

La gestion des erreurs

Table des matières

I. Contexte	3
II. Principe de la gestion des erreurs	3
III. Exercice : Appliquez la notion	4
IV. Gérer proprement les erreurs	5
V. Exercice : Appliquez la notion	7
VI. Lancer une exception	7
VII. Exercice : Appliquez la notion	10
VIII. Attraper une exception	10
IX. Exercice : Appliquez la notion	12
X. Throwable, Exception et Error	13
XI. Exercice : Appliquez la notion	15
XII. Créer des exceptions	16
XIII. Exercice : Appliquez la notion	18
XIV. Auto-évaluation	19
A. Exercice final	19
B. Exercice : Défi	22
Solutions des exercices	25

I. Contexte

Durée : 1 h

Environnement de travail : PHPSTORM

Pré-requis : Bases de PHP, programmation orientée objet (classe, interface, héritage)

Contexte

L'exécution d'un code peut rencontrer des anomalies.

Celles-ci peuvent provenir d'erreurs techniques introduites par les développeurs, un mauvais nom de variable ou de fonction par exemple. Elles peuvent également parfois provenir d'erreurs générées par les utilisateurs.

La problématique lorsqu'une erreur est rencontrée est que le code va s'arrêter brutalement, sans solutions alternatives.

Nous verrons que le concept de gestion des erreurs permettra d'y répondre.

Une erreur "attrapée" donnera lieu à l'exécution d'un code : nous pourrons donc informer et aider l'utilisateur vers le fonctionnement correct de l'application ou du site web.

II. Principe de la gestion des erreurs

Objectif

- Comprendre d'où proviennent les erreurs

Mise en situation

Développer une application complexe avec de nombreuses fonctionnalités et beaucoup de lignes de code et de couches d'abstraction peut s'avérer source d'erreurs pour les développeurs, mais également pour les utilisateurs.

Nous allons voir ici ce que l'on appelle une « erreur » et comment elles se manifestent.

Définition Que considère-t-on comme une erreur ?

Une erreur est une exécution anormale d'un code qui induit une impossibilité partielle ou totale d'utilisation d'une application ou d'un site web.

Selon la configuration du serveur, les erreurs sont affichées sur la page à partir de laquelle elles ont été générées, souvent sous la forme d'un texte brut.

Si elles ne sont pas affichées, le résultat de l'exécution du script sera une page blanche.

Complément L'influence de la configuration sur l'affichage des erreurs

Les serveurs peuvent être configurés pour n'afficher que certains types d'erreurs.

Par exemple, concernant les erreurs internes de PHP, il existe plusieurs niveaux d'erreurs selon leur gravité :

- **Notice** : erreur n'entraînant pas l'arrêt du script. Toutes les instructions s'exécutent.
- **Warning** : une instruction n'a pas pu être exécutée, mais n'entraîne pas un arrêt du script.
- **Fatal error** : erreur critique entraînant un arrêt du script.

Les différents types d'erreurs peuvent être retrouvés sur cette page¹.

Attention Les erreurs sur un serveur de production

En mode production, c'est-à-dire en utilisation réelle, ces erreurs ne doivent pas être affichées à l'utilisateur. Celles-ci pourraient permettre à un utilisateur mal intentionné de découvrir et d'exploiter des informations sensibles liées à l'application.

Exemple

Le code ci-dessous contient deux erreurs.

La variable `$userLogon` n'existe pas et la fonction `logUser ($userLogin)` est appelée sans son paramètre.

```
1 <?php
2 $userLogin = 'JohnDoe';
3
4 echo $userLogon;
5
6 logUser();
7
8 function logUser($userLogin) {
9     echo "$userLogin est maintenant connecté";
10 }
11 ?>
```

Voici ce qu'il résultera de son exécution :

```
1 // echo $userLogon affichera :
2 PHP Notice: Undefined variable: userLogon in /home/runner/erreurs/index.php on line 5
3
4 // logUser(); affichera :
5 PHP Fatal error: Uncaught ArgumentCountError: Too few arguments to function logUser(),
6 0 passed in index.php on line 7 and exactly 1 expected in index.php:9
```

Syntaxe À retenir

- Les erreurs sont réparties par ordre de gravité : *notice*, *warning*, *fatal*.
- Elles donnent lieu à des messages indiquant la raison : le nom de la variable ou fonction qui pose problème, ainsi que la ligne où se produit l'erreur dans le code.

