         $this->pctAutonomie =
10             min(array(100, $this->pctAutonomie + $tempsChargeEnMinute));
11     }
12 }
```

# Héritage

- Permet d'avoir une vision générale d'objets spécifiques
  - Ex : les véhicules se déplacent (peu importe leur nature spécifique)
  - Simplification : on ne s'intéresse qu'au minimum nécessaire (abstraction)

```
$tab[] = array();  
$tab[] = new Voiture();  
$tab[] = new Moto();  
$tab[] = new Vélo();  
for($tab as $v){  
    $v->déplacer(10);  
}
```

# Héritage : transtypage

- En PHP, les variables ne sont pas typées. Pas plus pour les objets que pour les types simples.
- Il n'y a donc généralement pas lieu de chercher à transtyper une variable dans un autre type.
- Il faut simplement se préoccuper du type de l'instance que l'on manipule et non de celui de la variable au travers de laquelle on y accède.
- Cependant il est possible de restreindre les types portés par un paramètre objet.
- Pour ce faire on fait précéder l'identifiant du paramètre par la classe avec laquelle il doit être compatible.
- Par compatible on entend : « Au moins aussi spécialisé ».

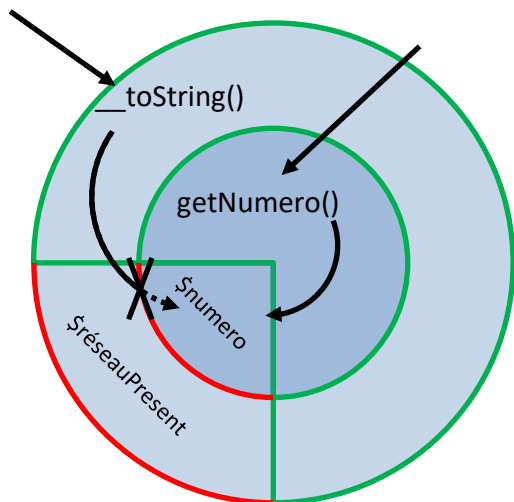
# Héritage : réutilisation de code

```
Telephone.class.php
1 <?php
2 class Telephone {
3     private $numero;
4
5     public function __construct($numero) {
6         $this->numero=$numero;
7     }
8
9     public function appeler($numero) {
10         echo(" appel du numéro $numero depuis la ligne ".$this->numero);
11     }
12
13     public function getNumero(){
14         return $this->numero;
15     }
16 }
```

```
TelephoneMobile.class.php
1 <?php
2 require_once 'Telephone.class.php';
3
4 class TelephoneMobile extends Telephone{
5
6     private $pctAutonomie;
7     private $reseauPresent;
8     private $operateurMobile;
9
10    public function __construct($numero, $operateurMobile) {
11        parent::__construct($numero);
12        $this->operateurMobile=$operateurMobile;
13    }
14
15    public function appeler($numero) {
16        if ($this->reseauPresent) {
17            parent::appeler($numero);
18        }
19    }
20
21    public function recharger($tempsChargeEnMinute) {
22        $this->pctAutonomie =
23            min(array(100, $this->pctAutonomie + $tempsChargeEnMinute));
24    }
25 }
26 $mob = new TelephoneMobile(654321987, "Pompabulé");
27 echo $mob->getNumero();
```

- Constructeur
  - Appel automatique au super-constructeur par défaut
  - Sinon, appel explicite au super-constructeur voulu :  
`parent::__construct()`
  - Réutilisation du bloc d'initialisation, pas de deuxième instance

# Héritage : visibilité des membres



- Bien qu'elle en soit constituée, la classe `TelephoneMobile` ne peut pas voir `$numero` car déclaré privé dans `Telephone`.
- Il existe un niveau intermédiaire, `protected`, qui permet de n'autoriser l'accès qu'aux sous-classes.

