

A close-up, low-angle shot of an elephant's head and trunk, showing the intricate texture of its skin. The elephant is positioned on the right side of the frame, with its trunk extending downwards. The background is dark and out of focus.

LA PROGRAMMATION ORIENTÉE OBJET EN PHP

Jean-Claude AZIAHA

SOMMAIRE

PARTIE A: Introduction

- 1- Qu'est-ce que la POO ?
- 2- Quelle est son utilité ?
- 3- Quel environnement de travail pour coder en POO ?

SOMMAIRE

PARTIE B : Les bases de la poo

- 1- Qu'est-ce donc qu'un objet ?
- 2- Qu'est-ce qu'une classe ?
- 3- Comment instancier une classe ?
- 4- Que peut contenir une classe ?
- 5- L'encapsulation
- 6- Les getters et setters
- 7- \$this
- 8- Le constructeur

SOMMAIRE

PARTIE B: Les bases de la poo

9- Les méthodes et attributs statiques

10- L'héritage

11- Redéfinition d'une méthode ou d'un attribut dans une classe fille

12- Classes et méthodes abstraites

13- Classes et méthodes finales

14- Les interfaces

15- Les traits

INTRODUCTION

1- Qu'est-ce que la programmation orientée objet

- La programmation orientée objet n'est pas un langage de programmation.
- C'est un paradigme de programmation.
- C'est-à-dire que c'est une manière de programmer dont le point central est l'utilisation des objets.

2- Quelle est son utilité

- La programmation orientée objet permet de mieux organiser son code en regroupant les données et les fonctionnalités associées au sein d'objets.
- Grâce à cette approche, il est possible de structurer les informations en plusieurs blocs solides et sécurisés.
- Cette méthode est particulièrement utile pour le développement de gros projets, où la gestion d'un grand volume de code nécessite une architecture claire et bien organisée.

3- Quel environnement de travail pour coder en POO ?

- L'environnement de travail reste identique à celui utilisé pour l'écriture du code PHP en mode procédural.
- Tous les éléments nécessaires à la programmation orientée objet sont déjà présents: un serveur(Apache), PHP, MySQL, un éditeur de texte.

LES BASES DE LA POO

1- Qu'est-ce donc qu'un objet ?

- En programmation, un objet est une instance d'une classe.
- Il peut contenir à la fois des propriétés (attributs) et des méthodes.

1- Qu'est-ce donc qu'un objet ?

- Les propriétés peuvent être de type scalaire (int, string...) et/ou de type composé (tableau, objet).
- Les méthodes sont en réalité des fonctions qui représentent les actions que peut effectuer cet objet.

1- Qu'est-ce donc qu'un objet ?

- Nous avons vu qu'un objet est obtenu par l'instanciation d'une classe.
- Prenons maintenant le temps de comprendre ce qu'est une classe, avant d'apprendre à créer des objets à partir de celle-ci.

2- Qu'est-ce qu'une classe ?

- La classe, c'est comme une maquette sur laquelle l'on se base pour créer le nombre d'objets que l'on souhaite.
- Elle définit ce que l'objet pourra faire et comment il se comportera.
- En PHP, une classe se déclare avec le mot-clé **class**, suivi de son nom (par convention, celui-ci commence par une majuscule).

2- Qu'est-ce qu'une classe ?

```
<?php  
class Robot  
{  
  
}  
?>
```

3- Comment instancier une classe

- Instancier une classe revient à créer un objet.
- Pour cela, on utilise le mot-clé **new** suivi du nom de classe.
- Exemple : `new Robot;`
- Afin de manipuler cet objet plus facilement dans le code, on le stocke dans une variable. La syntaxe devient donc:
`$robot0 = new Robot;`

3- Comment instancier une classe

```
<?php  
class Robot  
{  
  
}  
  
$robot1 = new Robot;  
?>
```

```
?>
```

```
$robot1 = new Robot;
```


4- Que peut contenir une classe ?

