Programmation orientée objet en PHP*

Partie 2 : Structurez des objets avec l'héritage

[R3.01 : Développement web]

1 Procédez à un héritage

Vous savez à présent créer des objets ! Mais les objets seuls ne proposent qu'une encapsulation de code. Il faut commencer à **structurer votre code pour réduire les répétitions et donner du sens aux objets.**

Voyons maintenant la mécanique de la programmation orientée objet qui nous aidera à réduire la duplication de code : l'héritage. L'**héritage** nous permet d'accéder aux propriétés et méthodes d'une classe "parent" depuis les "enfants". Voyons comment cela marche à travers notre fil rouge MatchMaker.

1.1 (Re)Découvrez la mécanique de l'héritage sur un exemple

Regardons ce code, permettant de définir un utilisateur :

```php
<?php

declare(strict_types=1);

class User
{
    private const STATUS_ACTIVE = 'active';
    private const STATUS_INACTIVE = 'inactive';

    public function __construct(public string $username,
            private string $status = self::STATUS_ACTIVE)
    {
    }

    public function setStatus(string $status): void
    {
        if (!in_array($status, [self::STATUS_ACTIVE, self::STATUS_INACTIVE]))
        {
```

*https://openclassrooms.com

```
            trigger_error(sprintf(
            'Le status %s n\'est pas valide. Les status possibles sont : %s',
             $status,
             implode(', ',
             [self::STATUS_ACTIVE, self::STATUS_INACTIVE])), E_USER_ERROR);
        };

        $this->status = $status;
    }

    public function getStatus(): string
    {
        return $this->status;
    }
}
```

Expliquons le code ci-dessus : Un utilisateur est défini par un pseudo et par un statut (actif ou inactif). Nous avons la possibilité de récupérer le statut avec `getStatus` et nous pouvons modifier le statut avec `setStatus` (qui déclenche une erreur si le statut n'est ni "actif", ni "inactif").

A présent créons un administrateur :

```
<?php

declare(strict_types=1);

class Admin
{
    private const STATUS_ACTIVE = 'active';
    private const STATUS_INACTIVE = 'inactive';

    // Ajout d'un tableau de roles pour affiner les droits des administrateurs :)
    public function __construct(public string $username,
            private array $roles = [], private string $status = self::STATUS_ACTIVE)
    {
    }

    public function setStatus(string $status): void
    {
        if (!in_array($status, [self::STATUS_ACTIVE, self::STATUS_INACTIVE]))
        {
            trigger_error(sprintf(
            'Le status %s n\'est pas valide. Les status possibles sont : %s',
             $status,
             implode(', ',
             [self::STATUS_ACTIVE, self::STATUS_INACTIVE])), E_USER_ERROR);
        };
```

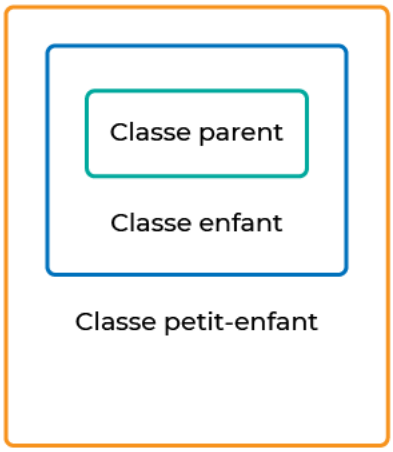
```
        $this->status = $status;
    }

    public function getStatus(): string
    {
        return $this->status;
    }

    // Méthode d'ajout d'un rôle, puis on supprime les doublons avec array_filter.
    public function addRole(string $role): void
    {
        $this->roles[] = $role;
        $this->roles = array_filter($this->roles);
    }

    // Méthode de renvoie des rôles, dans lequel on définit le rôle ADMIN par défaut.
    public function getRoles(): array
    {
        $roles = $this->roles;
        $roles[] = 'ADMIN';

        return $roles;
    }

    public function setRoles(array $roles): void
    {
        $this->roles = $roles;
    }
}
```

Voyez-vous des **choses similaires** entre ces deux classes `Admin` et `User` ? Dans les deux nous retrouvons des propriétés similaires, comme le nom et le statut. Il y a aussi des méthodes en commun pour la gestion du statut. La **seule** différence est l'ajout d'une gestion de rôles dans l'admin. **C'est dommage d'avoir à répéter autant de code**.

En imaginant créer une troisième classe `Joueur`, il est très probable que nous retrouverions de nouveau des similitudes avec le `User`. **L'héritage va nous permettre de supprimer les parties dupliquées**. Voyons maintenant comment faire en étendant les classes.

1.2 Étendez une classe

Nous allons modifier notre classe `Admin` pour qu'elle bénéficie du code de la classe `User`. Avec l'héritage, la classe hérite des propriétés, ainsi que des méthodes des classes "parentes".