```
TelephoneMobile.class.php
1 $mob = new TelephoneMobile(612345789, "pompablé");
2 echo $mob;
3
4 <?php
5 require_once 'Telephone.class.php';
6
7 class TelephoneMobile extends Telephone{
8
9     private $pctAutonomie;
10    private $reseauPresent;
11    private $operateurMobile;
12
13    public function __construct($numero, $operateurMobile){
14        parent::__construct($numero);
15        $this->operateurMobile=$operateurMobile;
16    }
17
18    public function __toString(){
19        return "mon numero:".$this->numero;
20    }
21
```

```
Console
<terminated> TelephoneMobile.class [PHP CLI Application] C:\wamp\bin\php\php5.3.5\php.exe
PHP Notice: Undefined property: TelephoneMobile::$numero in C:\dev\poo\Heritage1\TelephoneMobile.class.php:0
PHP Stack trace:
PHP 1. {main}() C:\dev\poo\Heritage1\TelephoneMobile.class.php:0
PHP 2. TelephoneMobile->__toString() C:\dev\poo\Heritage1\TelephoneMobile.class.php:18
```

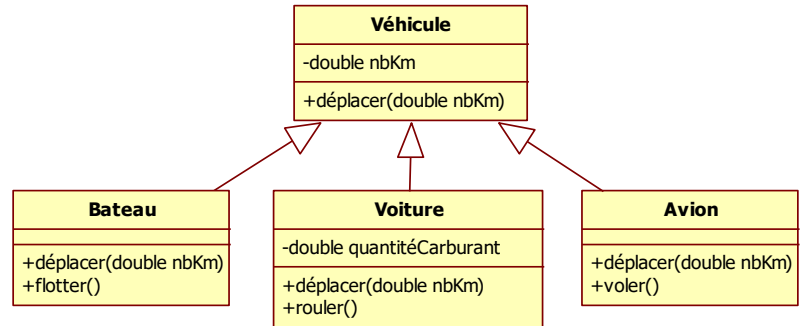
# Héritage : polymorphisme

- Partager des méthodes communes n'est pas suffisant  
ex : une voiture ne se déplace pas comme un bateau
  - ⇒ Classes spécifiques = comportement spécifique
  - ⇒ Mais garder une forme d'appel homogène  
ex : tous les Véhicules savent se déplacer sur une distance, mais pas de la même façon.
- Polymorphisme : un message homogène, pour des comportements différenciés.

# Héritage : polymorphisme

- On veut manipuler un tableau

```
for ($tab as $v){  
    v->déplacer(10);  
}
```



- Mais que déplacer soit spécifique selon que l'instance est de nature Bateau ou autre

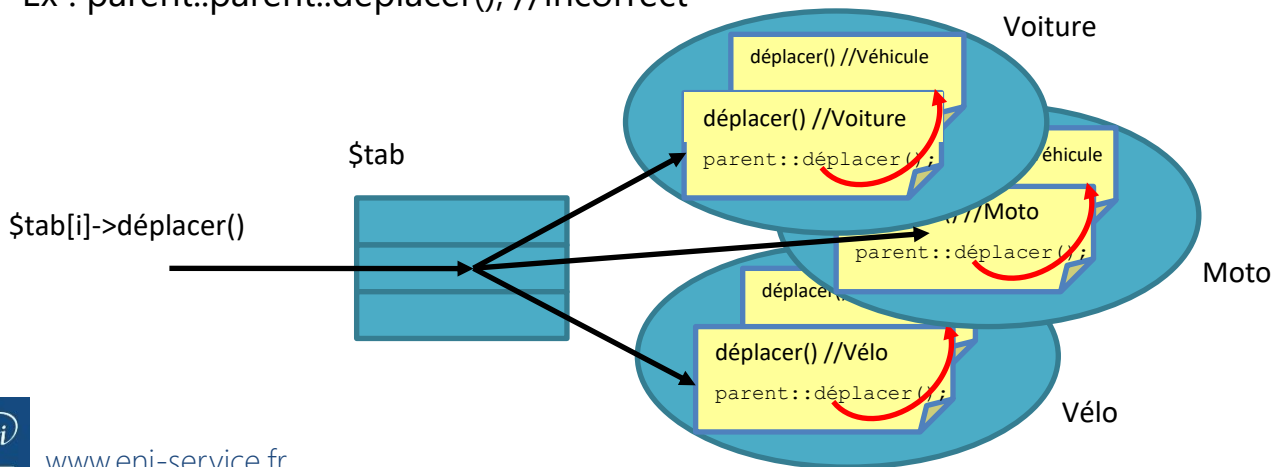
```
Vehicle.class.php
1 <?php
2 class Vehicule {
3     protected $km;
4
5     public function __construct($km) {
6         $this->setKm($km);
7     }
8
9     public function deplacer($distance) {
10        $this->km += $distance;
11    }
12
13    public function getKm() {
14        return $this->km;
15    }
16 }
```