Une classe contient :

- des **propriétés** qui sont des variables ou des constantes
- des **méthodes** qui sont des fonctions

Il faut savoir que pour déclarer un attribut ou une méthode dans une classe, il faut obligatoirement lui **préciser le mode de visibilité**.

La définition de ce mode de visibilité est communément appelée : **l'encapsulation**.

5- L'encapsulation

L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en empêchant l'accès aux données .

L' encapsulation, comme dit précédemment, c'est le mode de visibilité d'un attribut et/ou d'une méthode appartenant à une classe.

5- L'encapsulation

Il existe 3 modes:

- **public**: C'est le mode le plus ouvert. Les attributs et/ou les méthodes qui sont déclarés en mode public dans une classe sont accessibles partout. Que cela soit dans la classe elle-même ou en dehors.

5- L'encapsulation

protected: C'est le mode moyennement restrictif.

C'est-à-dire qu'en déclarant des attributs ou des méthodes en mode protected, ces derniers sont accessibles uniquement:

- Dans la classe elle-même
- Ses classes filles (C'est-à-dire les classes qui vont hériter de la classe parente).

5- L'encapsulation

- **private**: C'est le mode le plus restrictif. Comme le dit son nom, c'est privé! Cela veut donc dire que les attributs et/ou les méthodes qui sont déclarés en mode private dans une classe sont accessibles uniquement dans la classe qui les a définis.

5- L'encapsulation

```
<?php
```

```
class Vehicule
{
    public $marque = "Audi";
    protected $couleur = "bleue";
    private $nitro = true;
}
```

```
$vehicule0 = new Vehicule;
echo $vehicule0->marque . "</br/>"; // Audi
echo $vehicule0->couleur . "</br/>";
// Fatal error: Uncaught Error: Cannot access protected property
```

```
// Fatal error: Uncaught Error: Cannot access protected property
```

```
echo $vehicule0->couleur . "</br/>";
```

```
echo $vehicule0->couleur . "</br/>";
```

5- L'encapsulation

```
<?php
class Vehicule
{
    public $marque = "Audi";
    protected $couleur = "bleue";
    private $nitro = true;

    public function demarrer()
    {
        echo "Vroum!!! Elle démarre";
    }

    protected function rouler() // or private function
    {
        echo "Elle roule";
    }
}

$vehicule0 = new Vehicule;
echo $vehicule0->demarrer() . "</br/>"; // Vroum!!! Elle démarre
echo $vehicule0->rouler() . "</br/>";
// Fatal error: Uncaught Error: Call to protected method Vehicule::rouler()
```

```
// Fatal error: Uncaught Error: Call to protected method Vehicule::rouler()
```

5- L'encapsulation

Au final, faut-il déclarer les attributs d'une classe en public, protected ou private ?

❖ **Cela dépendra du besoin.**

Toutefois, de manière générale, il est plutôt conseillé de déclarer les **attributs** d'une classe en **privé** pour des **raisons de sécurité**.

Comment fait-on alors pour y **accéder** en dehors de la classe puisque les **attributs** sont **privés**?

6- Les getters et setters

- C'est là qu'entre en jeu les **getters** et les **setters**.
- Les **getters** sont donc des méthodes qui sont déclarées en public dans une classe.
- C'est grâce aux **getters** que nous allons récupérer les attributs déclarés en **protected** ou **private**
- Prenons un exemple pour mieux éclaircir tout ça.

5- L'encapsulation

index.php > ...

```
1  <?php
2      class Vehicule
3      {
4          public $marque = "Audi";
5          protected $couleur = "bleue";
6
7          public function getMarque()
8          {
9              return $this->marque;
10         }
11
12         public function getCouleur()
13         {
14             return $this->couleur;
15         }
16     }
17
18     $vehicule0 = new Vehicule;
19     echo $vehicule0->getMarque() . "\n"; // Audi
20     echo $vehicule0->getCouleur() . "\n"; //bleue
```

