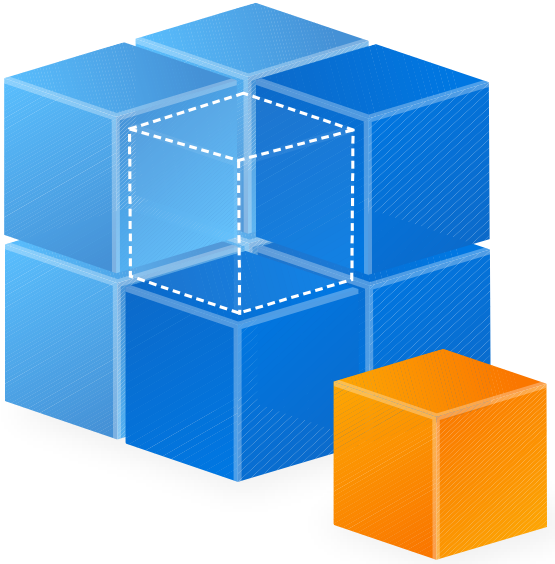
*Découvrir la*

# Programmation Orientée Objet



# Test Comparatif

## Bloc de code procédural

```
print($string);
```

versus

## Bloc de code orienté objet

```
$string->print();
```

# Imaginons...

Un bloc de (pseudo) code qui reçoit une suite de formes géométriques diverses, et qui a pour tâche de dessiner chacune d’entre elles.

# Programmation Orientée Procédure

```
# Généraliste: divise une tâche complexe en tâches simples.
fonction dessineFormes(conteneur formes, sortie écran) {
    pour chaque forme dans conteneur:
        si forme est carré:    exécute dessineCarré(   forme, sortie écran);
        si forme est cercle:   exécute dessineCercle(  forme, sortie écran);
        si forme est triangle: exécute dessineTriangle(forme, sortie écran);
        ...
}


# Spécialiste: divise une tâche simple en suite d'instructions.
fonction dessineCarré(   forme, sortie écran) { ... }
fonction dessineCercle(  forme, sortie écran) { ... }
fonction dessineTriangle(forme, sortie écran) { ... }
...


# Outil: divise une instruction en suite d'opérations.
fonction dessineDroite(x1, y1, x2, y2, sortie écran)          { ... }
fonction dessineCourbe(x, y, largeur, hauteur, sortie écran) { ... }
...
```

# Procédure: la faille du généraliste

*Le Procédural s'exprime au conditionnel,*
*et à la première* personne*:*

Si tel type est ceci, je fais comme ceci;
Si tel type est cela, je fais comme cela;
… et ainsi de suite pour *chaque* type.

Si n types sont ajoutés, il y aura:

- n modifications à faire dans le généraliste*;*
- n spécialistes à créer, et à implémenter;
- Des outils de plus en plus spécialisés, en nombre croissant.

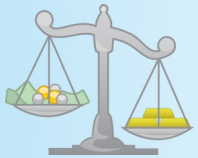
Chaque ajout de *fonctionnalités* va exiger**:**

- Un généraliste supplémentaire;
- *Autant* de spécialistes qu'il y a de types connus;
- Toujours plus d'outils, toujours plus *spécialisés*.

# Programmation Orientée Objet

```
fonction dessineFormes(conteneur formes, sortie écran) {  # Généraliste
    pour chaque Forme dans conteneur:
        Forme->dessine(sortie écran);
}

# Aucun spécialiste déclaré dans ce bloc de code.
```