III. Exercice : Appliquez la notion

Question

[solution n°1 p.27]

Exécutez le script suivant et identifiez quelles erreurs seront déclenchées.

```
1 <?php
2
3 function greetings($firstName, $lastName)
4 {
5     echo sprintf('Bonjour %s %s', $firstName, $lastName);
6 }
7
8 $name = 'Ulrich';
```

¹ <https://www.php.net/manual/fr/errorfunc.constants.php>

```
9 $age = 25;  
10 $defaultRole = DEFAULT_ROLE;  
11  
12 echo $city;  
13  
14 greetings($name);  
15
```

IV. Gérer proprement les erreurs

Objectifs

- Gérer les erreurs
- Enregistrer les erreurs

Mise en situation

Bien qu'une erreur soit survenue, un script doit toujours trouver une solution pour la résoudre, ou un chemin pour continuer son exécution. Nous allons voir pourquoi il faut bien gérer les cas d'erreurs potentielles.

Dans la réalité, les erreurs internes de PHP (*notice*, *warning*, *fatal*) doivent être évitées, ou au moins maîtrisées.

Il peut parfois s'agir d'erreurs "bêtes" dues à une inattention, une absence de relecture du code ou des tests insuffisants pendant le développement. Elles peuvent toutefois se produire.

D'autres erreurs plus courantes à gérer sont des erreurs provoquées par l'imbrication de fonctionnalités développées.

Prenons l'exemple d'une application dédiée à la comptabilité : simplifions, l'utilisateur saisit des chiffres et déclenche des calculs complexes afin d'établir des bilans. Certaines règles de calcul enchaînées entre elles peuvent donner lieu à des erreurs. Par exemple, une division par zéro, ou l'obtention d'un résultat négatif. Dans ce cas, il ne s'agit pas d'une erreur PHP, mais d'une erreur directement liée à la saisie de l'utilisateur, c'est-à-dire à l'aspect métier.

Méthode Cacher les erreurs internes

La première bonne pratique est de faire en sorte que les erreurs internes de PHP ne soient pas affichées au grand public. Elles sont très techniques, verbeuses et donc peu explicites pour l'utilisateur.

Exemple

Pour cela, au début d'un script PHP utilisez l'instruction :

```
1 <?php  
2  
3 ini_set('display_errors', 'off');
```

Dans le fichier de configuration du serveur : **php.ini**.

```
1 display_errors = Off;  
2 ou  
3 environnement=production;
```

Grâce à cette configuration, les erreurs sont automatiquement affichées dans un environnement de développement et cachées en production pour l'utilisateur.

Méthode Logguer les erreurs internes

Dans un second temps, il faut enregistrer ces erreurs dans des fichiers de logs stockés sur le serveur. Cela permettra d'en garder un historique, qui sera utile pour enquêter à posteriori sur un dysfonctionnement, analyser les erreurs rencontrées et optimiser le script.

Exemple

Dans la configuration du serveur (**php.ini**), décommentez ou ajoutez la ligne :

```
1 error_log = "/var/log/php_errors.log // Le fichier peut être renommé.
```

Conseil Archiver ses erreurs

Jour après jour, les utilisateurs vont générer des erreurs, mais ne remonteront pas forcément le bug en temps réel.

Il est donc judicieux de créer et d'archiver des fichiers de logs journaliers afin de faciliter les futures recherches.

L'archivage par jour est le plus répandu car il permet de retrouver plus facilement une erreur parmi un volume de logs réduit.

Les erreurs de logique métier

Nous entrons ici dans le vif du sujet de la gestion des erreurs. Celles-ci ne sont pas forcément générées automatiquement par PHP.

Parfois, on associe l'erreur à une contrainte pour le développeur et il est vrai qu'au sens informatique, elle empêche l'exécution correcte du script. Pourtant, les erreurs de logique métier sont nécessaires, elles font office de protection qui empêche l'utilisateur de réaliser des opérations non autorisées par les contraintes métiers. Après analyse, la remontée de ces causes d'erreurs jusqu'aux utilisateurs peut même aider à améliorer des pratiques sur le terrain.

Le développeur doit donc avoir une connaissance métier pour pouvoir les mettre en œuvre. Les erreurs de logique métier sont, en d'autres termes, des erreurs programmées.

Exemple La division par zéro

Admettons qu'une division soit effectuée au moyen de paramètres reçus dans une URL.

```
1 <?php
2
3 // Si nombre1 vaut 4 et nombre2 vaut 2, tout va bien, mais qu'en est-il si nombre2 vaut zéro ?
  ou si ces nombres ne sont pas définis ?
4
5 $division = $_GET['nombre1']/$_GET['nombre2'];
6
7 echo $division; // Affichera une valeur indéterminée (INF, UNDEFINED, NAN, etc).
8 // Génère un warning : PHP Warning: Division by zero in index.php on line 3
9
```

Fondamental Une gestion des erreurs nécessaires

La problématique de l'exemple ci-dessus est que la génération du *warning* n'interrompt pas l'exécution du reste du script.