```
Voiture.class.php
1 <?php
2 require_once 'Vehicule.class.php';
3
4 class Voiture extends Vehicule {
5     protected $qteCarb;
6
7     function __construct($km, $qteCarb) {
8         parent::__construct($km);
9         $this->qteCarb=$qteCarb;
10    }
11
12    public function deplacer($distance) {
13        $this->qteCarb -= $distance*0.07;
14    }
15 }
```



# Héritage : polymorphisme

- En PHP, réécriture d'une méthode qui possède exactement la même signature que la méthode héritée.
- Les 2 méthodes restent présentes dans l'instance.
- La nouvelle méthode masque l'ancienne
- Lors de l'appel sur une instance, c'est la dernière définie qui est immédiatement disponible
- La méthode redéfinie peut appeler la méthode héritée via "parent::" ⇒ Réutilisation de code  
ex : `parent::déplacer();`
- Mais elle ne peut pas "sauter une génération".  
Ex : `parent::parent::déplacer();` //incorrect



# Polymorphisme : exercice

- Quel est l'ordre d'exécution des méthodes déplacer et consommer à l'appel de la ligne 8 de script.php ?
- Et pourquoi ?

```
script.php
1 <?php
2 require_once 'Voiture.class.php';
3
4 $flotteVehicules[] = new Voiture(70, 35);
5 $flotteVehicules[] = new Voiture(10, 20);
6
7 foreach ($flotteVehicules as $vehicule) {
8     $vehicule->deplacer(50);
9 }
```

```
Vehicule.class.php
1 <?php
2 class Vehicule {
3     protected $km;
4
5     A public function deplacer($distance) {
6         $this->km += $distance;
7         $this->consommer($distance);
8     }
9
10    B public function consommer($distance){
11        echo 'Le véhicule consomme';
12    }
```