# And the winner is: Object



*L'objet s'exprime avec l'impératif, et à la deuxième personne:*

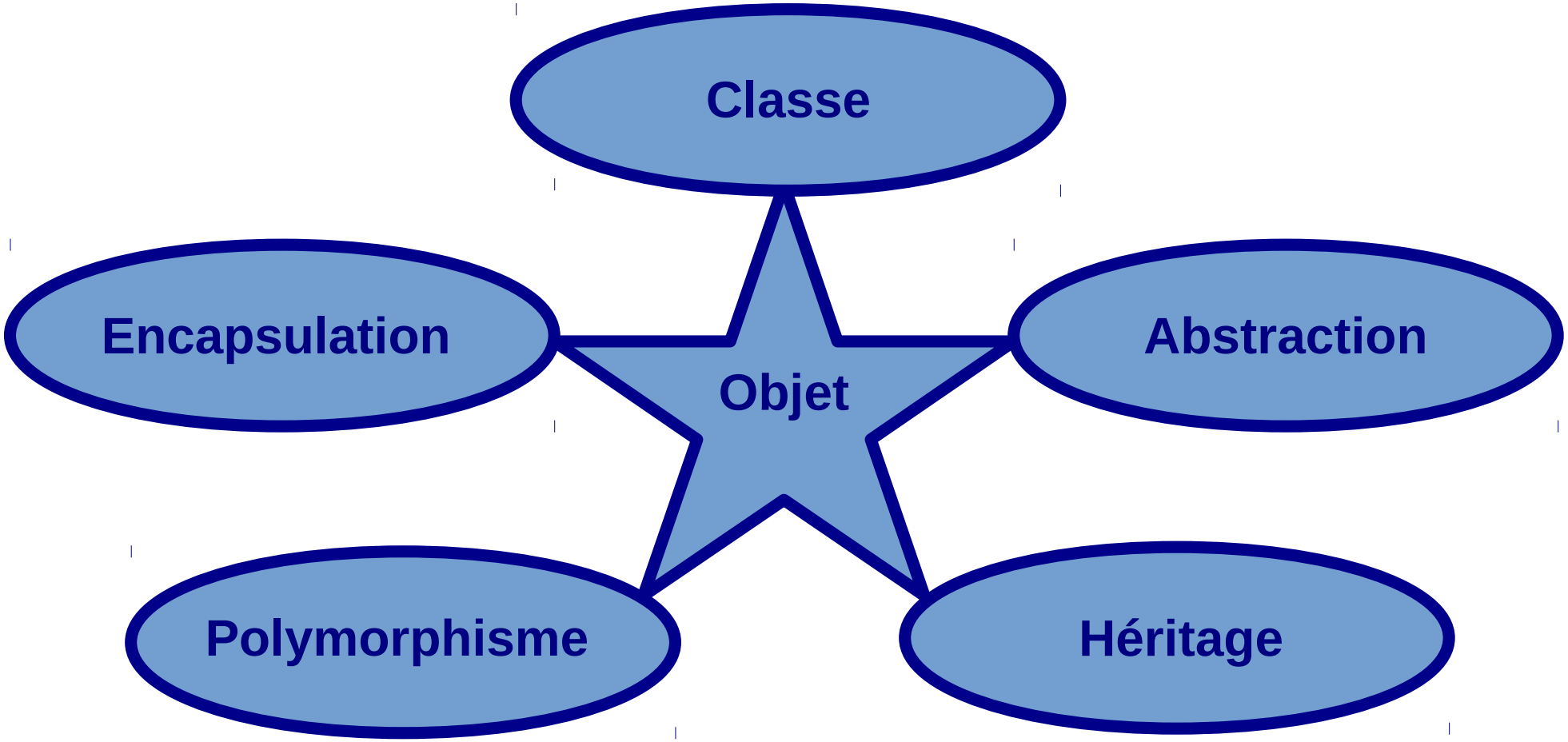
Puisque tu sais de quel type tu es, et *comment* tu fonctionnes, dessine-toi tout seul.

*Ou plus simplement:* Dessine-toi !

L'abstraction ne se fait plus par conditions (si), mais par *généralisation:*

- Même si d'autres types sont ajoutés par la suite, ce généraliste lui ne changera jamais.
- Si d'autres *fonctionnalités* sont ajoutées, elles seront implémentées dans le corps du code qui déclare les types, pas dans celui qui les utilise.