En PHP, comme en C# (contrairement au C++ ou au Python) une classe ne peut hériter que d'un parent à la fois, qui lui-même ne pourra avoir qu'un parent au maximum, etc.

Pour faire en sorte que notre classe `Admin` hérite de la classe `User`, nous allons employer le mot clé `extends`, suivi du nom de la classe que nous souhaitons étendre : `Admin extends User`.

Appliquons cela à notre exemple :

```php
<?php

declare(strict_types=1);

class User
{
    public const STATUS_ACTIVE = 'active';
    public const STATUS_INACTIVE = 'inactive';

    public function __construct(public string $username,
            public string $status = self::STATUS_ACTIVE)
    {
    }

    public function setStatus(string $status): void
    {
        if (!in_array($status, [self::STATUS_ACTIVE, self::STATUS_INACTIVE]))
        {
            trigger_error(sprintf(
            'Le status %s n\'est pas valide. Les status possibles sont : %s',
```

```
            $status,
            implode(', ',
            [self::STATUS_ACTIVE, self::STATUS_INACTIVE])), E_USER_ERROR);
        };

        $this->status = $status;
    }

    public function getStatus(): string
    {
        return $this->status;
    }
}

class Admin extends User
{
    // Ajout d'un tableau de roles pour affiner les droits des administrateurs :)
    public function __construct(public string $username,
            public array $roles = [],
            public string $status = self::STATUS_ACTIVE)
    {
    }

    // Méthode d'ajout d'un rôle, puis on supprime les doublons avec array_filter.
    public function addRole(string $role): void
    {
        $this->roles[] = $role;
        $this->roles = array_filter($this->roles);
    }

    // Méthode de renvoie des rôles, dans lequel on définit le rôle ADMIN par défaut.
    public function getRoles(): array
    {
        $roles = $this->roles;
        $roles[] = 'ADMIN';

        return $roles;
    }

    public function setRoles(array $roles): void
    {
        $this->roles = $roles;
    }
}
```

Le code est maintenant bien plus **léger** à lire. Puisque nous avons éliminé du code, c'est aussi **moins de risque d'erreur, d'oubli lors d'une mise à jour, un code simplifié et moins de**

mémoire utilisée par PHP. Que du bon en fait !

Toutes les méthodes, les propriétés et les constantes de la classe **parente** (`User`) seront accessibles dans la classe **enfant** (`Admin`).

Dans cet exemple, nous avons utilisé l'héritage entre 2 classes. Nous pourrions continuer ainsi de manière infinie afin d'avoir un enfant d'un enfant, d'un enfant, d'un enfant, ..., d'une classe parente.

2 Profitez des propriétés et méthodes héritées

Maintenant que vous savez pourquoi et comment mettre en place l'héritage, nous allons découvrir comment profiter des propriétés et méthodes héritées.

2.1 Accédez aux propriétés de la classe parente

Lorsque vous utilisez l'héritage, depuis l'objet comme depuis la classe, vous pouvez accéder aux propriétés de la classe parente de la même manière qu'avant, avec la flèche `->`.

| Testez le code ci-après.

```php
<?php

declare(strict_types=1);

class User
{
    public const STATUS_ACTIVE = 'active';
    public const STATUS_INACTIVE = 'inactive';

    public function __construct(public string $username,
            public string $status = self::STATUS_ACTIVE)
    {
    }
}

class Admin extends User
{
    // ...

    public function printStatus()
    {
        // vous pouvez accéder au statut comme si la propriété appartenait à Admin
```

```
        echo $this->status;
    }
}

$admin = new Admin('Lily');
$admin->printStatus();
```

2.1.1 Propriétés statiques

Il en va de même pour les propriétés statiques.

Rappelez-vous, le mot clé `static` fait référence à la **classe**, et non à l'objet.

Depuis la classe enfant, vous pourrez faire référence aux mêmes propriétés statiques que le parent, peu importe que vous fassiez référence à la classe enfant ou à la classe parente. En revanche, vous ne pourrez pas faire référence aux propriétés (statiques ou non) de l'enfant depuis le parent.

Pour faire référence à un parent, vous devez utiliser le nouveau mot clé `parent`. Il permet de faire référence à une classe parente. En revanche, lorsque votre classe hérite d'une classe, qui elle-même hérite d'une classe, qui elle-même. . .bref, vous ne pouvez pas cibler la classe parente désirée.Avec le mot clé `parent` vous remontez l'arbre généalogique, jusqu'à trouver l'élément parent auquel vous faites référence. S'il s'agit d'une méthode, elle peut alors elle aussi faire référence à un parent.

