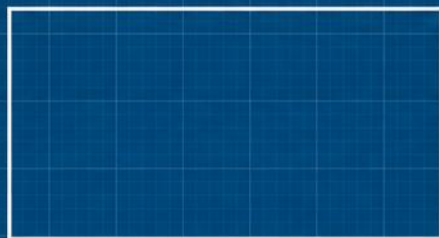
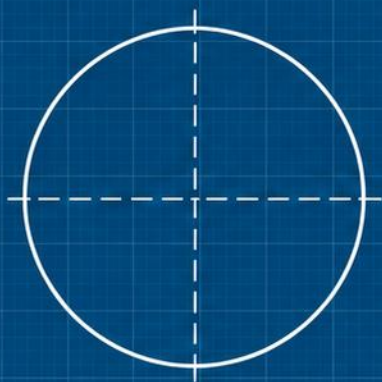


Bloc P00 • chapitre 8

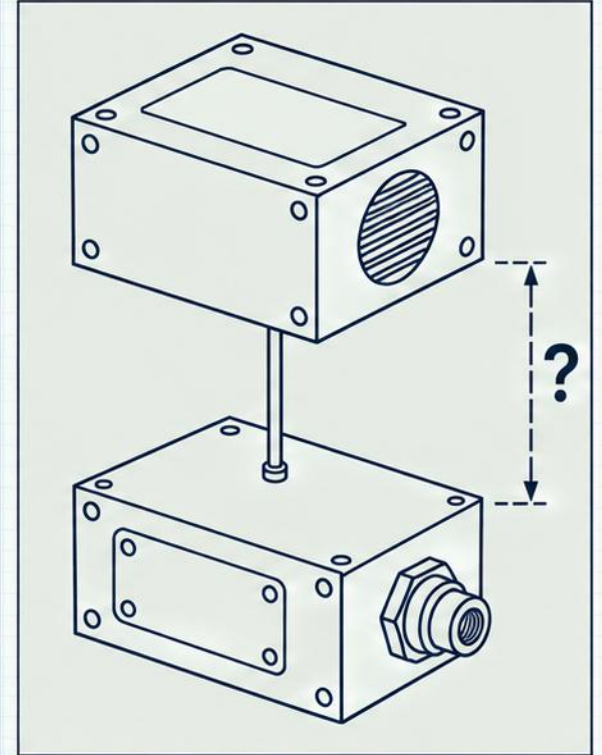
Atelier : dessiner avec des objets

Des points, des formes, un canvas, et à la fin un fichier SVG ouvert dans votre navigateur.



Un JPG est-il un SVG ?

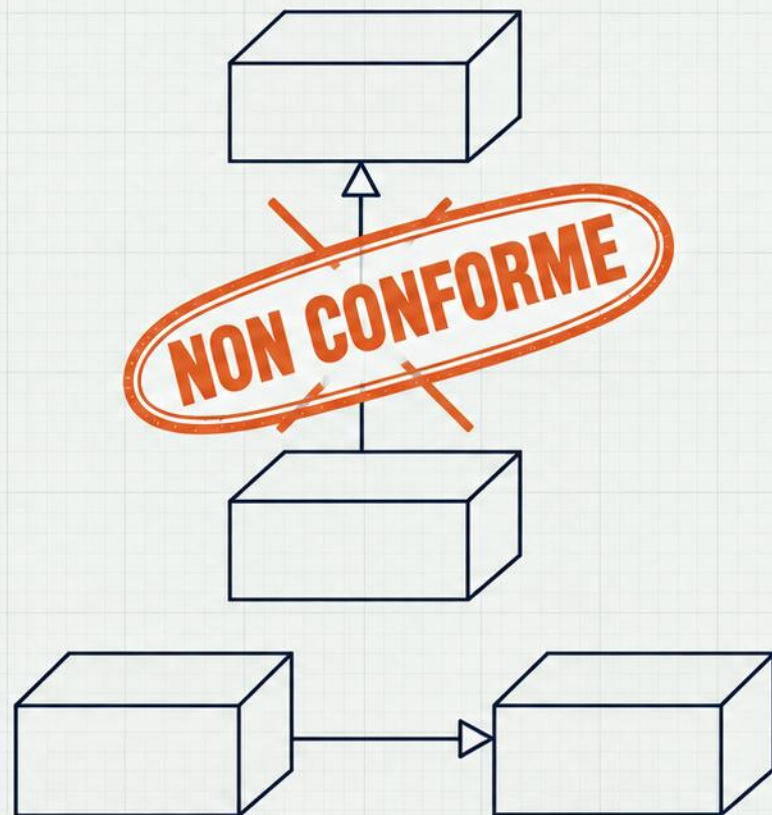
```
1  class SvgRenderer implements Renderer
2  {
3      public function render(): string { /* fabrique le texte SVG */ }
4      public function save(string $path): void { /* écrit le .svg */ }
5  }
6
7  class JpgRenderer extends SvgRenderer implements Renderer
8  {
9      public function save(string $path): void { /* écrit le .jpg */ }
10 }
```