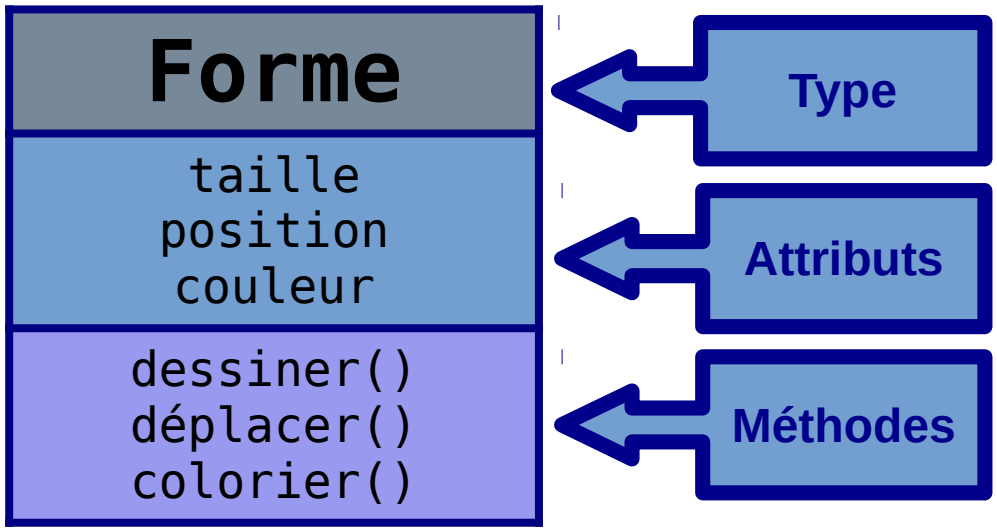
# POO : Les cinq piliers



# Tout commence avec l'objet

**Un objet est structuré en trois parties :**

- Un type, exprimé par le *nom* de l'objet;
- Un état, exprimé par des *attributs*;
- Un comportement, exprimé par des *méthodes.*



# Ça, c'est la classe !

Une classe déclare le code qui spécifie un objet (autrement dit, un *type*).

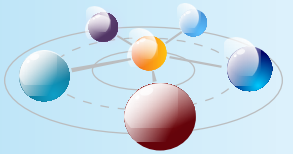
Si l'objet est un cake, alors la classe est la *recette* d'un objet de type Cake: Avec la même recette, on peut réaliser plusieurs cakes distincts.

Couper une part du cake **X** laissera entier le cake **Y**, parce que chaque objet se trouve dans un espace mémoire distinct : Chacun d'entre eux est *l'instance* d'une classe.

# Exemple de classe en PHP

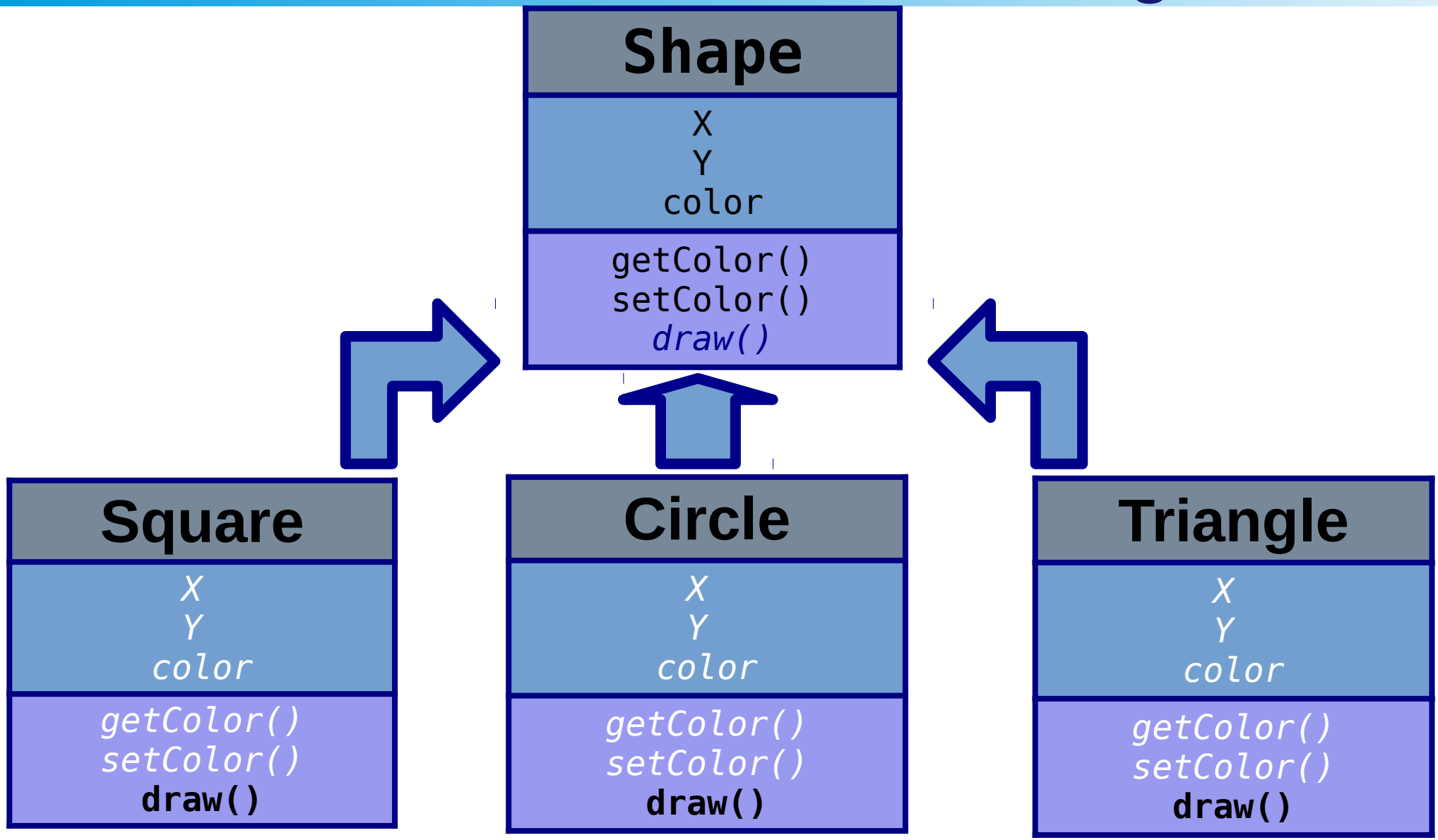
```php
class Shape { # type généraliste concret

    private $x;             # Attribut: position horizontale
    private $y;             # Attribut: position verticale
    private $color;         # Attribut: couleur

    # Constructeur: initialise l'instance avec des valeurs d'attributs
    # Exécuté par l'instruction '$object = new Shape()' qui réalise l'instance.
    public __construct(int $x, int $y, string $color) {
        $this->x     = $x;
        $this->y     = $y;
        $this->color = $color;
    }

    # Accesseur: lit la valeur d'un attribut
    public getColor() : string { return $this->color; }

    # Mutateur: écrit la valeur d'un attribut
    public setColor(string $color) : void { $this->color = $color; }

     # Méthode spécialisée: pas implémentable dans un type général!
    public draw(Screen $output) : void { ??? }
}
```