Testez le code ci-après.

```
<?php

declare(strict_types=1);

class User
{
    public const STATUS_ACTIVE = 'active';
    public const STATUS_INACTIVE = 'inactive';

    public static $nombreUtilisateursInitialisés = 0;

    public function __construct(public string $username,
            public string $status = self::STATUS_ACTIVE)
    {
    }
}
```

```
class Admin extends User
{
    public static $nombreAdminInitialisés = 0;

    // MàJ des valeurs des propriétés statiques de la classe courante avec `self`,
    // et de la classe parente avec `parent`
    public static function nouvelleInitialisation()
    {
        self::$nombreAdminInitialisés++; // classe Admin
        parent::$nombreUtilisateursInitialisés++; // classe User
    }
}

Admin::nouvelleInitialisation();
var_dump(Admin::$nombreAdminInitialisés,
        Admin::$nombreUtilisateursInitialisés,
        User::$nombreUtilisateursInitialisés);
var_dump(User::$nombreAdminInitialisés); // ceci ne marche pas.
```

Prenez le temps d'analyser le code utilisé ci-dessus. C'est important de bien avoir cette mécanique en tête pour la suite !

2.2 Accédez aux méthodes de la classe parente

En ce qui concerne les méthodes de classe, depuis un objet, tout comme les accès aux propriétés, rien ne change. Vous pourrez accéder aux méthodes de la même manière qu'avant, avec la flèche `->` comme vous pouvez le voir dans le code ci-après.

Testez le code ci-dessous.

```
<?php

declare(strict_types=1);

class User
{
    public const STATUS_ACTIVE = 'active';
    public const STATUS_INACTIVE = 'inactive';

   public function __construct(public string $username,
```

```
            public string $status = self::STATUS_ACTIVE)
    {
    }

    public function printStatus()
    {
        echo $this->status;
    }
}


class Admin extends User
{
    // ...

    public function printCustomStatus()
    {
        echo "L'administrateur {$this->username} a pour statut : ";
        $this->printStatus(); // appelle printStatus du parent depuis l'enfant
    }

}


$admin = new Admin('Lily');

// Affiche "L'administrateur Lily a pour statut : active"
$admin->printCustomStatus();

// printStatus n'existe pas dans la classe Admin,
// donc printStatus de la classe User sera appelée grâce à l'héritage
$admin->printStatus();
```

2.2.1 Surchargez une méthode

PHP vous permet également de réécrire une méthode existante d'un parent, dans une classe enfant. On parle de **surcharge**.

On appelle méthode réécrite une méthode surchargée, parce qu'il s'agit la plupart du temps de lui donner une charge de travail supplémentaire. Une raison supplémentaire est que certains langages de programmation autorisent d'avoir plusieurs fois le même nom de méthode dans une classe, chacune ayant un nombre différent d'arguments. Le nombre d'arguments décide de la méthode à appeler. Ce n'est pas possible en PHP.

Que vous permet de faire PHP, et que vous impose-t-il lorsque vous réécrivez/surchargez une méthode existante dans une classe parente ? Pour commencer, sa signature doit rester compatible avec la méthode d'origine :

- vous ne pouvez **pas enlever** des arguments ;
- vous pouvez **ajouter** un argument **uniquement** s'il est **optionnel** ;
- Vous pouvez **changer** le type d'un **argument uniquement** s'il est **compatible** avec le type d'origine ;
- vous pouvez **changer** le type de **retour** de la méthode **uniquement** s'il est **compatible** avec le type d'origine.

Lorsque vous surchargez une méthode, vous pouvez choisir d'exploiter la méthode parente **et** d'ajouter un comportement supplémentaire, **ou** de choisir de complètement réécrire le comportement. Dans le premier cas, lorsque vous souhaitez appeler la méthode parente, plutôt que la flèche, il faut utiliser `parent::`.

| Testez le code ci-après.

```php
<?php

declare(strict_types=1);

class User
{
    public const STATUS_ACTIVE = 'active';
    public const STATUS_INACTIVE = 'inactive';

    public function __construct(public string $username,
            public string $status = self::STATUS_ACTIVE)
    {
    }

    public function setStatus(string $status): void
    {
        if (!in_array($status, [self::STATUS_ACTIVE, self::STATUS_INACTIVE]))
        {
            trigger_error(sprintf(
            'Le status %s n\'est pas valide. Les status possibles sont : %s',
            $status,
            implode(', ',
            [self::STATUS_ACTIVE, self::STATUS_INACTIVE])), E_USER_ERROR);
        };

        $this->status = $status;
```

```
    }

    public function getStatus(): string
    {
        return $this->status;
    }
}

class Admin extends User
{
    public const STATUS_LOCKED = 'locked';

    // la méthode est entièrement réécrite ici
    // seule la signature reste inchangée
    public function setStatus(string $status): void
    {
        if (!in_array($status,
        [self::STATUS_ACTIVE, self::STATUS_INACTIVE, self::STATUS_LOCKED]))
        {
            trigger_error(sprintf(
            'Le status %s n\'est pas valide. Les status possibles sont : %s',
            implode(', ',
            [self::STATUS_ACTIVE, self::STATUS_INACTIVE, self::STATUS_LOCKED])),
            E_USER_ERROR);
        }

        $this->status = $status;
    }

    // utilise la méthode de la classe parente et ajoute un comportement
    public function getStatus(): string
    {
        return strtoupper(parent::getStatus());
    }
}

$admin = new Admin('Paddington');
$admin->setStatus(Admin::STATUS_LOCKED);
echo $admin->getStatus();
```

Comme vous avez pu le voir dans l'exemple précédent ciblant une propriété parente, lorsqu'une méthode est héritée depuis une classe enfant, vous pouvez choisir :

- d'appeler la méthode de la classe courante avec `$this` (ou `self` pour les méthodes statiques) ;
- ou bien de cibler la méthode parente avec le mot clé `parent::`.