- Cinq minutes, en binôme. Cet extrait vient d'un ancien projet de départ. Un détail d'écriture y trahit un défaut de conception. Lequel ?

Un JPG n'est pas un SVG : cet héritage ne dit rien de vrai

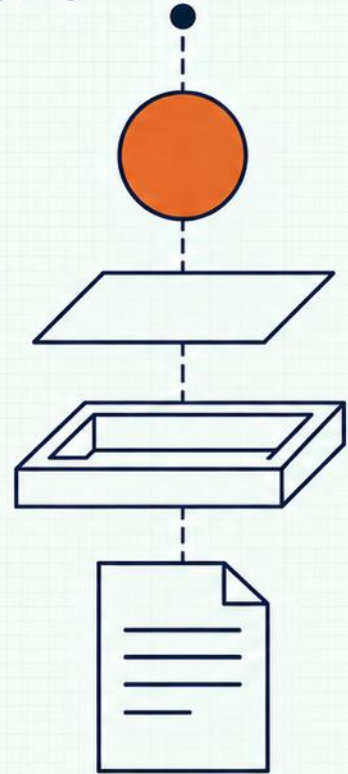
- extends récupère `render()` sans effort, mais pour PHP tout `JpgRenderer` devient aussi un `SvgRenderer`. C'est faux dans la réalité.
- Toute méthode ajoutée demain à `SvgRenderer` arrivera dans `JpgRenderer` sans qu'on l'ait demandée.
- La bonne relation est celle du chapitre 6 : `JpgRenderer` **a un** `SvgRenderer`. Il lui demande le texte SVG, puis le convertit en image.



Devant tout extends, une phrase : « un X est un Y » est-elle vraie ?

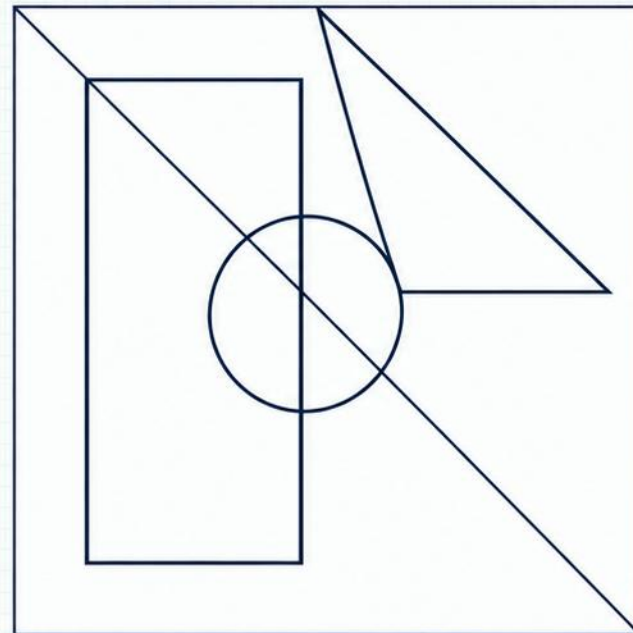
- une valeur qui ne change plus :
un Point, deux coordonnées, readonly.
- un héritage vrai : un cercle **est** une forme,
un rectangle **est** une forme.
- une classe abstraite utile : toutes les formes
ont une aire, chacune la calcule autrement.
- une agrégation : un Canvas **contient** des formes,
il n'en est pas une.
- une interface : Renderer, deux façons
de dessiner les mêmes objets.

