Découvrez la Mécanique de l'Héritage en PHP

1. Intro à la redondance du code

   Prenons l'exemple suivant ou on définit une classe user représentant un utilisateur classique

```
<?php

declare(strict_types=1);

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

    public function __construct(public string $username, p
    {
    }

    public function setStatus(string $status): void
    {
        if (!in_array($status, [self::STATUS_ACTIVE, self:
            trigger_error(sprintf('Le statut %s n\'est pas
        }

        $this->status = $status;
    }

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

Dans cette classe, on a un utilisateur défini par un pseudo (' `username` ') et un `status` ('status').
Ce statut est récupéré avec la méthode ' `getStatus` ' et modifié via ' `setStatus` ', tout en validant valeurs possible.

2. Création admin (sans héritage)

   Imaginons, ont créé une classe 'Admin' qui est égal à un administrateur.

   Le code serait :

```
<?php

declare(strict_types=1);

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

    public function __construct(public string $username, p
    {
    }

    public function setStatus(string $status): void
    {
        if (!in_array($status, [self::STATUS_ACTIVE, self:
            trigger_error(sprintf('Le statut %s n\'est pas
        }

        $this->status = $status;
    }

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

    public function addRole(string $role): void
```

```
    {
        $this->roles[] = $role;
        $this->roles = array_filter($this->roles);
    }

    public function getRoles(): array
    {
        $roles = $this->roles;
        $roles[] = 'ADMIN';

        return $roles;
    }

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

Toujours les propriétés `'username'` et 'status' + méthodes '`getStatus`' et '`setStatus`'. Seule différence est l'ajout de la gestion de rôles ('roles'), avec méthode pour ajouter et récupérer les rôles.

3. Intro à l'Héritage

   Remarquez-vous la redondance entre `User` et `Admin` ? Les propriétés `username` et `status`, ainsi que les méthodes de gestion du statut, sont identiques dans les deux classes. C'est ici que l'héritage entre en jeu.

   L'héritage permet de définir une classe parent, qui contient toutes les propriétés et méthodes communes, et d'étendre cette classe pour créer des sous-classes spécialisées.

4. Refactoring avec l'Héritage

   On peut refactoriser classes `user` et `Admin` en utilisant heritage. Dabord définissons classe `user` :

```php
<?php

declare(strict_types=1);

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

    public function __construct(protected string $username
    {
    }

    public function setStatus(string $status): void
    {
        if (!in_array($status, [self::STATUS_ACTIVE, self:
            trigger_error(sprintf('Le statut %s n\'est pas
        }

        $this->status = $status;
    }

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

Ensuite créons classe `Admin` en tant que sous classe de `user` :

```php
<?php

declare(strict_types=1);

class Admin extends User
{
    private array $roles = [];
```

```
    public function __construct(string $username, array $r
    {
        parent::__construct($username, $status);
        $this->roles = $roles;
    }

    public function addRole(string $role): void
    {
        $this->roles[] = $role;
        $this->roles = array_filter($this->roles);
    }

    public function getRoles(): array
    {
        $roles = $this->roles;
        $roles[] = 'ADMIN';

        return $roles;
    }

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

Explications :

- Classe `user` :
  - Est maintenant la classe parent, contient des propriétés et méthodes communes aux utilisateurs et administrateurs.
  - Propriété, `username` et `status` sont protégés (`protected`),ce qui permet aux sous-classes d'y accéder.
- Classe `Admin` :
  - Classe `Admin` étend `user` , héritant ainsi de ses propriétés et méthodes.

- Elle ajoute également une gestion des rôles spécifique aux admins.
- Constructeur de `Admin` appelle le constructeur de la classe parent avec `parent::__construc()` en passant argument nécessaire.

Conclusion :

Grace a héritage = supprime redondance de code en gardant structure claire et extensible. Si on doit rajouter new classe, juste en rajouter dans `user`.

Fondamental en POO, permet de créer des hiérarchies de classe bien structurer, facilitant maintenance et extension du code.

Étendre une Classe en PHP

1. Intro Héritage

Imaginons un arbre généalogique. Une classe “enfant” hérite des propriétés et méthodes de sa classe “parent”. Cette classe parent peut avoir un parent aussi, créant une chaîne d’héritage.