```
Voiture.class.php
2 require_once 'Vehicule.class.php';
3
4 class Voiture extends Vehicule {
5     protected $qteCarb;
6
7     C public function deplacer($distance) {
8         parent::deplacer($distance);
9         $this->userPneus($distance);
10    }
11
12    D public function consommer($distance){
13        $this->qteCarb -= $distance*0.07;
14        parent::consommer($distance);
15    }
```

# Héritage : les classes abstraites

- Pas d'exemplaire pour le concept
  - Empêcher l'appel au constructeur par new
  - Forcer l'héritage
  - But : factoriser et réutiliser du code
  - Améliorer la conformité à la réalité  
ex : il n'existe pas de véhicule qui ne soit que véhicule mais, ni une voiture, ni un exemplaire d'aucune des sous-classes
- ⇒ Syntaxe : **abstract** class
- Le bloc constructeur reste pertinent
  - Appel via parent : `__construct()` dans les classes filles
  - Sert de bloc d'initialisation factorisé

# Héritage : les méthodes abstraites

- Cas de définition incomplète
  - Par ex : un véhicule consomme de l'énergie pour son déplacement
  - Mais les modalités de consommations sont inconnues à ce stade
  - ⇒ Ça dépend du type de véhicule, ie. de la sous-classe réelle
  - ⇒ **abstract** protected function consommer(\$distance);
- Par conséquent la classe est abstraite et doit être marquée comme telle.
  - Pas d'implémentation de consommer dans Véhicule
  - Impossible de créer une instance incomplète
- Cela induit un contrat entre la classe abstraite et ses sous-classes :
  - Soit les classes dérivées donnent une implémentation
  - Soit elles sont elles-mêmes abstraites

# Héritage : les classes finales

- Logique de propriété/économique/contractuelle :
  - Empêcher de réutiliser nos classes par héritage
  - Garantit que la classe finale ne sera jamais classe mère
  - ⇒ Syntaxe : **final** class
  - ⇒ Ne peut être utilisé conjointement avec abstract (objectifs contraires)
- ⇒ N'empêche pas la réutilisation par composition
- ⇒ Oblige à revenir sur le code pour étendre les capacités par héritage
- Méthodes finales : granularité plus fine
  - Interdire la redéfinition d'une méthode dans les classes filles
  - Syntaxe (exemple) : **final** public function f(){...}

A black and white photograph of a person in a suit, with their hands cupped together holding a large, stylized white cloud graphic. A horizontal blue band with a white cloud outline runs across the middle of the image, containing the title text.

# Concepts avancés

# Les interfaces

- Les interfaces permettent de définir un contrat qui spécifie quelles méthodes une classe doit implémenter, sans avoir à définir comment ces méthodes fonctionneront
  - Elles sont utilisées pour représenter des capacités transverses de classes
- Comme pour une classe abstraite, une interface n'est pas instanciable
- Une interface ne peut contenir que les signatures des méthodes
  - Eventuellement des constantes
  - Mais pas d'attributs !
- Une classe qui choisit de respecter le contrat de l'interface, « réalise » ce contrat
  - On dit également « implémente » ce contrat
- Les objets de cette classe auront un nouveau type supplémentaire, celui de l'interface

# Les interfaces : Règles

- Une interface est constituée de méthodes qui n'ont pas d'implémentations
  - Ce qui signifie que les méthodes d'interface sont des méthodes abstraites
    - Mais le mot clé `abstract` ne doit pas être utilisé ...
- Toutes les méthodes dans les interfaces doivent avoir une portée de visibilité `public`
- Les interfaces sont différentes des classes car une classe peut hériter d'une seule classe alors que la classe peut implémenter une ou plusieurs interfaces
- La déclaration d'une interface se fait avec le mot clé `interface` à la place du mot `class`
- Une classe déclare qu'elle implémente une interface avec le mot clé `implements`



# Les interfaces : Déclaration et usage

- La classe Abonne doit redéfinir les 3 méthodes de l'interface
  - Sinon elle sera considérée comme abstraite !

```
interface Utilisateur {  
    public const ABONNEMENT = 15;  
    public function getNom();  
    public function setPrixAbo();  
    public function getPrixAbo();  
}
```

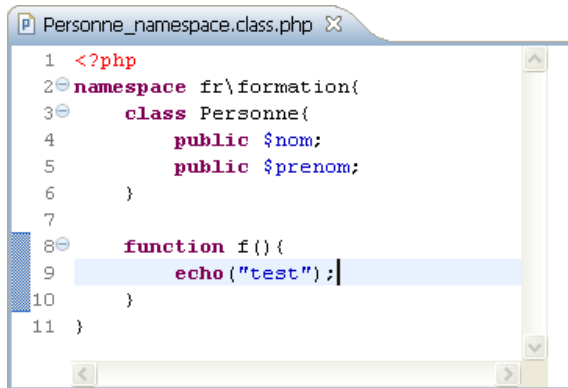
Méthodes venant de  
l'interface.  
A redéfinir dans la classe.



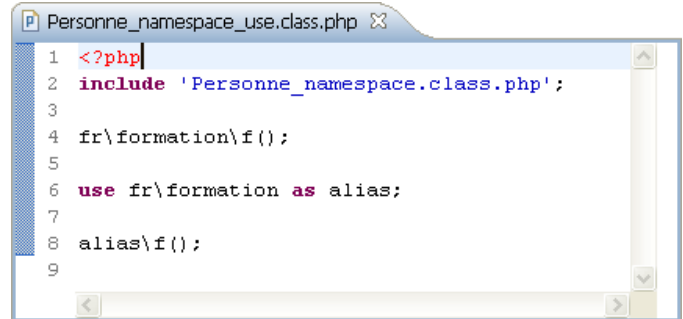
```
class Abonne implements Utilisateur{  
    protected $user_name;  
    protected $user_region;  
    protected $prix_abo;  
    protected $user_pass;  
  