6- Les getters et setters

```
index.php > ...
1  <?php
2      class Vehicule
3      {
4          public $marque = "Audi";
5          protected $couleur = "bleue";
6
7          public function setMarque($marque)
8          {
9              $this->marque = $marque;
10         }
11
12         public function setCouleur($couleur)
13         {
14             $this->couleur = $couleur;
15         }
16
17         public function getMarque()
18         {
19             return $this->marque;
20         }
21
22         public function getCouleur()
23         {
24             return $this->couleur;
25         }
26     }
```

6- Les getters et setters

```
27
28 $vehicule0 = new Vehicule;
29 $vehicule0->setMarque("Tesla") . "\n";
30 $vehicule0->setCouleur("blanche") . "\n";
31
32 echo $vehicule0->getMarque() . "\n"; //Tesla
33 echo $vehicule0->getCouleur() . "\n"; //blanche
34
```

6- Les getters et setters

- Dans cet exemple, nous avons modifié la valeur des attributs \$marque et \$couleur de notre classe **Vehicule**.
- Bien que les attributs soient déclarés en privé, nous avons pu réaliser cette opération grâce aux setters **setMarque() {}** et **setCouleur() {}**.
- Afin d'afficher la nouvelle valeur que contient \$marque et \$couleur, nous avons utilisé les getters **getMarque() {}** et **getCouleur() {}**

7- \$this

Nous avons commencé depuis le chapitre sur les getters et les setters à utiliser **\$this**.

\$this lorsqu'il est utilisé dans une classe **désigne l'objet courant**.
C'est-à-dire l'objet qui sera généré à partir de cette classe.

\$this est utilisé pour désigner un attribut et/ou une méthode au sein d'une classe, en faisant référence à l'objet issu de cette classe.

7- \$this

index.php > ...

```
1  <?php
2  class Vehicule
3  {
4      public $marque = "Audi";
5      protected $couleur = "bleue";
6
7      public function getMarque()
8      {
9          return $this->marque;
10     }
11
12     public function getCouleur()
13     {
14         return $this->couleur;
15     }
16
17     public function getPhrase()
18     {
19         echo "La voiture est de marque: " . $this->marque . " et sa couleur est " . $this->couleur . ".\n";
20     }
21 }
22 $vehicule0 = new Vehicule();
23 $vehicule0->getPhrase() . "\n"; //La voiture est de marque: Audi et sa couleur est bleue.
```

```
53 $vehicule0->getPhrase() . "\n"; //La voiture est de marque: Audi et sa couleur est bleue.
```

8- Le constructeur

Jusqu'ici, dans notre classe Vehicule, les attributs ont toujours contenu des valeurs par défaut.

Pour être plus clair, à chaque fois qu'on crée un objet sur la base de la classe Vehicule, cet objet aura toujours par défaut une marque « Audi » et une couleur « bleue ».

8- Le constructeur

Si notre classe est censée être une maquette sur laquelle on peut se baser pour créer plusieurs véhicules, il serait plus intéressant d'avoir la possibilité de définir la marque, ainsi que la couleur nous-même lors de la création de cet objet.

C'est dans ce sens qu'intervient la fonction magique **__construct()**

Encore appelée, le constructeur d'une classe.

8- Le constructeur

Le constructeur est une fonction magique dans le sens où c'est elle qui est appelée en premier à chaque fois que l'on instancie une classe.

Cette fonction existe par défaut dans une classe.

Elle est donc toujours là, que l'on la déclare nous-même ou non.

La particularité des fonctions magiques est que leur nom est précédé de __

Cette fonction peut prendre des paramètres ou non.

Commençons par la créer sans paramètres.

8- Le constructeur

index.php > ...

```
1  <?php
2      class Vehicule
3      {
4          public $marque;
5          protected $couleur;
6
7          public function __construct()
8          {
9
10         }
11         public function getMarque(){return $this->marque;}
12         public function getCouleur(){return $this->couleur;}
13
14         public function setMarque($marque){$this->marque = $marque;}
15         public function setCouleur($couleur){$this->couleur = $couleur;}
16     }
17
18     $vehicule0 = new Vehicule();
19     $vehicule0->setMarque("Renault");
20     echo $vehicule0->getMarque() . "\n";
21
```

echo \$vehicule0->getMarque() . "\n";

8- Le constructeur

Dans cet exemple, le constructeur n'a pas de paramètres.

```
$vehicule0 = new Vehicule;
```

Ou

```
$vehicule0 = new Vehicule();
```

Ces 2 déclarations veulent dire la même chose mais on préférera la seconde.