Contrairement au C++ ou Python, en Php, on peut hériter que d’une classe à la fois. Ça simplifie la gestion, mais limite à une hiérarchie linéaire

2. Mot clé `extends` = hériter d’une classe

quand code est dupliqué, il est bon de la factoriser en héritage. Permet d’éliminer la redondance et réduire les risque d’erreur lors des MAJ (Plus de modif a plusieurs endroits).

Nous avons une classe `Admin` qui partage des caractéristiques avec une classe `User`. On peut étendre `User` pour éviter de dupliquer le code.

Exemple sans héritage :

```
<?php

declare(strict_types=1);

class User
{
```

```
    public const STATUS_ACTIVE = 'active';
    public const STATUS_INACTIVE = 'inactive';

    public function __construct(public string $username, p
    {
    }

    public function setStatus(string $status): void
    {
        if (!in_array($status, [self::STATUS_ACTIVE, self:
            trigger_error(sprintf('Le statut %s n\'est pas
        }

        $this->status = $status;
    }

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

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

    public function __construct(public string $username, p
    {
    }

    public function setStatus(string $status): void
    {
        if (!in_array($status, [self::STATUS_ACTIVE, self:
            trigger_error(sprintf('Le statut %s n\'est pas
        }

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

```
    }

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

    public function addRole(string $role): void
    {
        $this->roles[] = $role;
        $this->roles = array_filter($this->roles);
    }

    public function getRoles(): array
    {
        $roles = $this->roles;
        $roles[] = 'ADMIN';

        return $roles;
    }

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

Exemple avec héritage :

```
<?php

declare(strict_types=1);

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

```
    public function __construct(public string $username, p
    {
    }

    public function setStatus(string $status): void
    {
        if (!in_array($status, [self::STATUS_ACTIVE, self:
            trigger_error(sprintf('Le statut %s n\'est pas
        }

        $this->status = $status;
    }

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

class Admin extends User
{
    public function __construct(public string $username, p
    {
        parent::__construct($username, $status);
    }

    public function addRole(string $role): void
    {
        $this->roles[] = $role;
        $this->roles = array_filter($this->roles);
    }

    public function getRoles(): array
    {
        $roles = $this->roles;
        $roles[] = 'ADMIN';

        return $roles;
```

```
    }

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

Explication du code :

- Classe `User` :
  - contient propriété `username` et `status` + méthode pour définir et obtenir statut de l'utilisateur.
  - `User` = classe parent pour toute classe souhaitant bénéficier des propriétés et méthodes.
- Classe `Admin` :
  - Etendant `User` , `Admin` hérite de toutes ses propriétés et méthodes.
  - `extends` est utilisé pour signifier que `Admin` est sous-classe de `User` .
  - `Admin` ajoute ses propres propriétés et méthodes, spécifique à la gestion des rôles.
- Parent Constructor (`parent::__construct()`):
  - `Admin` est instancier, il utilise constructeur de `User` pour init `username` et `status` .

4. Avantage de l'Héritage
   - **Réduction redondance**, avec classe parent, on n'a pas à dupliquer le code commun. Donc plus facile à maintenir
   - **Simplifie-les MAJ**, Toutes modifs apportées à la classe parent se répercutent automatiquement sur les sous-classes, réduisant risque d'erreurs.
   - **Structure du code**, aide à structurer le code en fonction des relations hiérarchiques, comme le cas de `User` et `Admin`

Conclusion

Héritage, permet de passer propriété et méthodes d'une classe parent à une classe enfant. Aide à structurer le code en évitant duplication. `extends` sert à indiquer cette relation, et les classes enfant peuvent bénéficier de toutes les fonctionnalités de la classe parent.

Accéder aux Propriétés de la Classe Parente en PHP

1. Accès propriété classe parente

Utilisé héritage en PHP = objet d'une classe enfant peuvent accéder aux propriétés et méthode de la classe parente comme si les méthodes et prop faisaient partie de classe enfant.

exemple :

```
<?php

declare(strict_types=1);

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

    public function __construct(public string $username, publ
    {
    }
}

class Admin extends User
{
    // Méthode pour afficher le statut de l'administrateur
    public function printStatus(): void
    {
        // Accès à la propriété 'status' de la classe parente
        echo $this->status;
    }
```

```
}

$admin = new Admin('Lily');
$admin->printStatus(); // Affiche: 'active'
```

La classe `Admin` hérite de la classe `user`. La méthode `printStatus()` de `Admin` accède a `status` Définie dans `User` . `$this->status` fonctionne comme si `status` faisait partie de la classe `Adm in`

2. Accès propriétés Statique de la classe parente.

Propriétés statiques = `static`, appartiennent à la classe plutôt qu'instance de celle-ci. Enfant peuvent accéder aux prop statique des parents et inversement, mais existe limite à comprendre.