Éviter de recopier trois méthodes ne suffit pas
à justifier un héritage.



Un SVG est un fichier texte : cinq balises suffisent

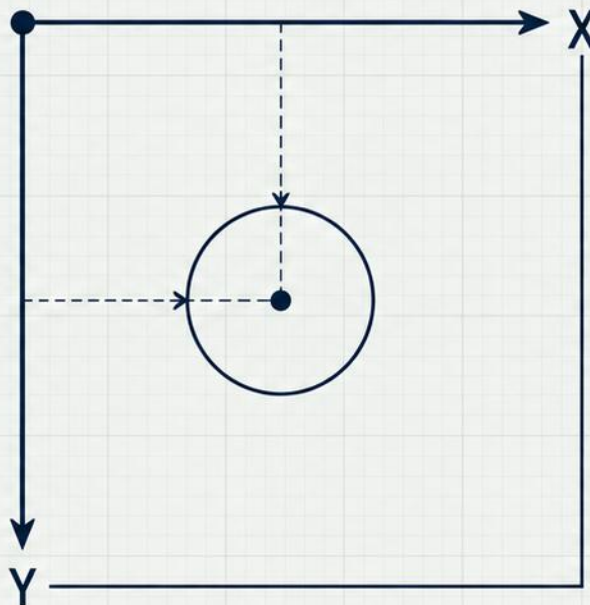
```
1 <svg xmlns="http://www.w3.org/2000/svg" width="500" height="500" viewBox="0 0 500 500">
2   <rect x="0" y="0" width="500" height="500" fill="#FFFFFF" />
3   <line x1="0" y1="0" x2="500" y2="500" stroke="#0000FF" stroke-width="1" />
4   <circle cx="250" cy="250" r="80" fill="#00FF00" />
5   <rect x="50" y="50" width="250" height="400" fill="#FF0000" />
6   <polygon points="250,0 300,200 500,200" fill="#FFFF00" />
7 </svg>
```



- Vous n'allez pas manipuler des pixels. Vous allez écrire des balises, comme en HTML. Ces cinq balises suffisent pour tout l'atelier.

L'origine (0, 0) est en haut à gauche, et l'axe Y descend

- Une figure placée avec les coordonnées d'un repère mathématique apparaît retournée.
- `cx="250"` `cy="250"` : le centre du disque se mesure depuis le coin supérieur gauche.



Le dépôt : tout est rouge, les tests sont le cahier des charges

```
git clone https://github.com/<votre-compte>/formes-26.git
cd formes-26
composer install
composer test
```

```
composer test:etape-1
composer test:etape-2
composer test:etape-3
composer test:etape-4
composer test:bonus
```

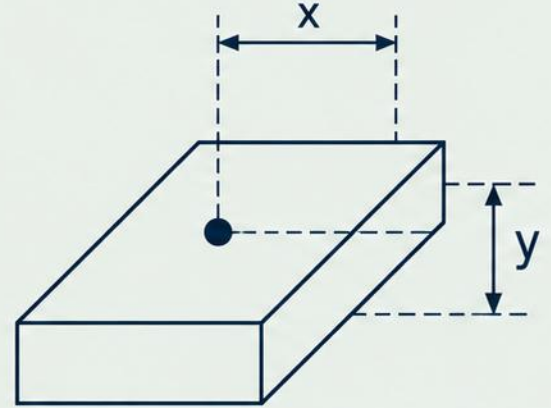


bonus

Les fichiers de `src/` sont des squelettes : chaque corps à écrire lève `LogicException('À implémenter')`. Les tests de `tests/` ne se modifient pas. Envoyez votre travail à la fin de chaque étape : l'onglet Actions de votre fork affiche une coche par étape. Lisez la ligne `Failed asserting that...` avant de modifier votre code.