Dans l'exemple qui va suivre,
nous allons maintenant ajouter des paramètres.

8- Le constructeur

index.php > ...

```
1  <?php
2      class Vehicule
3      {
4          private $marque;
5          private $couleur;
6
7          public function __construct($marque, $couleur)
8          {
9              $this->marque = $marque;
10             $this->couleur = $couleur;
11         }
12         public function getMarque(){return $this->marque;}
13         public function getCouleur(){return $this->couleur;}
14     }
15
16     $vehicule0 = new Vehicule("Renault", "grise");
17     echo $vehicule0->getMarque() . "\n"; //Renault
18     echo $vehicule0->getCouleur() . "\n"; //grise
19
```

8- Le constructeur

Le constructeur de la classe s'attend donc à prendre en paramètres et respectivement, la marque et la couleur du véhicule.

Lors de la création d'un objet sur la base de la classe Vehicule, il faut donc obligatoirement lui passer en paramètres ces 2 éléments et dans le bon ordre.

Ensuite, on récupère les valeurs passées en paramètres et on les sauvegarde dans les attributs concernés.

8- Le constructeur

index.php > ...

```
1  <?php
2  class Vehicule
3  {
4      private $marque;
5      private $couleur;
6
7      public function __construct($marque, $couleur)
8      {
9          $this->marque = $marque;
10         $this->couleur = $couleur;
11     }
12     public function getMarque(){return $this->marque;}
13     public function getCouleur(){return $this->couleur;}
14 }
15
16 $vehicule0 = new Vehicule("Renault", "grise");
17 $vehicule1 = new Vehicule("Peugeot", "bleue");
18 $vehicule2 = new Vehicule("Ferrari", "rouge");
19
20 echo $vehicule0->getMarque() . "\n"; //Renault
21 echo $vehicule1->getMarque() . "\n"; //Peugeot
22 echo $vehicule2->getMarque() . "\n"; //Ferrari
```

9- Les méthodes et attributs statiques

Ce sont des méthodes qui agissent sur la classe et non sur l'objet.

On ne retrouvera donc **jamais** l'opérateur **\$this** dans ce type de méthodes.

Pour appeler une méthode statique, l'opérateur utilisé est le double deux points ::

Il est également possible d'appeler cette méthode sur un objet.

9- Les méthodes et attributs statiques

```
<?php
class Vehicule
{
    private $marque;
    private $couleur;
    private $vitesse;
    private $nb_roues;
    private $proprietaire;

    public static function rouler()
    {
        echo "Ce vehicule peut rouler \n";
    }
}

Vehicule::rouler(); //Ce vehicule peut rouler

$vehicule1 = new Vehicule;
$vehicule1->rouler(); //Ce vehicule peut rouler
```

```
>>
```

```
>>
```

```
$vehicule1->rouler(); //Ce vehicule peut rouler
```

9- Les méthodes et attributs statiques

index.php > ...

```
1  <?php
2      class Vehicule
3      {
4          private $marque;
5          private $couleur;
6          private static $nbre_instances = 0;
7
8          public function __construct($marque, $couleur)
9          {
10             $this->marque = $marque;
11             $this->couleur = $couleur;
12             self::$nbre_instances++;
13         }
14         public function getMarque(){return $this->marque;}
15         public function getCouleur(){return $this->couleur;}
16         public static function getNbreInstances(){return self::$nbre_instances;}
17     }
18
19     $vehicule0 = new Vehicule("Renault", "grise");
20     $vehicule1 = new Vehicule("Peugeot", "bleue");
21     $vehicule2 = new Vehicule("Ferrari", "rouge");
22     echo Vehicule::getNbreInstances(); //3
23     $vehicule0->getNbreInstances();
24
25     $vehicule0->getNbreInstances();
```

10- L'héritage

En poo, l'héritage consiste à créer une nouvelle classe (enfant) à partir d'une classe existante (parente). Ce qui permet de réutiliser les attributs et les méthodes de la classe parente.