Notez que si la méthode a été "surchargée" (réécrite) plusieurs fois dans l'arbre généalogique, le mot clé `parent::` ne permet pas de préciser si vous souhaitez faire appel à la méthode du parent, ou bien du grand-parent, etc. PHP va remonter l'arborescence de classes héritées, et utiliser la première définition de la méthode qu'il trouve. Méthode qui elle-même pourra éventuellement faire appel à la méthode parente.

2.3 Travail à faire

Reprenons notre projet fil rouge.

Déposez dans votre dossier de travail `php-poo` le script `index.php` archivé dans le fichier `Etud.zip` de ce TP.

Si vous testez ce script, vous devez voir l'erreur suivante :

Fatal error: Uncaught Error: Class "QueuingPlayer" not found in C:\wamp64\www\php8-poo\index.php on line *33*

Error: Class "QueuingPlayer" not found in C:\wamp64\www\php8-poo\index.php on line *33*

Call Stack

#	Time	Memory	Function	Location
1	0.0242	362704	{main}( )	...\index.php:0
2	0.0242	362920	Lobby->addPlayers(...$players = *variadic*(class Player { protected string $name = 'greg'; protected float $ratio = 400 }, class Player { protected string $name = 'jade'; protected float $ratio = 476 }))	...\index.php:75
3	0.0242	363296	Lobby->addPlayer($player = class Player { protected string $name = 'greg'; protected float $ratio = 400 })	...\index.php:39

Ce fichier `index.php`, contient le code d'une "salle d'attente" (la classe `Lobby`), ainsi que le code d'un joueur (la classe `Player`). Lorsqu'un joueur s'enregistre dans le Lobby, il devient un joueur en attente.
Un joueur en attente possède une propriété `range` qui est un entier. Le but de cette propriété est d'accroître la portée de la recherche d'un adversaire, lorsqu'aucun ne correspond au niveau du joueur. L'objectif étant de trouver un adversaire, quitte à ce qu'il soit plus faible ou plus fort.

Votre tâche est de :

- créer une classe `QueuingPlayer` qui étend la classe `Player` ;
- et de lui ajouter la propriété `range`.

Voici le résultat que vous devez obtenir :

```
C:\wamp64\www\php8-poo\index.php:95:
array (size=1)
  1 =>
    object(QueuingPlayer)[5]
      protected int 'range' => int 1
      protected string 'name' => string 'jade' (length=4)
      protected float 'ratio' => float 476
```

Créez une nouvelle version de votre projet avec l'enchaînement de commandes ci-après :

```
> git add --all
> git commit -m "Profitez des propriétés et méthodes héritées"
> git push -u origin main
```

3 Contrôlez l'accès aux propriétés et aux méthodes

Bravo ! Vous commencez à structurer votre code avec de plus en plus d'expertise.

Souvent, lorsque vous allez obtenir une technique donnant des possibilités d'extension d'un code, vous allez aussi croiser des situations où vous auriez aimé, au contraire, **restreindre les accès** à certaines propriétés ou méthodes. C'est ce que nous allons voir dans ce chapitre.

3.1 Empêchez l'accès aux propriétés et aux méthodes

Nous avons déjà croisé 2 formes de visibilité, publique et privée, dans la première partie (TP précédent). Mais jusqu'ici, c'était uniquement via une seule classe. Que se passe-t-il lorsque je définis une propriété ou une méthode comme privée, puis que j'hérite de cette classe ?

En guise d'exemple, dans le code ci-après, nous avons changé la classe `User` en passant tout en `private`.