Étape 1: un Point est une valeur, il ne change plus

```
1  final readonly class Point implements \Stringable
2  {
3      public function __construct(
4          public float $x,
5          public float $y,
6      ) {}
7
8      public function translate(float $dx, float $dy): self
9      {
10         return new self($this->x + $dx, $this->y + $dy);
11     }
12
13     public function __toString(): string
14     {
15         return '(' . $this->x . ', ' . $this->y . ')';
16     }
17
18     // ... distanceTo(), equals()
19 }
```

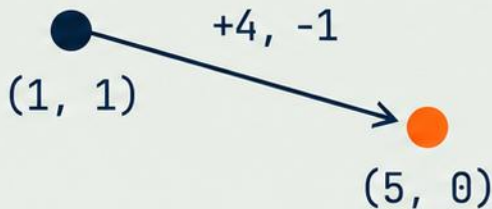


- Un héros perd des points de vie. Un point, non : (3, 4) reste (3, 4). `readonly` devant `class` rend toutes les propriétés `readonly` d'un coup. L'antislash de `\Stringable` dit que l'interface vient de PHP, pas du dossier Shapes.

translate() ne déplace rien : il renvoie un nouveau point

```
1 $depart = new Point(1, 1);  
2 $arrivee = $depart->translate(4, -1);  
3  
4 echo $depart, PHP_EOL;  
5 echo $arrivee, PHP_EOL;
```

```
(1, 1)  
(5, 0)
```

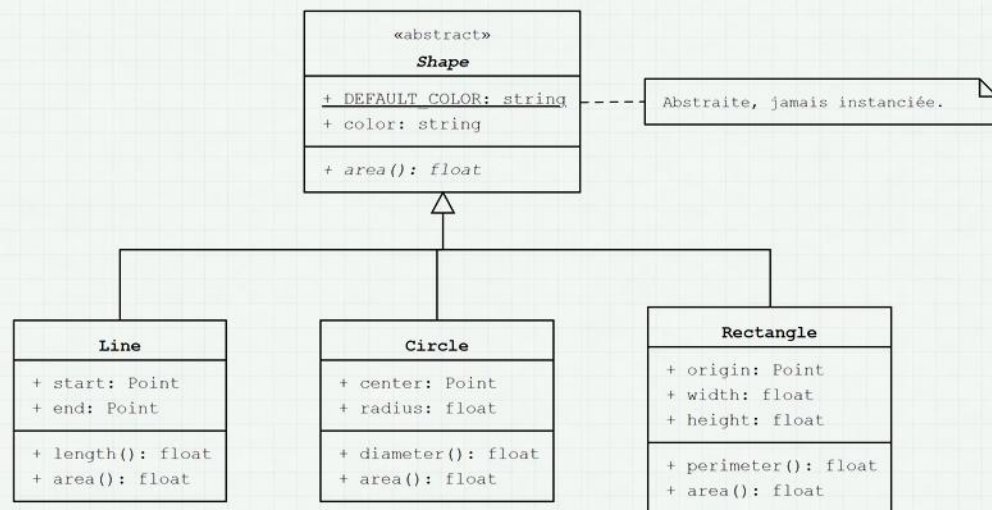


Toute écriture dans un point déjà construit est refusée par PHP :

```
Fatal error: Uncaught Error: Cannot modify readonly property Shapes\Point::$x
```

Indice : ne recopiez pas la formule de la distance dans Line. Point::distanceTo() la connaît déjà.

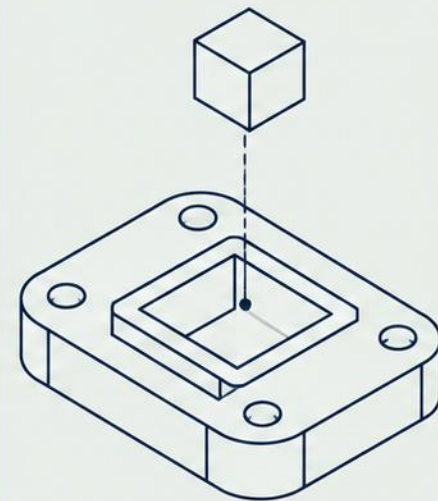
Étape 2 : ce qui est commun monte dans Shape, ce qui diffère reste abstrait