On dit que cette nouvelle classe **hérite** de la classe mère.

Une classe mère peut avoir plusieurs filles mais une classe fille ne peut avoir qu'une classe mère en PHP.

10- L'héritage

index.php > ...

```
1  <?php
2
3  class Vehicule
4  {
5      private $marque = "Ferrari";
6
7      public function getMarque()
8      {
9          return $this->marque;
10     }
11 }
12
13 class Voiture extends Vehicule
14 {
15
16 }
17
18 $voiture0 = new Voiture;
19 echo $voiture0->getMarque(); // Ferrari
20
```

30

10

```
echo $voiture0->getMarque(); // Ferrari
```

10- L'héritage

index.php > ...

```
1  <?php
2      class Vehicule
3      {
4          private $marque = "Ferrari";
5
6          public function getMarque()
7          {
8              return $this->marque;
9          }
10     }
11
12     class Voiture extends Vehicule{}
13
14     class Moto extends Vehicule{}
15
16     $voiture0 = new Voiture;
17     echo $voiture0->getMarque() . "\n"; // Ferrari
18
19     $moto0 = new Moto;
20     echo $moto0->getMarque() . "\n"; // Ferrari
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

10- L'héritage

Dans cet exemple, Voiture et Moto sont des classes enfants et qui ont pour parent la classe Vehicule. Elles ont donc hérité de tous les attributs et méthodes de la classe Véhicule.

L'héritage a été réalisé grâce au mot clé **extends**.

Voiture et Moto possèdent désormais :

- * l'attribut \$marque ainsi que sa valeur
- * La méthode getMarque();

10- L'héritage

Le concept d'héritage ne s'arrête pas à la duplication du contenu d'une classe.

Là où cela devient intéressant, c'est que la classe fille peut ajouter ses propres attributs et méthodes (en plus de celles dont elle a hérité).

10- L'héritage

```
#!/php
class Vehicule
{
    private $marque = "BMW";
    private $couleur;
    private $nb_roues;

    public function getMarque()
    {
        return $this->marque;
    }
}

class Voiture extends Vehicule{
    private $position_volant = "gauche";
    public function getPositionVolant()
    {
        return $this->position_volant;
    }
}

class Moto extends Vehicule{
}

$voiture1 = new Voiture;
$moto1 = new Moto;

echo $voiture1->getPositionVolant(); //gauche
echo $moto1->getPositionVolant(); //Error: Call to undefined method Moto::getPositionVolant()
```

```
echo $moto1->getPositionVolant(); //Error: Call to undefined method Moto::getPositionVolant()
echo $voiture1->getPositionVolant(); //gauche
```


11- Redéfinition d'une méthode ou d'un attribut dans une classe

- La classe **parente A** possède une méthode **m()**.
- La classe **filles A1** qui **hérite de A** possède automatiquement la même méthode **m()**.

La classe fille **A1** peut modifier le contenu de sa méthode **m()**.

On parle alors de redéfinition ou de surcharge.

11- Redéfinition d'une méthode ou d'un attribut dans une classe

```
<?php
class Vehicule{
    private $marque;
    private $couleur;
    private $nb_roues;

    public function afficher_Message()
    {
        echo "Ceci est un véhicule. ";
    }
}

class Voiture extends Vehicule
{
    public function __construct($marque_param)
    {
        $this->marque = $marque_param;
    }

    public function afficher_Message()
    {
        parent::afficher_Message();
        echo "Plus précisément, une voiture de marque: " . $this->marque . ".";
    }
}