Testez le code ci-après et vérifiez qu'il renvoie le message d'erreur suivant :

Fatal error: Uncaught Error: Undefined constant Admin::STATUS_ACTIVE in C:\wamp64\www\php8-poo\index.php on line *39*

Error: Undefined constant Admin::STATUS_ACTIVE in C:\wamp64\www\php8-poo\index.php on line *39*

Call Stack

#	Time	Memory	Function	Location
1	0.0287	362672	{main}( )	...\index.php:0
2	0.0287	362752	Admin->setStatus($status = 'locked')	...\index.php:55

```php
<?php

declare(strict_types=1);

class User
{
    private const STATUS_ACTIVE = 'active';
    private const STATUS_INACTIVE = 'inactive';

    public function __construct(private string $username,
```

```
            private string $status = self::STATUS_ACTIVE)
    {
    }

    private function setStatus(string $status): void
    {
        assert(
            in_array($status, [self::STATUS_ACTIVE, self::STATUS_INACTIVE]),
            sprintf(
            'Le status %s n\'est pas valide. Les status possibles sont : %s',
            $status, implode(', ', [self::STATUS_ACTIVE, self::STATUS_INACTIVE]))
        );

        $this->status = $status;
    }

    private function getStatus(): string
    {
        return $this->status;
    }
}


class Admin extends User
{
    public const STATUS_LOCKED = 'locked';

    // la méthode est entièrement ré-écrite ici
    // seule la signature reste inchangée
    public function setStatus(string $status): void
    {
        assert(
            in_array($status,
            [self::STATUS_ACTIVE, self::STATUS_INACTIVE, self::STATUS_LOCKED]),
            sprintf(
            'Le status %s n\'est pas valide. Les status possibles sont : %s',
            $status,
            implode(', ',
            [self::STATUS_ACTIVE, self::STATUS_INACTIVE, self::STATUS_LOCKED]))
        );

        $this->status = $status;
    }

    // la méthode utilise celle de la classe parente, et ajoute un comportement
    public function getStatus(): string
```

```
    {
        return strtoupper(parent::getStatus());
    }
}


$admin = new Admin('Paddington');
$admin->setStatus(Admin::STATUS_LOCKED);
echo $admin->getStatus();
```

Ce code me renvoie une erreur. Plus aucune des méthodes surchargées de la classe `Admin` ne fonctionne ! PHP m'indique que j'ai tenté d'accéder à une méthode privée, ou encore que les propriétés auxquelles nous tentons d'accéder n'existent pas...

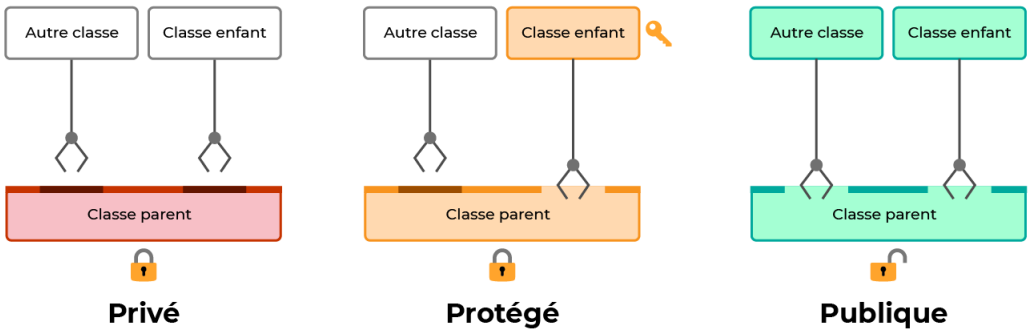
Un élément privé devient uniquement accessible pour la classe dans laquelle il se trouve. L'héritage est interrompu pour cet élément, qu'il s'agisse d'une propriété ou d'une méthode.

Cependant il existe une troisième visibilité située entre `public` et `private` nous apportant un peu de finesse. Voyons ça tout de suite.

3.2 Autorisez l'accès uniquement aux enfants

Cette visibilité s'applique par le biais du mot clé `protected`. Donc pour synthétiser, ?nous avons 3 moyens d'exposer nos propriétés et nos méthodes :

- `public` ouvre l'accès à tous ;
- `private` ferme à tous ;
- `protected` permet de fermer à l'extérieur, mais d'ouvrir à l'héritage.



Testez le code précédent après avoir changé notre classe `User` pour lui appliquer la visibilité `protected`. Cela doit fonctionner et renvoyer le résultat "ACTIVE" comme précédement.

En pratique, ici, les constantes seraient probablement déclarées publiques, les propriétés en protégé, les accesseurs et les mutateurs en public, et les méthodes "internes" (qui ne devraient être utilisées que par la classe elle-même) en privé, ou protégées, selon l'usage.

4 Contraignez l'usage de vos classes

4.1 Imposez l'héritage d'une classe

Lorsque vous voulez garantir un usage particulier, mais laisser les classes enfants décider de la manière dont le code doit fonctionner, alors l'abstraction est LA technique désirée !

Jusqu'à maintenant, nous avons appris à créer des classes, les étendre, surcharger les méthodes, et gérer la visibilité. À présent, ajoutons des mécanismes pour contraindre et assouplir en même temps l'héritage avec l'**abstraction**. C'est très utile pour **anticiper des variations futures** dans notre code, sans les connaître à l'avance. Le code est ainsi beaucoup plus **ouvert aux changements**, et nous ne serons pas contraints à dupliquer des pans entiers de code.