# Puis vint l'abstraction



```
abstract class Shape { # type généraliste abstrait (donc incomplet)

    protected $x;          # Ces attributs sont déclarés protected
    protected $y;          # pour permettre un accès privilégié à
    protected $color;      # mes spécialistes, et seulement eux.

    # Une classe abstraite ne peut pas être instanciée directement:
    # Ce constructeur ne peut être appelé que par celui d'un spécialiste.
    public __construct(int $x, int $y, string $color) {
        $this->x     = $x;
        $this->y     = $y;
        $this->color = $color;
    }

    # Accesseur: lit la valeur d'un attribut
    public getColor() : string { return $this->color; }

    # Mutateur: écrit la valeur d'un attribut
    public setColor(string $color) : void { $this->color = $color; }

    # Méthode spécialisée abstraite: à charge aux spécialistes d'implémenter !
    public abstract draw(Output $stream) : void;
}
```

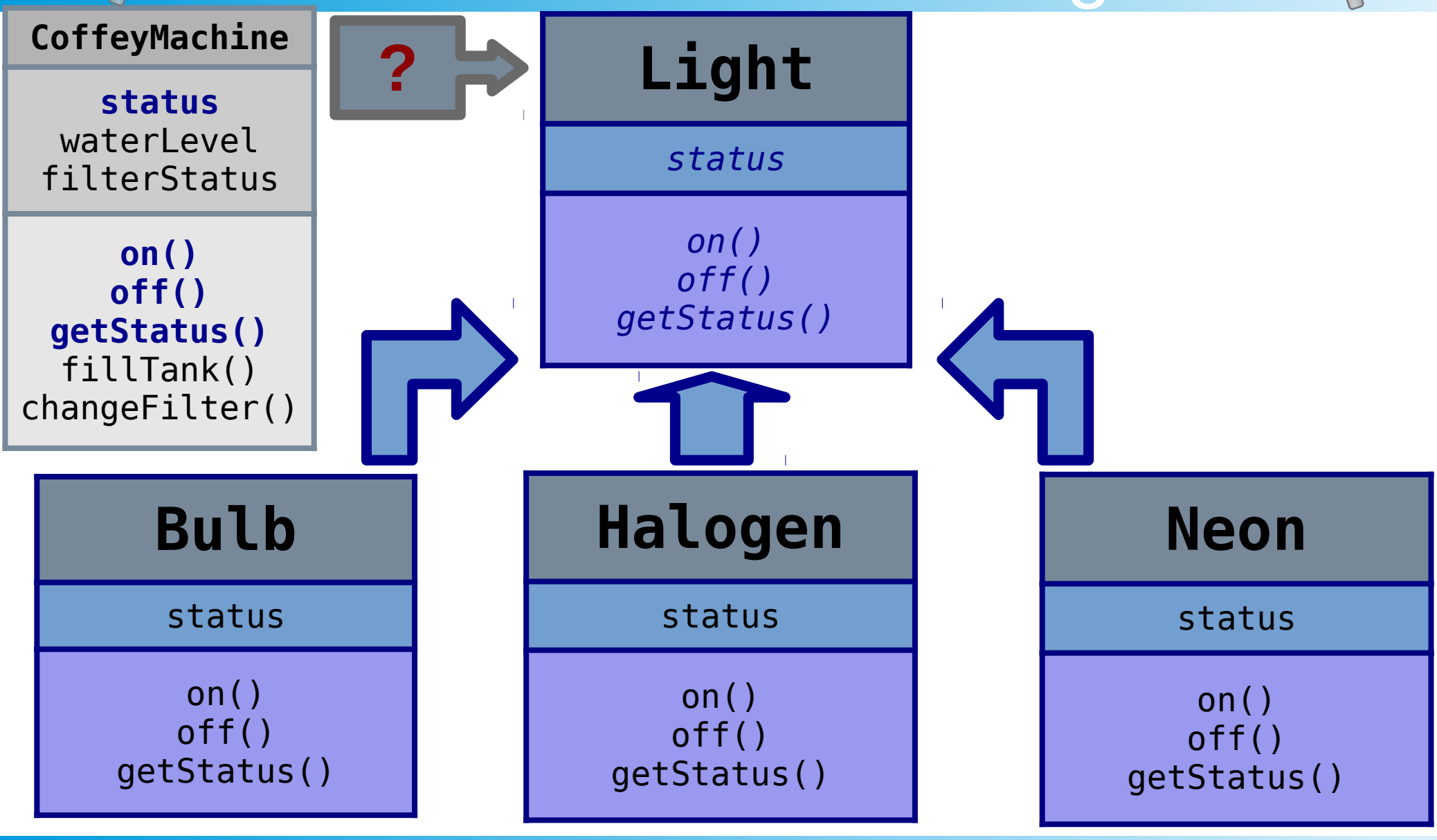
# De l’abstraction à l’héritage



# Exemple d'une classe spécialiste

```
# Type spécialiste concret: Shape ne connaît pas Circle, ni aucun descendant
class Circle extends Shape {

    # un spécialiste peut aussi déclarer des attributs qui lui sont propres.
    protected $radius;

    # Un objet s'adresse à sa classe mère avec le mot-clé 'parent'
    public __construct(int $x, int $y, string $color, int $radius) {
        $this->radius = $radius;
        parent::__construct($x, $y, $color);
    }

    # Méthode spécialisée dans un type spécialiste: no problemo !
    public draw(Screen $output) : void {
        # Implémentation réalisable
    }
}
```

# Les limites de l'héritage



# Supposons...

… Que mon programme à besoin d'intégrer un objet cafetière (`CoffeyMachine`).

Je réalise que cet objet a une interface similaire avec les objets luminaires (type `Light`) : un état allumé ou éteint, des méthodes pour allumer ou éteindre.


**Question:** dois-je déclarer le type `CoffeyMachine` comme descendant du type `Light` pour partager la même interface?

# Réponse...

**Non !**

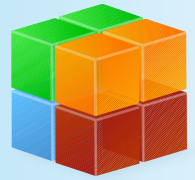
Une cafetière n'est **pas** un luminaire, donc l'héritage n'est pas la solution. Si je l'utilisais, ça serait comme dire: « *une cafetière est un luminaire spécialisé* », ce qui serait plutôt bizarre.

Et puis ma cafetière a d'autres méthodes propres à **son** type (changer le filtre, remplir le réservoir) : On ne peut décemment pas demander à un *luminaire* de faire ces choses-là.