$voiture1 = new Voiture("Fiat");
$voiture1->afficher_Message();
//Ceci est un véhicule. Plus précisément, une voiture de marque: Fiat.
```

12- Classes et méthodes abstraites

Une classe abstraite est une classe qui ne peut pas être instanciée.

C'est à dire que **l'on ne peut pas faire**: `$obj = new ClasseAbstraite();`
On ne peut donc pas créer des objets à partir d'une classe.

Une classe abstraite peut avoir des méthodes abstraites et des méthodes non abstraites.

Il n'est pas possible de restreindre le mode de visibilité d'une méthode abstraite dans sa classe fille.

12- Classes et méthodes abstraites

```
<?php
abstract class Vehicule
{
    abstract function typeVehicule();
}

class Voiture extends Vehicule
{
    public function typeVehicule()
    {
        echo "Ceci est un vehicule de type Voiture";
    }
}

$voiture1 = new Voiture();
echo $voiture1->typeVehicule(); //Ceci est une voiture et de marque: BMW
```

```
echo $voiture1->typeVehicule(); //Ceci est une voiture et de marque: BMW
```

13- Classes finales

Comme son nom l'indique, une classe finale ne peut pas avoir de classes filles.

Les méthodes finales ne peuvent pas être redéfinies ou surchargées.

Au niveau de la syntaxe, il faut simplement ajouter le mot clé **final** devant les classes ou les fonctions désirées finales.

13- Classes finales

```
<?php
```

```
class Vehicule  
{
```

```
}
```

```
final class Moto extends Vehicule  
{
```

```
}
```

```
class MotoEnfant extends Moto  
{
```

```
}
```

```
//Fatal error: Class MotoEnfant may not inherit from final class (Moto)
```

```
<?php  
class Vehicule  
{  
    public static $nombre = 1;  
}  
class Moto extends Vehicule  
{  
    public static $nombre = 2;  
}  
class MotoEnfant extends Moto  
{  
    public static $nombre = 3;  
}
```

14- Les interfaces

Les interfaces peuvent être considérées comme des classes totalement abstraites.

Elles ne peuvent donc pas être instanciées.

Toutes les méthodes d'une interface doivent être publiques et redéfinies au sein des classes dans lesquelles l'interface est implémentée.

Les méthodes que contient une interface ne peuvent être ni abstraites ni finales.

11- Les interfaces

```
<?php
interface Actions
{
    public function enAutomatique();
}

class Vehicule
{
    protected $marque;
    protected $couleur;
    protected $proprietaire;
}

class Voiture extends Vehicule implements Actions
{
    public function enAutomatique()
    {
        echo "Boite de vitesse automatique.";
    }
}

$voiture1 = new Voiture();
echo $voiture1->enAutomatique(); //Boite de vitesse automatique.
```

```
echo $voiture1->enAutomatique(); //Boite de vitesse automatique.
```


11- Les traits

Les traits représentent un mécanisme qui permet **d'intégrer** du **code** d'un fichier **indépendant dans** une ou plusieurs **autres classes**.

C'est un peu comme faire le “**include**” d'un fichier dans un autre.

Les traits ressemblent énormément aux interfaces mais il n'est pas possible de les instancier; Seulement de les utiliser.

Ils regroupent généralement des méthodes réutilisables dans plusieurs classes différentes qui ont des points communs.

11- Les traits

```
#!/php
trait Auteur
{
    protected $nom_auteur;
    public function getAuteur(){return $this->nom_auteur;}
}

class Ecrivain
{
    use Auteur;
    public function __construct($nom)
    {
        $this->nom_auteur = $nom;
    }
}

class Peintre
{
    use Auteur;
    public function __construct($nom)
    {
        $this->nom_auteur = $nom;
    }
}
```

11- Les traits

```
<?php
```

```
$ecrivain = new Ecrivain("J. K. Rowling");  
$peintre  = new Peintre("Pierre Soulages");
```

```
echo $ecrivain->getAuteur() . "<br/>";  
echo $peintre->getAuteur() . "<br/>";
```

```
echo $peintre->getAuteur() . "<br/>";  
echo $ecrivain->getAuteur() . "<br/>";
```