Dans notre logiciel, l'utilisateur est étendu par un administrateur, puis nous avons travaillé avec des classes `Player` et `QueuingPlayer`. Finalement, on se rend compte que les `Player` pourraient eux aussi hériter de `User`, et que jamais nous n'allons instancier la classe `User` seule puisque nos utilisateurs seront forcément soit des joueurs, soit des administrateurs. Une idée serait d'interdire l'usage de `User` seul, et dans le même temps, de forcer son héritage. La classe servira de "squelette" pour les classes suivantes. Elle contient les bases minimalistes nécessaires.

Dans notre code, ceci est représenté par le mot clé `abstract`. Il va venir en préfixe de la déclaration d'une classe :

```php
<?php

abstract class User
{
}
```

Rendez abstraite votre classe `User`, puis essayez de l'instancier et vérifiez que PHP renvoie le message d'erreur suivant :

Fatal error: Uncaught Error: Cannot instantiate abstract class User in C:\wamp64\www\php8-poo\index.php on line *53*				
Error: Cannot instantiate abstract class User in C:\wamp64\www\php8-poo\index.php on line *53*				
Call Stack				
#	Time	Memory	Function	Location
1	0.0366	362624	{main}( )	...\index.php:0

À présent, il n'est donc plus possible d'instancier la classe `User` seule. Le mot clé `abstract` empêche cette action. Ce comportement nous force à étendre la classe pour utiliser son contenu.

Mais ça ne signifie pas pour autant que cette abstraction doive être vide. Notre classe `User` peut rester identique à ce qu'elle était, mais avec le mot clé `abstract` en plus.

Ce qui est pratique avec le mot clé `abstract` qui se situe devant le mot clé `class`, c'est qu'une fois qu'il est là, on peut aussi l'utiliser devant une méthode de classe ! Cela permet de déclarer la signature de la méthode (visibilité, statique ou non, nom, arguments, et type de retour) comme nécessaire, mais nous n'écrivons pas son comportement. Nous terminons là par un ';', nous ne mettons pas d'accolades, et pas de code non plus. Pas tout de suite. Nous exprimons juste que cette méthode doit exister dans les classes enfants, sans dire comment elle fonctionne. C'est à la classe enfant de porter cette responsabilité.

Pour illustrer ce point, dans le code ci-après la méthode `getUsername()` renvoie l'e-mail pour un administrateur, et le pseudo pour un joueur.
Pour tester, créez un joueur et un administrateur puis affichez le résultat de l'appel à cette méthode pour chacun d'eux.

```php
<?php

declare(strict_types=1);

abstract class User
{
    public const STATUS_ACTIVE = 'active';
    public const STATUS_INACTIVE = 'inactive';

    public function __construct(public string $email,
    public string $status = self::STATUS_ACTIVE)
    {
    }

    public function setStatus(string $status): void
    {
        assert(
            in_array($status, [self::STATUS_ACTIVE, self::STATUS_INACTIVE]),
            sprintf(
```

```
            'Le status %s n\'est pas valide. Les status possibles sont : %s',
            $status, [self::STATUS_ACTIVE, self::STATUS_INACTIVE])
        );

        $this->status = $status;
    }

    public function getStatus(): string
    {
        return $this->status;
    }

    abstract public function getUsername(): string;
}

class Admin extends User
{
    // Ajout d'un tableau de roles pour affiner les droits des administrateurs
    public function __construct(string $email,
    string $status = self::STATUS_ACTIVE, public array $roles = [])
    {
        parent::__construct($email, $status);
    }

    // ...

    public function getUsername(): string
    {
        return $this->email;
    }
}

class Player extends User
{
    // Ajout d'un tableau de roles pour affiner les droits des administrateurs
    public function __construct(string $email, public string $username,
    string $status = self::STATUS_ACTIVE)
    {
        parent::__construct($email, $status);
    }

    // ...

    public function getUsername(): string
    {
        return $this->username;
```

```
    }
}
```

Lorsque vous décidez de déclarer des méthodes abstraites, vous devez impérativement écrire leur logique dans une classe enfant, sous peine de recevoir l'erreur suivante :
`Class Admin contains 1 abstract method and must therefore be declared abstract or implement the remaining methods.`
Autrement dit, **chaque classe enfant de** `User` **devra posséder une méthode** `getUsername`. Ou alors cette classe devra elle-même être déclarée abstraite.