Exemple de code :

```
<?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, publ
    {
    }
}

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

    // Méthode pour mettre à jour les propriétés statiques de
    public static function nouvelleInitialisation(): void
    {
```

```
        self::$nombreAdminInitialisés++; // Incrémente le nom
        parent::$nombreUtilisateursInitialisés++; // Incrémen
    }
}

Admin::nouvelleInitialisation();
var_dump(Admin::$nombreAdminInitialisés, Admin::$nombreUtilis
var_dump(User::$nombreAdminInitialisés); // Ceci ne fonctionn
```

- Propriété statique via `self` et `parent` :
  - Méthode `nouvelleInitialisation()` dans `Admin` utilise `self::$nombreAdminInitialisés` pour accéder à une prop statique de la classe courante (`Admin`)
  - `parent::$nombreUtilisateurInitialisés` est utilisé pour accéder à une prop statique de la classe parente (`User`).
- Accès limité des P.Statique :
  - `Admin` peut accéder à la statique de `User` grâce au mot clé `parent`
  - Inversement, `user` accède à la statique d'`Admin`, car héritage fonctionne dans le sens descendant uniquement.

3. Mot clé `parent`

   `parent` = Utilisé dans classe enfant pour faire réf aux méthodes ou prop de classe parente. Permet pas de cibler une classe spécifique dans une chaine d'héritage si plusieurs niveaux s'ont présent, remonte à classe parente immédiate.

   Exemple référence a un parent :

```
<?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, p
    {
    }
}

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

    public static function nouvelleInitialisation(): void
    {
        self::$nombreAdminInitialisés++;
        parent::$nombreUtilisateursInitialisés++;
    }
}

Admin::nouvelleInitialisation();
```

- `self` ; accède aux prop ou méthode statique de classe courante.
- `parent` : Accède aux prop ou méthode de la classe parente.

Conclusion :

Héritage permet classes enfants accéder aux prop et méthodes définies dans leurs classe parentes, simplifiant le code. `self` et `parent` permet de gérer les prop et méthodes statique entre classe parents et enfant. Héritage fonctionne de manière DESCENDANTE, et non l'inverse.

Notes sur l'Accès aux Méthodes de la Classe Parente en PHP

Accès direct aux méthodes de la classe parente :

- Héritage des méthodes : classe enfant peut accéder directement aux méthodes de la classe parente de la même manière qu'elle accède aux siennes, en utilisant `->` .

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

class Admin extends User
{
    public function printCustomStatus()
    {
        echo "L’administrateur {$this->username} a pour
        $this->printStatus(); // Appel de la méthode pa
    }
}

$admin = new Admin('Lily');
$admin->printCustomStatus(); // Affiche: “L’administrat
$admin->printStatus(); // Appel de la méthode `printSta
```

- Point clé : Si méthode pas définie dans une classe enfant, PHP va automatiquement utiliser la méthode définie dans la classe parente.

Surcharge de méthode (Overide)

- surcharge = permet réécrire une méthode d'une classe parente à une enfant.
- règles :
  - **Signature compatible** : signature de la méthode surchargée = compatible a celle de la méthode parente.
  - **Argument** :
    - Impossible d'enlever des arguments
    - Ajout d'arguments possible uniquement s'ils sont optionnels
  - Types :
    - Changement de type d'un argument possible que si compatible avec le type d'origine.
    - Changement de type de retour possible que s'il reste compatible avec le type d'origine.

Exploitation Méthode `parent::`

- **Choix lors de la surcharge :** Lors de la réécriture d'une méthode, on peut :
  - Réécrire la méthode
  - Appeler la méthode parente avec `parent::` et y ajouter des comportements spécifique.

Exemple :

```
<?php

declare(strict_types=1);