De plus, l'utilisateur n'a pas l'information qu'un mauvais calcul s'est produit, car ces erreurs sont normalement cachées.

L'objectif est donc de mettre en place un système de gestion des erreurs de logique métier.

PHP possède un gestionnaire d'erreurs permettant de lancer une erreur et de l'attraper au moment voulu dans le script.

DivisionByZeroError

Précision : seul un PHP *warning* est généré en premier pour une division par zéro.

L'exemple suivant utilisera un modulo, un autre opérateur de division permettant d'obtenir le reste de la division. Celui-ci déclenche quant à lui une erreur fatale lors d'une division par zéro.

```
1 <?php
2
3 try {
4     $result = 5 % 0;
5 } catch (DivisionByZeroError $e) {
6     echo $e->getMessage();
7 }
8
1 echo $e->getMessage(); // Affichera : Modulo by Zero
```

Remarque La syntaxe de la gestion d'erreurs

Nous verrons la syntaxe de la gestion d'erreurs un peu plus tard.

Retenons simplement que nous venons d'attraper (mot clé `catch`, traduit par « attraper ») notre première erreur. **DivisionByZero** est une classe fournie par PHP.

Si nous n'avons pas attrapé cette erreur, l'utilisateur aurait, au mieux, vu une page blanche (si les erreurs sont masquées), au pire un message d'erreur verbeux et pas très explicite pour un non-développeur.

Grâce à cette méthode, il est possible d'afficher un message à l'utilisateur pour le prévenir de l'erreur de façon compréhensible et maîtrisée par le développeur.

Syntaxe À retenir

Pourquoi gérer proprement les erreurs ?

- Pour garder un historique dans le cas où une erreur ne remonterait pas immédiatement,
- Pour aider les développeurs à tester et déboguer un script,
- Pour avertir l'utilisateur d'une potentielle erreur de manipulation de façon maîtrisée.

V. Exercice : Appliquez la notion

Question

[solution n°2 p.27]

À ce stade, vous ne savez pas encore gérer des exceptions. Pour autant, vous devez être en mesure de sécuriser l'exécution d'un code. Masquez l'affichage des erreurs dans ce script et sécurisez l'exécution de ce code.

```
1 <?php
2
3 $division = $_GET['nombre1']/$_GET['nombre2'];
4
5 echo $division;
```

VI. Lancer une exception

Objectif

- Savoir lancer une exception

Mise en situation

Pour les erreurs de type logique métier, nous parlerons de gestion des exceptions. Nous allons voir, dans un premier temps, comment lancer ou lever une exception dans le code.

Fondamental

`Exception` est la classe de base pour toutes les exceptions utilisateur en PHP 7.

Cette classe, ou toute **autre classe héritant d'`Exception`**, devra toujours être utilisée lorsque nous lancerons une exception.

Syntaxe Déclencher une exception

Une exception est lancée¹ grâce au mot-clé **`throw`**, suivi de l'objet lancé : `throw new Exception()`.

Un message personnalisé peut être indiqué en paramètre : `throw new Exception('Attention erreur')`.

Une exception devra être lancée partout où une erreur potentielle pourrait survenir dans le code.

Exemple

La logique métier souhaite que le résultat du chiffre d'affaires moins les charges ne puisse pas être négatif. De même, les charges ou les ventes ne peuvent être inférieures à 0. Comme il ne s'agit pas d'une erreur interne à PHP, nous lançons nous-mêmes une exception :

```
1 <?php
2
3 function getProfit($sales, $charges) {
4     if ($sales < $charges) {
5         throw new Exception('Attention : le résultat est négatif');
6     }
7     if ($sales<0 || $charges<0) {
8         throw new Exception('Attention : les ventes ou les charges ne peuvent être négatives');
9     }
10    return $sales - $charges;
11 }
12
```

Si un des cas se produit, le texte *Attention : le résultat est négatif* ou *Attention : les ventes ou les charges ne peuvent être négatives* sera affiché à l'endroit où sera gérée l'exception.

Remarque

Le fait de déclencher une exception va interrompre l'exécution de la fonction dans laquelle elle se situe. Dans l'exemple ci-dessus, l'exécution de la fonction `getProfit` s'interrompra au moment où l'exception est levée. La commande `return` ne sera pas exécutée.