Cela ne vous impose pas de l'écrire dans l'enfant direct de votre classe abstraite. Vous pouvez avoir plusieurs classes abstraites qui se suivent. Voici un exemple ci-après.

```php
<?php

declare(strict_types=1);

abstract class User
{
    public const STATUS_ACTIVE = 'active';
    public const STATUS_INACTIVE = 'inactive';

    public function __construct(public string $email,
            public string $status = self::STATUS_ACTIVE)
    {
    }

    public function setStatus(string $status): void
    {
        assert(
            in_array($status, [self::STATUS_ACTIVE, self::STATUS_INACTIVE]),
            sprintf(
            'Le status %s n\'est pas valide. Les status possibles sont : %s',
            $status, [self::STATUS_ACTIVE, self::STATUS_INACTIVE])
        );

        $this->status = $status;
    }

    public function getStatus(): string
    {
        return $this->status;
```

```
    }

    abstract public function getUsername(): string;
}

abstract class Player extends User
{
    // Ajout d'un tableau de roles pour affiner les droits des administrateurs
    public function __construct(string $email, public string $username,
            string $status = self::STATUS_ACTIVE)
    {
        parent::__construct($email, $status);
    }

    // ...
}

class QueuingPlayer extends Player
{
    public function getUsername(): string
    {
        return $this->username;
    }
}

$Qp = new QueuingPlayer('roland@gras.ros', 'Roland');
var_dump($Qp);
```

Grâce à cette mécanique, vous êtes à présent capable de créer des structures **préparées**, avec des attentes particulières, qui devront être étendues et écrites plus tard.

Il nous reste encore une situation : celle où nous souhaitons indiquer à un développeur que nous estimons que le code ne devrait plus évoluer.

4.2 Empêchez l'héritage

Nous avons notre utilisateur, que nous souhaitons étendre. C'est chose faite avec l'administrateur. Mais une fois que nous avons notre administrateur, que pourrait-on lui apporter de plus ? À priori rien. Je décide donc d'interdire à quiconque d'en hériter.

Pour interdire l'héritage d'une classe, nous allons utiliser le mot clé `final`. Il va venir préfixer le mot clé `class`, de la même manière que le mot clé `abstract`.

Testez le code ci-après. Essayez de créer une nouvelle classe qui étend l'Admin. Vous devez obtenir l'erreur suivante :

Fatal error: Class SuperAdmin may not inherit from final class (Admin) in C:\wamp64\www\php8-poo\index.php on line *49*

```php
<?php

declare(strict_types=1);

abstract class User
{
    public const STATUS_ACTIVE = 'active';
    public const STATUS_INACTIVE = 'inactive';

    public function __construct(public string $email,
            public string $status = self::STATUS_ACTIVE)
    {
    }

    public function setStatus(string $status): void
    {
        assert(
            in_array($status, [self::STATUS_ACTIVE, self::STATUS_INACTIVE]),
            sprintf(
            'Le status %s n\'est pas valide. Les status possibles sont : %s',
            $status, implode(', ',[self::STATUS_ACTIVE, self::STATUS_INACTIVE]))
        );

        $this->status = $status;
    }

    public function getStatus(): string
    {
        return $this->status;
    }

    abstract public function getUsername(): string;
}

final class Admin extends User
{
    // Ajout d'un tableau de roles pour affiner les droits des administrateurs
    public function __construct(string $email,
            string $status = self::STATUS_ACTIVE, public array $roles = [])
```

```
    {
        parent::__construct($email, $status);
    }

    public function getUsername(): string
    {
        return $this->email;
    }

    // ...
}

// Ceci est impossible maintenant que la classe Admin est marquée comme finale.
// class SuperAdmin extends Admin {}


$admin = new Admin('trompete@guy.com', 'Ibrahim Maalouf');
var_dump($admin);
```

Vous pouvez non seulement contraindre une classe, mais aussi décider de ne restreindre que certaines méthodes de classe, en les préfixant du mot clé `final`.

4.3 Travail à faire

Modifiez le code pour garantir l'extensibilité des méthodes, en particulier créez une classe abstraite pour les joueurs. Et si vous voyez d'autres modifications utiles, effectuez-les ! N'hésitez pas à prendre une itiniative pour utiliser, par exemple, le mot clé `final`.

Créez une nouvelle version de votre projet avec l'enchaînement de commandes ci-après :

```
> git add --all
> git commit -m "Contraignez l'usage de vos classes"
> git push -u origin main
```

5 Gérez le comportement d'une classe parente

Vous êtes à présent capable de réduire la quantité de code grâce à l'héritage, de toute la finesse nécessaire pour contrôler les différents accès de vos classes grâce à la visibilité, et d'imposer l'écriture d'une méthode ou un héritage avec l'abstraction. Mais il existe encore une situation très commune.

L'usage des valeurs des constantes et des propriétés statiques.

En règle générale, ce n'est pas à la classe parente de connaître le fonctionnement des classes enfants. C'est à l'enfant de s'adapter. Mais avec les constantes et les propriétés statiques, le code peut anticiper les variations de 2 manières différentes. Voyons ça tout de suite.

5.1 Changez la valeur d'une propriété statique ou d'une constante

Lorsqu'il s'agit d'une constante, le but est que la valeur ne change pas : il n'est pas prévu qu'une constante soit écrasée. Cependant, il est possible de définir une nouvelle constante portant le même nom dans une classe enfant.