OK, mais si je veux allumer (ou éteindre) des ampoules, **et** des cafetières, j'ai *besoin* d'une interface commune pour manipuler ces types, sans tenir compte de leurs spécialisations.

Alors comment faire ?

# Abstract++: Le polymorphisme



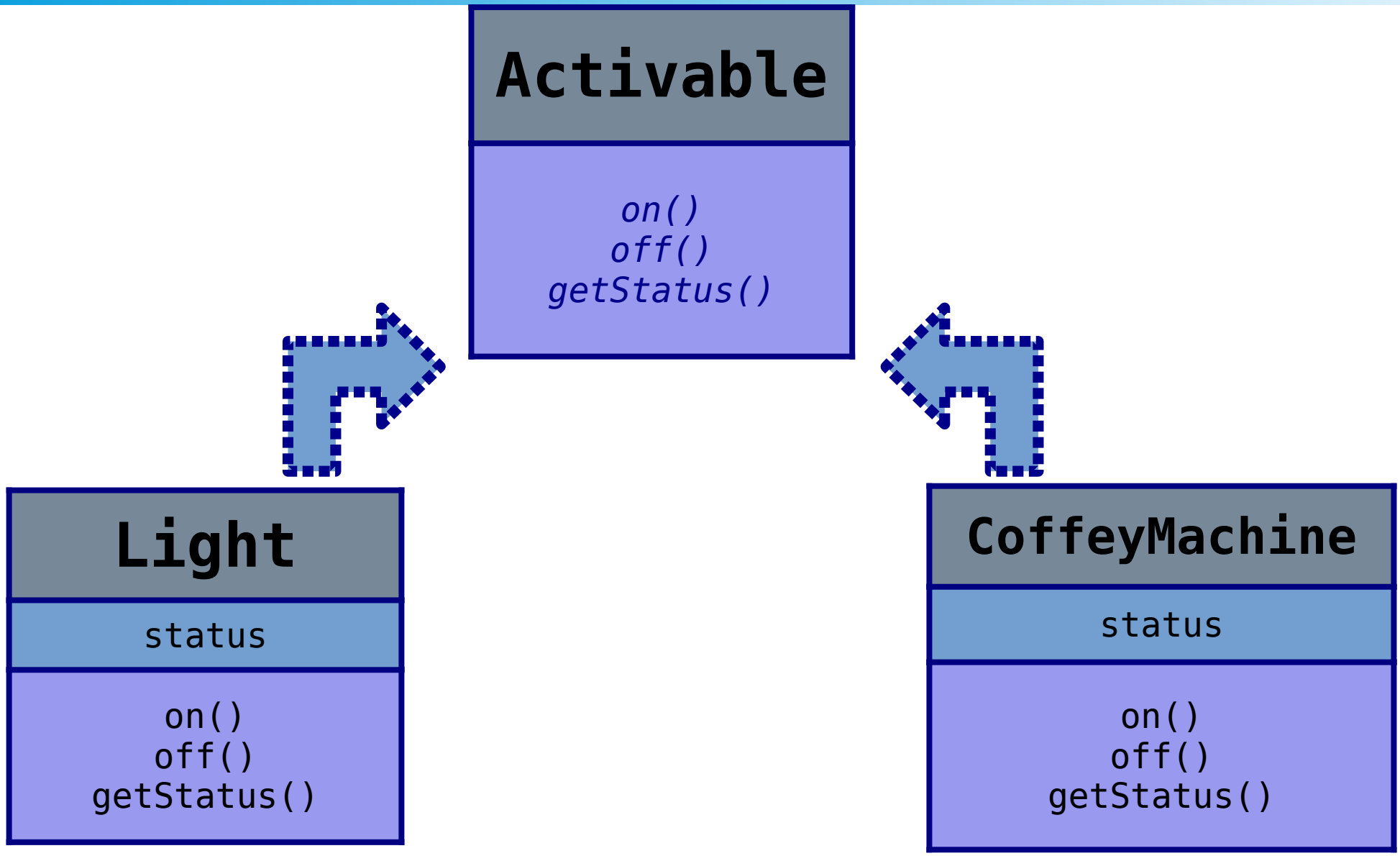
Si j'ai besoin d'une *interface* commune entre plusieurs types hétérogènes, il me suffit simplement de la déclarer, comme telle:

```
# Pure type abstrait généraliste
interface Activable {

    public on() : void;

    public off() : void;

    public getStatus() : string;
}
```

Une interface ne contient *ni* attributs, *ni* méthodes concrètes, mais seulement la définition d'un type totalement abstrait.