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

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

```
    }

    public function setStatus(string $status): void
    {
        if (!in_array($status, [self::STATUS_ACTIVE, self:
            trigger_error(sprintf('Le statut %s n\'est pas
        }

        $this->status = $status;
    }

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

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

    // Réécriture complète de la méthode avec une signatur
    public function setStatus(string $status): void
    {
        if (!in_array($status, [self::STATUS_ACTIVE, self:
            trigger_error(sprintf('Le statut %s n\'est pas
        }

        $this->status = $status;
    }

    // Appel de la méthode parente, puis ajout de comporte
    public function getStatus(): string
    {
        return strtoupper(parent::getStatus());
    }
}
```

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

Point clés :

- Utilisation de `parent::` permet appeler la méthode de la classe depuis la classe enfant.
- Contrôle flexible : le dev choisi de réécrire complètement une méthode ou de l'étendre en appelant la méthode parente.

Résumé

- Les méthodes d'une classe parente sont accessibles depuis une classe enfant via l'héritage, utilisant la flèche `->`.
- La surcharge permet de modifier ou d'étendre le comportement d'une méthode parente.
- `parent::` est utilisé pour appeler une méthode de la classe parente depuis une méthode surchargée dans la classe enfant.

Notes sur l'Accès Restreint aux Propriétés et Méthodes en PHP

Notes sur l'accès restreint Propriétés et Méthodes en PHP

Visibilité : Public, Privé et Protéger l'Héritage

- Rappel sur les visibilités :
  - Public : Accessible de n'importe où.
  - Privée : Accessible que dans la classe ou la propriété ou la méthode est définie.
- Problème avec les éléments privés :
  - Lorsqu'une méthode ou une propriété est définie comme privée dans une classe parent, elle n'est pas accessible depuis une classe enfant.
  - Effet : L'héritage est interrompu pour ces éléments privés.

Exemple :

- Définition de `User` avec des élément privée :

```
<?php

declare(strict_types=1);

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

    public function __construct(private string $username,
    {
    }

    private function setStatus(string $status): void
    {
        assert(
            in_array($status, [self::STATUS_ACTIVE, self::
            sprintf('Le statut %s n\'est pas valide. Les s
        );
        $this->status = $status;
    }

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

- Tentative d’héritage dans `Admin`

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

    // Réécriture de la méthode avec la même signature
```

```
    public function setStatus(string $status): void
    {
        assert(
            in_array($status, [self::STATUS_ACTIVE, self::
            sprintf('Le statut %s n\'est pas valide. Les s
        );
        $this->status = $status;
    }

    // Tentative d'appel à la méthode parente (échoue)
    public function getStatus(): string
    {
        return strtoupper(parent::getStatus());
    }
}

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

On Observe :

- Erreur :
  - Le code ne fonctionne pas, car les méthodes `setStatus` et `getStatus` sont privées dans la classe `User`.
  - PHP renvoie une erreur indiquant que la méthode ou la propriété à laquelle on essaie d'accéder est privée ou n'existe pas.
- Conclusion :
  - Les éléments définis comme privés sont inaccessibles à la classe enfant.
  - Impact : Ces éléments ne sont pas hérités, donc la classe enfant ne peut pas les surcharger ou les utiliser.

Solution : Propriété et Méthode protégée

- `protected`

    - permet de définir des éléments qui sont accessibles à la classe ou ils sont définis et aux classes enfants
- Utilisation :
    - Permet contrôler accès tout en autorisant l'héritage.
    - L'héritage n'est pas interrompu, contrairement à l'usage de `private`
- À retenir :
    - `private` : limite l'accès strictement à la classe actuelle.
    - `protected` : Limite l'accès actuelle et à ses classe d'enfants, permettant un héritage sécurisé.

Notes sur la Visibilité des Propriétés et Méthodes en PHP

Différentes Visibilités : Public, Privé et Protégé

- Public (`public`) :
    - Accès ouvert à tous, propriété ou méthode accessible depuis n'importe quelle classe ou objet.
- Privé (`private`) :
    - Accès restreint, Seul classe ou pro et méthode définis peut accéder.
    - Héritage Impossible, Classe enfant ne peuvent pas accéder aux prop ou méthode privée des parents
- Protégé (`protected`) :
    - Accès limité, Prop ou méthode est accessible que depuis la classe ou elle est définie et ses classe enfants.
    - Permet Héritage, classe enfants peuvent utiliser et modifier les prop ou méthode protégés du parent.

Pourquoi Utiliser `protected` :

- Cas d'usage :
    - Idéal pouir les méthodes ou prop qui doivent être utilisées seulement à l'interieur d'une classe et ses lcasse dérivées.

- Permet d'encapsuler des comportements tout en les rendants dispo pour héritage.

Exemple :

- Définition de `User` avec `protected` :