Testez le code suivant qui illustre ce point tout en conservant les éléments dont nous disposons déjà. Vous devez obtenir l'écran suivant :

```
C:\wamp64\www\php8-poo\index.php:52:
object(Admin)[1]
  public array 'roles' =>
    array (size=0)
      empty
  public string 'email' => string 'michel@petrucciani.com' (length=22)
  public string 'status' => string 'is_active' (length=9)
```

Fatal error: Uncaught AssertionError: Le status is_inactive n'est pas valide. Les status possibles sont : active, inactive in C:\wamp64\www\php8-poo\index.php on line *18*

AssertionError: Le status is_inactive n'est pas valide. Les status possibles sont : active, inactive in C:\wamp64\www\php8-poo\index.php on line *18*

Call Stack

#	Time	Memory	Function	Location
1	0.0294	362664	{main}( )	...\index.php:0
2	0.0296	363136	User->setStatus($status = 'is_inactive')	...\index.php:53
3	0.0296	363456	assert($assertion = FALSE, $description = 'Le status is_inactive n\'est pas valide. Les status possibles sont : active, inactive')	...\index.php:18

```php
<?php

declare(strict_types=1);

abstract class User
{
    public const STATUS_ACTIVE = 'active';
    public const STATUS_INACTIVE = 'inactive';

    public function __construct(public string $email,
            public string $status = self::STATUS_ACTIVE)
    {
    }

    public function setStatus(string $status): void
    {
        assert(
            in_array($status, [self::STATUS_ACTIVE, self::STATUS_INACTIVE]),
            sprintf(
            'Le status %s n\'est pas valide. Les status possibles sont : %s',
```

```
            $status, implode(', ',[self::STATUS_ACTIVE, self::STATUS_INACTIVE]))
        );

        $this->status = $status;
    }

    public function getStatus(): string
    {
        return $this->status;
    }

    abstract public function getUsername(): string;
}

final class Admin extends User
{
    public const STATUS_ACTIVE = 'is_active';
    public const STATUS_INACTIVE = 'is_inactive';

    // Ajout d'un tableau de roles pour affiner les droits des administrateurs
    public function __construct(string $email,
            string $status = self::STATUS_ACTIVE, public array $roles = [])
    {
        parent::__construct($email, $status);
    }

    public function getUsername(): string
    {
        return $this->email;
    }

    // ...
}

$admin = new Admin('michel@petrucciani.com');
var_dump($admin);
$admin->setStatus(Admin::STATUS_INACTIVE);
```

Explications

La valeur du statut de l'administrateur est maintenant `is_active` au lieu de `active`. Le constructeur de la classe `Admin` fait appel à la constante se trouvant dans celle-ci, à l'aide du mot clé `self`.

L'assignation du nouveau statut `Admin::STATUS_INACTIVE` échoue, parce que la méthode `setStatus` fait référence aux constantes se trouvant dans la classe `User`. Les valeurs ne correspondent pas !

Comme ici, il arrive de vouloir utiliser une constante de la classe par défaut et d'utiliser une surcharge des classes enfant s'il y en a. Nous souhaitons utiliser les constantes de `User` par défaut, mais si une classe enfant les redéfinit (comme c'est le cas avec `Admin`), nous souhaitons les utiliser à la place. Pour autoriser cette situation, il faut utiliser un mot clé que vous avez déjà rencontré. Il s'agit de `static`. Au lieu de le placer devant une déclaration de méthode ou de propriété, nous allons l'utiliser en remplacement du mot clé `self`.

`self` fait référence à la classe courante, et permet de faire appel à un élément de cette classe. `static` fait aussi référence à la classe courante, et permet de faire appel à un élément de cette classe, mais si l'élément est redéclaré dans une classe enfant, alors c'est celui-ci qui sera utilisé à la place.

Modifiez donc le code pour faire disparaître le message d'erreur :

```
C:\wamp64\www\php8-poo\index.php:52:
object(Admin)[1]
  public array 'roles' =>
    array (size=0)
      empty
  public string 'email' => string 'michel@petrucciani.com' (length=22)
  public string 'status' => string 'is_active' (length=9)
```

Le mot clé `static` nous permet d'utiliser la déclaration de la constante la plus proche de la classe instanciée, dans la hiérarchie.

Cette technique est appelée **late static binding** ; en français, *résolution statique à la volée*. La résolution des valeurs s'effectue non pas lorsque le code est analysé, mais quand il est exécuté.

5.2 Travail à faire

Admettons qu'une compétition spéciale soit organisée. C'est un serveur en mode rapide, et il va rester en ligne pendant 1 semaine. Les joueurs doivent commencer à 1 200 au lieu de 400, et leur ratio doit évoluer 4 fois plus vite.

Créez une classe `BlitzPlayer` pour l'occasion. Si vous voyez d'autres modifications utiles, effectuez-les.

Créez une nouvelle version de votre projet avec l'enchaînement de commandes ci-après :

```
> git add --all
> git commit -m "Gérez le comportement d'une classe parente"
> git push -u origin main
```