    public function __construct($n, $p, $r) {  
        $this->user_name = $n;  
        $this->user_pass = $p;  
        $this->user_region = $r;  
    }  
  
    public function getNom() {  
        // Implémentation à fournir...  
    }  
  
    public function getPrixAbo() {  
        // Implémentation à fournir...  
    }  
  
    public function setPrixAbo() {  
        // Implémentation à fournir...  
    }  
}
```

# Classes et espaces de nom

- Multiplicité des classes - fichiers
- Nécessité de rangement
  - Niveau physique : répertoires
  - Niveau logique : « namespace » → espace de nom
- Une classe est déclarée dans un espace de nom : mot-clé namespace
- Au minimum elle se trouve dans l'espace de nom global
- Une classe connaît celles qui sont dans le même namespace
- Dans un espace de nom il faut nommer explicitement les espaces de nom des autres fonctionnalités qui s'y trouvent : mot-clé use



```
1 <?php
2 namespace fr\formation{
3     class Personne{
4         public $nom;
5         public $prenom;
6     }
7
8     function f(){
9         echo("test");
10    }
11 }
```



```
1 <?php
2 include 'Personne_namespace.class.php';
3
4 fr\formation\f();
5
6 use fr\formation as alias;
7
8 alias\f();
9
```

A black and white photograph of a person in a suit, with their hands cupped together holding a large, stylized white cloud graphic. A horizontal blue band with a white cloud outline runs across the middle of the image, containing the title text.

# Les exceptions

# Les exceptions

- Besoin de gérer les erreurs

```
double diviser(double numerateur, double denominateur){  
    return numerateur/denominateur; //pb possible  
}  
q = diviser(n,d);
```

- Historiquement : code retour d'une fonction

- Détournement du fondement mathématique de la syntaxe (exemple en C)

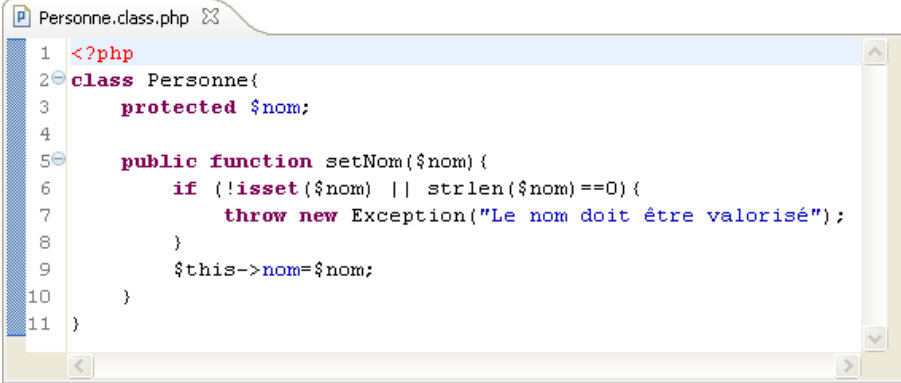
```
int diviser(double numerateur, double denominateur, double *resultat){  
    if(denominateur != 0){  
        *resultat = numerateur/denominateur; //pb écarté  
        return CODE_OK;  
    }  
    else{  
        return CODE_DIV_0;  
    }  
}  
code = diviser(n,d,&q);  
if (code==CODE_OK) {...}
```

- Un mal nécessaire

# Les exceptions

- Besoin d'un autre canal de sortie : les erreurs
  - Séparation du flux de données et d'erreur
  - La syntaxe d'appel redevient naturelle
  - Syntaxe appropriée en cas d'erreur
- En PHP les erreurs sont des objets :
  - Issus de la classe `Exception`
  - Il faut les créer comme pour tous les objets
- 2 parties
  - La détection du cas d'erreur = émission de l'erreur
  - Le traitement de l'erreur

# Les exceptions : origine de l'erreur