```
<?php

declare(strict_types=1);

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

    public function __construct(protected string $username
    {
    }

    protected function setStatus(string $status): void
    {
        assert(
            in_array($status, [self::STATUS_ACTIVE, self::
            sprintf('Le status %s n\'est pas valide. Les s
        );
        $this->status = $status;
    }

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

- Héritage dans `Admin` :

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

    // Réécriture de la méthode avec la même signature
    public function setStatus(string $status): void
    {
        assert(
            in_array($status, [self::STATUS_ACTIVE, self::
            sprintf('Le status %s n\'est pas valide. Les s
        );
        $this->status = $status;
    }

    // Utilisation de la méthode parente protégée
    public function getStatus(): string
    {
        return strtoupper(parent::getStatus());
    }
}

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

Observation :

- Succès :
  - code fonctionne car méthode `setStatus` et `getStatus` de `User` sont `protected`
  - Héritage et accessibilité, Méthode protégées sotn accessible et peuvent être utilisées ou surchargées par la classe enfant `Admin`

Résumé

- Visibilité :
  - `private` : Limite l'accès strictement a la classe.

    - `protected` : permet l’accès à la classe et a ses enfants.
    - `public` : Ouvert a tous
- Bonne pratique :
    - Utilisez `public` pour les prop ou méthode qui doivent être accessible a tous.
    - Utilisez `private` Pour qui ne doivent être accessible dans la classe actuel.
    - Utilisez `protected` Pour celle qui doivent être accessible à la classe actuelle et à ses classe enfants.
- **Accès des enfants** :
    - Pour qu'une propriété soit modifiable depuis une classe enfant, elle doit être `public` ou `protected` ou posséder un mutateur public.

Notes sur l'Usage des Classes Abstraites et Finales en PHP

Classe Abstraite : Concept et Usage

- Définition :
    - Une classe abstraite ne peut pas être instanciée directement.
    - Elle sert de modèle pour les classe heritent, en définissant des pros et méthodes de base.
- Usage :
    - `abstract` pour délcarer classe abstraite.
    - Elel force héritage, les classes enfants doivent implémenter les méthodes abstraites définies dans la classe parentes.
- Avantages :
    - structure commune : permet de créer une base solide et commune pour plusieurs classes
    - Extensibilité : Facilite l’evolutions futures du code sans duplicaztion.

Exemple :

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

    public function __construct(public string $email, publ
    {
    }

    public function setStatus(string $status): void
    {
        assert(
            in_array($status, [self::STATUS_ACTIVE, self::
            sprintf('Le status %s n\'est pas valide. Les s
        );
        $this->status = $status;
    }

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

    abstract public function getUsername(): string; // Mét
}
```

Remarque : Impossible d'instancier `user` directment entrainerait une erreur.

Méthode Abstraite : Déclaration et Implémentation.

- Définition :
  - Méthode abstraite a pas de corp (pas de code)
  - déclaré `abstract` et doit être implémentée dans chaque classe enfant.
- Usage

- Forcer la redéfinition : les classes qui héritent de la classe abstraite doivent implémenter ces méhodes
- Signature seulement : On définit que la signature de la méthode.

Exemple :

```
abstract public function getUsername(): string;
```

- Dans la classe enfant :

```
class Admin extends User
{
    public function getUsername(): string
    {
        return $this->email;
    }
}

class Player extends User
{
    public function __construct(string $email, public stri
    {
        parent::__construct($email, $status);
    }

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

Classe final : empêcher l’héritage

- Définition :
  - ne peut être étendue. Aucune autre classe peut hériter d’elle.

- Usage
  - `final` pour empêcher héritage de certainer classes ou méthodes .

Exemple :

```
final class Admin extends User
{
    public function getUsername(): string
    {
        return $this->email;
    }
}

// Impossible d'étendre Admin :
// class SuperAdmin extends Admin {} -> Erreur !
```

- Finalité :
  - Sécuriser le code : sécurise des classes ou méthodes pour qu'elles ne soient pas modifiées par des classes dérivées.
  - Stabilité : garantir qu'une classe ou une méthode ne subira pas d'altération future par héritage.