- Un cercle et un rectangle ont deux choses en commun : une couleur et une aire.
- La couleur a du code : Shape la valide et la met en majuscules, une fois pour toutes. 
- L'aire n'en a pas : rien ne se calcule sans savoir de quelle forme il s'agit. `area()` reste **abstraite**.
- Line aussi est une forme : elle rejoint la hiérarchie, avec `area()` à 0.0.

abstract class Shape : une constante, une couleur validée, une méthode sans corps

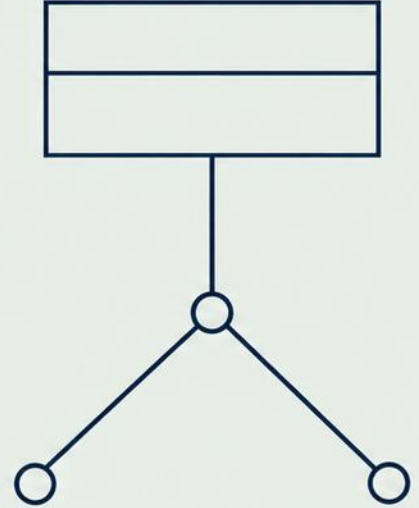
```
1  abstract class Shape
2  {
3      public const string DEFAULT_COLOR = '#000000';
4
5      /** La couleur, au format #RRGGBB, toujours en majuscules. Lisible partout, écrivez ici une fois. */
6      public readonly string $color;
7
8      public function __construct(string $color = self::DEFAULT_COLOR)
9      {
10         if (preg_match('/^#[0-9A-Fa-f]{6}$/', $color) !== 1) {
11             throw new \InvalidArgumentException("Couleur invalide : {$color}");
12         }
13
14         $this->color = strtoupper($color);
15     }
16
17     abstract public function area(): float;
18 }
```



- `public const string DEFAULT_COLOR` : une **constante de classe**, lue avec `self::` dedans et `Shape::` ailleurs.
- `preg_match()` vérifie que le texte suit un motif : un `#` puis six chiffres hexadécimaux.
- `$color` **n'est pas promu**, c'est voulu : on vérifie, puis on écrit la version en majuscules. Une promotion aurait écrit la valeur brute avant toute vérification.

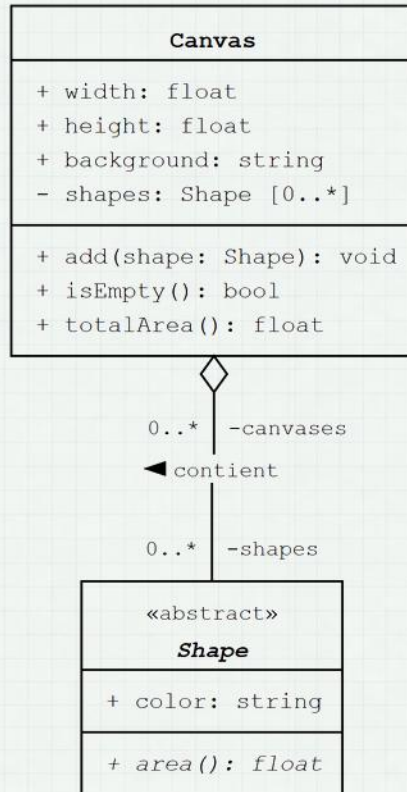
Line extends Shape : la fille appelle parent::__construct() elle-même

```
1  final class Line extends Shape
2  {
3      public function __construct(
4          public readonly Point $start,
5          public readonly Point $end,
6          string $color = self::DEFAULT_COLOR,
7      ) {
8          parent::__construct($color);
9      }
10
11     public function area(): float
12     {
13         return 0.0;
14     }
15
16     // ... length()
17 }
```



- Supprimez la couleur en double de Line, puis relancez les tests de l'étape 1 : ils doivent rester verts pendant que vous déplacez ce code. PHP n'appelle pas le constructeur parent à votre place.