```
1 <?php
2 class Personne{
3     protected $nom;
4
5     public function setNom($nom) {
6         if (!isset($nom) || strlen($nom)==0) {
7             throw new Exception("Le nom doit être valorisé");
8         }
9         $this->nom=$nom;
10    }
11 }
```

- Test sur les pré-conditions
- throw : envoi d'un objet Exception au premier bloc de traitement possible

# Les exceptions : traitement de l'erreur

script.php

```
1 <?php
2 require_once 'Personne.class.php';
3 $p = new Personne();
4
5 // ligne potentiellement génératrice d'exception
6 $p->setNom($_GET["nom"]);
```

Ici c'est le gestionnaire d'erreurs par défaut qui va décider quoi faire de l'exception

Ici nous décidons par du code spécifique quoi faire de l'exception

script.php

```
1 <?php
2 require_once 'Personne.class.php';
3 $p = new Personne();
4 try{
5     // ligne potentiellement génératrice d'exception
6     $p->setNom($_GET["nom"]);
7 }
8 catch(Exception $e){
9     echo $e->getMessage();
10 }
```

- Gérer les exceptions est presque obligatoire
- Responsabilités du traitement :
  - Il doit se terminer explicitement bien
    - Aucune exception n'est survenue
    - Ou interception d'une erreur et traitement curatif dans le catch :  
try{...} catch(Exception e){...}
  - ou il doit se terminer explicitement mal
    - les couches basses le signalent via un throw
    - Au niveau de l'interface utilisateur on affiche un message

# Exercice



[www.eni-service.fr](http://www.eni-service.fr)



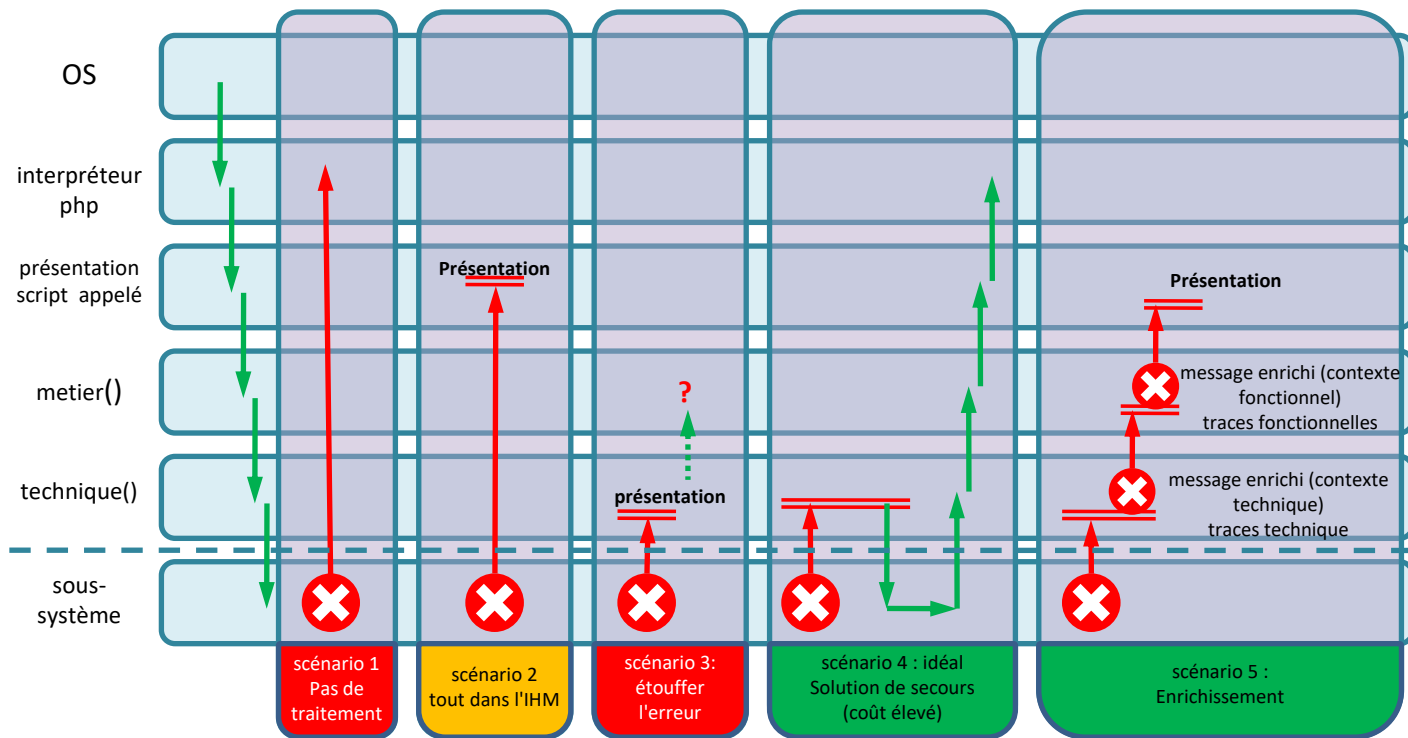
# Les exceptions : exercice

Dans quel ordre les instructions de f() sont-elles exécutées ?