Mes ampoules et mes cafetières seront désormais exprimables avec une interface commune: Le type `Activable`.

# Modèle d'interface



# Interface : Implémentation

```php
abstract class Light implements Activable {
    ...
}

class CoffeyMachine implements Activable {
    ...
}


# Usage procédural externe
# $device peut être aussi bien une ampoule qu'une cafetière: peu importe !
function start(Activable $device) : void {
    $device->on();
}
```

Abstraite ou concrète, une classe peut implémenter autant d'interfaces que l'on veut. Une interface peut elle même hériter de *plusieurs* autres interfaces.

Si ma classe implémente **n** interfaces, alors je pourrais typer de **n + 1** façons différentes les objets instanciés par cette classe.

Telle est la puissance du *polymorphisme*.

# Et voilà l'encapsulation

Tant qu'il est développé ou maintenu, le code évolue continuellement. Une encapsulation est une *organisation* du code, qui doit accomplir deux choses:

- Totalement isoler un code-service de changements faits dans un code-client;
- Protéger (autant que possible) le code-client de changements faits dans un code-service.

Elle est à la fois locale et globale: elle s'applique sur *tout* élément ou ensemble de code: attributs, méthodes, classes, hiérarchies, composants, jusqu'à l'ensemble du programme.

Un code encapsulé est plus robuste, résilient et adaptatif aux changements.

# Techniques d'encapsulation

## Interdire au client l'accès *direct* aux attributs d'une classe

- Par défaut, je déclare toujours un attribut `private`.
- Si mon type est généraliste, je déclare l'attribut `protected` pour que ses spécialistes en hérite (si besoin).
- L'accès aux attributs se fait toujours à travers des méthodes qui *elles*, sont `public`.

Raison: Si un client a l'accès direct, et change une valeur d'attribut, l'objet ne peut pas vérifier si cette nouvelle valeur est non seulement *valide*, mais surtout *consistante* avec son état.

Et si l'état d'un objet devient inconsistant dans ces conditions, personne ne saura ni le quoi, ni le comment, ni le pourquoi !

# Techniques d'encapsulation

Séparer le *quoi*, et le *comment*

- Je n'ai pas besoin de savoir comment fonctionne une voiture pour la conduire;
- En POO, c'est pareil: je n'ai pas besoin de savoir comment un objet est implémenté pour m'en servir.

Raison: Une implémentation peut être modifiée à tout moment, et aussi par n'importe qui. Si mon code est structuré pour utiliser *cette* implémentation, tôt ou tard: il cassera.

# Techniques d'encapsulation

## Séparer ce qui change, et ce qui ne change pas

En résonance avec le principe précédent:

- Une implémentation change, mais une interface ne change pas.
- Un type spécialisé change, mais un type généraliste ne change pas.

Raison: Si mon code-client est correctement structuré pour communiquer avec *une* interface et *seulement* elle, il ne cassera jamais.

# Techniques d'encapsulation

## Utiliser le type le plus général possible

Quand *j'utilise* un type `Square`, je vais plutôt structurer mon code pour communiquer avec le type `Shape`. Le type de base déclare les méthodes communes pour tout ses descendants, qui eux sont des *spécialistes*.

Raison: Si mon code était structuré pour utiliser le type `Square`, et que je décide plus tard de remplacer `Square` par `Cube`, alors je devrais traquer tout les blocs qui utilise `Square` pour les modifier. Youpi.

# Techniques d'encapsulation

## Développer le code par ajout, plutôt que par modification

Ou, autrement dit: il vaut toujours mieux créer un nouveau type, au lieu d'en modifier un autre. *Une* variation de comportement s'exprime toujours dans *un* spécialiste *distinct*, dédié à ce comportement.

Raisons:

- Plusieurs variations implémentées dans le même type sont contradictoires, voire antagonistes avec la *nature* du type;
- Cela obscurcit la représentation qu'un humain pourrait se faire de ce type;
- Cela génère des classes ''couteau-suisse'', implémentées avec des grappes de `if` et de `switch` indigestes.

# L’école est finie !