Résumé et bonne pratique :

- Abstraction :
  - Utilisez calsse et méthode abstraite pour créer des structures de base extensible et imposer des contraintes aux classes enfants.
  - Méthode Abstraite : Déclarez dans la classe parente, implémentez dans les classe enfants
- Finalisation :
  - `final` pour protéger des classe et méthodes contre héritage ou modif de structure.
  - Sécurisation : Assurez vous que certaines partie du code soit immuable et non extensible si nécesaire.

Gestion des Constantes et Propriétés Statiques dans les Classes PHP

Compréhension des Constantes et Propriétés Statiques

- **Constantes** :
  - Valeurs immuables déclarées dans une classe.
  - Elles sont accessibles via `self::CONSTANTE` dans la classe où elles sont définies.
  - **Règle** : Par défaut, les constantes ne changent pas et ne sont pas censées être modifiées dans les classes enfants.
- **Propriétés Statiques** :
  - Elles appartiennent à la classe elle-même plutôt qu'à une instance spécifique.
  - Elles sont accessibles avec `self::$property` ou `static::$property` selon le contexte.

Changer la Valeur d'une Constante dans une Classe Enfant

- **Concept** :
  - Bien que les constantes ne soient pas censées être modifiées, une classe enfant peut redéfinir une constante du même nom pour ajuster son comportement.
- **Problème Potentiel** :
  - Si une méthode dans une classe parente utilise `self::CONSTANTE`, elle se réfère toujours à la constante de la classe parente, même si une classe enfant a redéfini la constante.
  - Cela peut entraîner des incohérences, notamment si les classes enfants ont des besoins spécifiques.
- **Exemple** :

```
phpCopier le code
<?php

declare(strict_types=1);

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

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

    public function setStatus(string $status): voi
d
    {
        assert(
            in_array($status, [self::STATUS_ACTIV
E, self::STATUS_INACTIVE]),
            sprintf('Le status %s n\'est pas valid
e. Les status possibles sont : %s', $status, implo
de(', ', [self::STATUS_ACTIVE, self::STATUS_INACTI
VE]))
        );
        $this->status = $status;
    }

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

    abstract public function getUsername(): strin
g;
}
```

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

    public function __construct(string $email, str
ing $status = self::STATUS_ACTIVE, public array $r
oles = [])
    {
        parent::__construct($email, $status);
    }

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

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

- **Résultat** :
  - L'assignation du nouveau statut `Admin::STATUS_INACTIVE` échoue car la méthode `setStatus` dans `User` utilise `self::STATUS_INACTIVE`, qui fait référence aux constantes de la classe `User` plutôt qu'à celles de `Admin`.

Late Static Binding (Résolution Statique à la Volée)

- **Solution** :
  - Utiliser `static::CONSTANTE` au lieu de `self::CONSTANTE`.
  - Cela permet de faire référence à la constante la plus proche dans la hiérarchie, c'est-à-dire celle définie dans la classe enfant si elle existe.

- **Exemple Modifié** :

```
phpCopier le code
<?php

declare(strict_types=1);

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

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

    public function setStatus(string $status): voi
d
    {
        assert(
            in_array($status, [static::STATUS_ACTI
VE, static::STATUS_INACTIVE]), // Utilisation de s
tatic au lieu de self
            sprintf('Le status %s n\'est pas valid
e. Les status possibles sont : %s', $status, implo
de(', ', [static::STATUS_ACTIVE, static::STATUS_IN
ACTIVE]))
        );
        $this->status = $status;
    }

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

    abstract public function getUsername(): strin
```

```
g;
}

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

    public function __construct(string $email, str
ing $status = self::STATUS_ACTIVE, public array $r
oles = [])
    {
        parent::__construct($email, $status);
    }

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

$admin = new Admin('michel@petrucciani.com');
var_dump($admin);
$admin->setStatus(Admin::STATUS_INACTIVE); // Main
tenant, cela fonctionne
```

- **Explication** :
  - `static::CONSTANTE` permet d'utiliser la constante définie dans la classe `Admin` plutôt que celle de `User`.
  - Ce mécanisme est appelé **Late Static Binding** (résolution statique tardive) car la constante utilisée est déterminée au moment de l'exécution et non lors de l'analyse du code.