```
function f(){
    i1();
    try{
        i2();
        try{
            g();
            i3();
        }
        catch(Exception $e){
            i4();
            g();
            i5();
        }
        i6();
    }
    catch(Exception $e){
        i7();
    }
    i8();
}
```

```
function g(){
    i9();
    try{
        pb(); // provoque une PbException
        i10();
    }
    catch(PbException $e){
        i11();
        throw new Exception("problème");
        i12();
    }
    i13();
}
```

# Les exceptions : différents scénarios



# Les exceptions : chaînage

- Constructeur

- `Exception($message=null, $code=null, $previous=null);`

- Usage

```
try{
    unTraitementTechnique();
}
catch(ExceptionTechnique $e){
    throw new ExceptionFonctionnelle("message fonctionnel", null, $e);
}
```

- Méthode `getPrevious()`

- Permet de remonter à l'Exception technique en interceptant l'exception fonctionnelle

```
try{
    unTraitementFonctionnel();
}
catch(ExceptionFonctionnelle $e){
    echo("message technique:". $e->getPrevious()->getMessage());
}
```

# Les exceptions utilisateur

- Avoir ses propres classes d'exception
  - Exceptions techniques
  - Exceptions métiers
  - Exceptions de présentation
- ⇒ Hériter de Exception
- Reprendre les constructeurs
  - Ajouter des propriétés et des méthodes

# Les exceptions : bonnes pratiques

- Qu'est-ce qu'un cas d'erreur?
  - Les pré-conditions ne sont pas respectées
  - Le sous-système de la fonctionnalité est défaillant
- Qu'est-ce qui n'est pas un cas d'erreur?
  - Un scénario applicatif qui doit refuser une fonctionnalité  
ex : interdiction de connexion suite à un mot de passe erroné
  - Le try-catch ne doit pas remplacer le if-else

# Les exceptions : le finally

- Parfois, il faut avoir la garantie d'exécuter du code même en cas d'exception
  - Exemple : libération de ressources

## ⇒ Le bloc `finally`

```
$maRessource = new UneRessource();  
try{  
    maRessource->traiter();  
}  
catch(ExceptionTechnique $e){  
    throw new ExceptionFonctionnelle("message fonctionnel", null, $e);  
}  
finally{  
    maRessource->close();  
}
```

- Le `catch` n'est pas obligatoire

# Travaux pratiques