Étape 3 : un Canvas contient des formes, il n'en est pas une

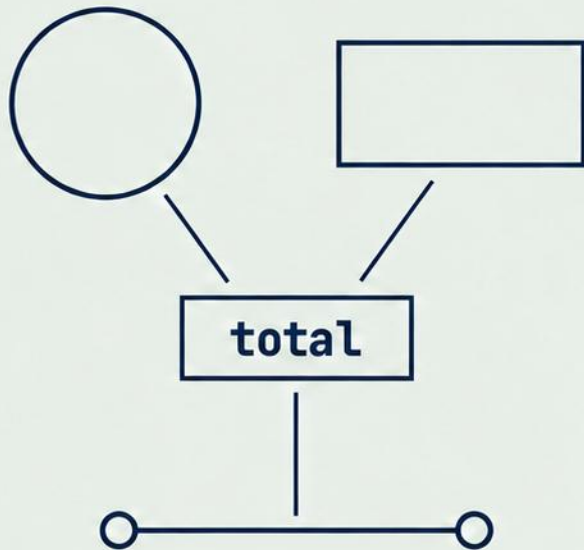


- Losange vide : les formes sont construites à l'extérieur et données par `add()`. Une même forme peut aller dans deux canvas, et elle survit à la disparition de l'un.
- Pas de triangle entre Canvas et Shape : « un canvas est une forme » ne tient pas quand on le dit à voix haute.
- `add(Shape $shape)` n'a aucun if : le type du paramètre refuse un Point avec un `TypeError`.



totalArea() ne demande jamais à une forme ce qu'elle est

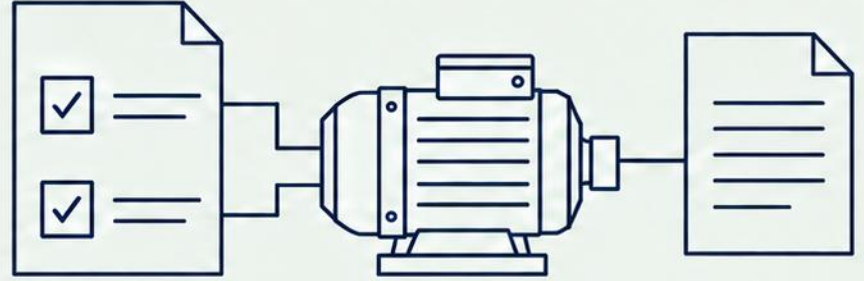
```
1 public function totalArea(): float
2 {
3     $total = 0.0;
4
5     foreach ($this->shapes as $shape) {
6         $total += $shape->area();
7     }
8
9     return $total;
10 }
```



- À chaque tour de boucle, PHP appelle **la méthode** `area()` de l'objet parcouru : celle de `Circle`, de `Rectangle` ou de `Line`. C'est le polymorphisme du chapitre 6, et il évite d'écrire un cas par forme.

Étape 4 : Renderer, deux méthodes et aucune ligne de code

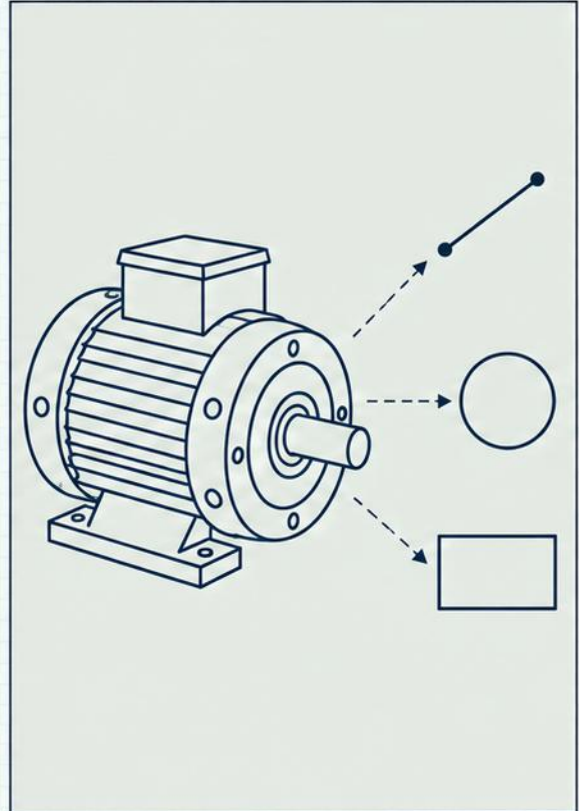
```
1 interface Renderer
2 {
3     public function render(): string;
4
5     public function save(string $path): void;
6 }
```



- Comme Fighter au chapitre 7 : l'interface dit ce qu'un moteur de rendu doit savoir faire, pas comment.
- SvgRenderer reçoit le canvas dans son constructeur. Il ne le fabrique pas, il l'utilise pour produire du texte.
- Le document commence par `<?xml version="1.0" encoding="UTF-8"?>`, le fond est peint en premier, les formes suivent dans l'ordre d'ajout.
- `save()` écrit dans le fichier exactement ce que `render()` renvoie.

renderShape() : match (true) traduit chaque forme en balise

```
1 private function renderShape(Shape $shape): string
2 {
3     return match (true) {
4         $shape instanceof Line => sprintf(
5             '<line x1="%s" y1="%s" x2="%s" y2="%s" stroke="%s" stroke-width="1" />',
6             $this->number($shape->start->x),
7             $this->number($shape->start->y),
8             $this->number($shape->end->x),
9             $this->number($shape->end->y),
10            $shape->color,
11        ),
12        // ... Circle, Rectangle, Polygon ...
13        default => throw new \InvalidArgumentException(
14            'Forme inconnue du renderer : '.$shape::class,
15        ),
16    };
17 }
```

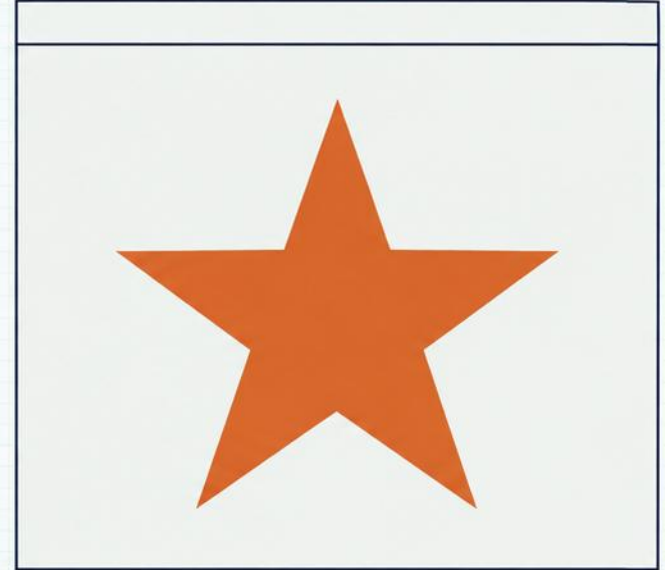


- Le chapitre 6 déconseillait la cascade de `instanceof`. Ici c'est un choix de conception : le format de sortie n'appartient pas aux formes, connaître chaque type est le métier du moteur de rendu. Le `default` lève une exception : une forme inconnue se signale, elle ne dessine pas du vide.

Le résultat se regarde : ouvrez `exampLes/star.svg` dans votre navigateur

```
php exemples/star.php
```

```
Étoile écrite dans exemples/star.svg (2 formes, aire totale : 77527 px²).
```



- Ensuite, modifiez `exampLes/star.php` et dessinez ce que vous voulez : une maison, un logo, une fractale. Une seule contrainte : au moins trois types de formes différents.

À vous : quatre étapes vertes, une étoile, puis votre dessin

Le rendu

1. Les quatre étapes vertes, en local et dans l'onglet Actions de votre fork.
2. `php exemples/star.php` écrit l'étoile.
3. Modifiez `exemples/star.php` et dessinez ce que vous voulez, avec au moins trois types de formes.
4. Postez votre SVG sur le canal du cours, avec le lien vers votre fork.

Sur papier, avant chaque étape suivante : les quatre questions de fin d'étape du chapitre.

Au choix : `src/Ellipse.php` et son cas dans `renderShape()`.

Bonus : `JpgRenderer`, qui **a un** `SvgRenderer` au lieu d'en hériter, avec l'extension GD.

Bonus difficile : `Canvas::withBackground()`, un nouveau canvas sans modifier l'original.