¹ <https://www.php.net/manual/fr/language.exceptions.php>

Complément Remontée des exceptions

Dans des scripts complexes utilisant plusieurs niveaux d'objets et de fonctions, les exceptions lancées remontent la pile d'exécution des fonctions.

Exemple

```
1 <?php
2
3 function firstFunction($sales, $charges) {
4     return secondFunction($sales, $charges);
5 }
6
7 function secondFunction($sales, $charges) {
8     return getProfit($sales, $charges);
9 }
10
11 function getProfit($sales, $charges) {
12     if ($sales < $charges) {
13         throw new Exception('Attention le résultat est négatif');
14     }
15
16     return $sales - $charges;
17 }
18
19 echo '<pre>';
20
21 echo firstFunction(10, 15);
22
```

Ici, l'exception est lancée en fin de pile dans la fonction `getProfit`, l'appel à `firstFunction` la fera remonter jusqu'à ce qu'elle soit interceptée. Si elle ne l'est pas, une erreur fatale sera levée.

```
1 Fatal error:  Uncaught Exception: Attention le résultat est négatif in index.php:13
2
3 Stack trace:
4 #0 index.php(8): getProfit(10, 15)
5 #1 index.php(4): secondFunction(10, 15)
6 #2 index.php(21): firstFunction(10, 15)
7 #3 {main}
8   thrown in index.php on line 13
9
```

Syntaxe À retenir

- Une exception est lancée avec le mot-clé **throw** suivi d'un objet **Exception** ou d'une sous-classe de cet objet.
- Celle-ci remonte la pile d'exécution jusqu'à être attrapée. Dans le cas contraire, une erreur fatale sera levée.

Complément

Throw¹

Exception²

1 <https://www.php.net/manual/fr/language.exceptions.php>

2 <https://www.php.net/manual/fr/class.exception.php>

VII. Exercice : Appliquez la notion

Question

[solution n°3 p.27]

Modifiez le code suivant afin de déclencher des exceptions plutôt que d'afficher de simples messages d'erreur.

```
1 <?php
2
3 // On vérifie qu'on a bien deux variables de définies
4 if (isset($_GET['nombre1']) && isset($_GET['nombre2'])) {
5     // On vérifie qu'il s'agit de numériques
6     if (is_numeric($_GET['nombre1']) && is_numeric($_GET['nombre2'])) {
7         // On vérifie qu'on n'essaie pas de faire une division par zéro
8         if ((int)$_GET['nombre2'] !== 0) {
9             $division = $_GET['nombre1'] / $_GET['nombre2'];
10
11             echo $division;
12         } else {
13             echo 'Vous ne pouvez pas diviser un nombre par zéro';
14         }
15     } else {
16         echo 'Vous devez fournir des chiffres';
17     }
18 } else {
19     echo 'Vous devez définir deux nombres';
20 }
21
22
23
```

VIII. Attraper une exception

Objectif

- Savoir attraper une exception

Mise en situation

Une fois l'exception lancée, il faut indiquer à PHP de l'attraper au moment voulu. Généralement, les exceptions sont lancées à l'intérieur d'une fonction. Elles devront alors être attrapées lors de l'appel à cette fonction.

Syntaxe

Pour attraper une exception, il faut entourer le code par un bloc try/catch :

```
1 try {
2     // Code à tester, où est déclenchée l'exception.
3 } catch (Exception $e) {
4     // Code à exécuter si l'exception est attrapée.
5 }
```

Un bloc **finally** peut être ajouté après le bloc **catch**. Celui-ci permettra d'exécuter du code, qu'importe qu'une exception ait ou non été attrapée.

```
1 try {  
2  
3 } catch (Exception $e) {  
4  
5 } finally {  
6 // Code à exécuter après try et catch.  
7 }
```

L'ordre d'exécution sera donc le suivant : d'abord le code du bloc `try`, puis celui du bloc `catch`, puis celui du bloc `finally` et enfin le reste du code.

Attention

L'un ne va pas sans l'autre : une exception lancée doit toujours être attrapée. Le cas contraire donnera lieu à une erreur PHP *fatal error*. Si ce cas se produit, le bloc `finally` sera quand même exécuté avant la levée de l'erreur fatale, ce qui peut permettre d'effectuer des opérations avant l'interruption du script.

Exemple

```
1 <?php  
2  
3 try {  
4     echo 'Exécution du bloc try' . PHP_EOL;  
5     getProfit(180, 200);  
6 } catch (Exception $e) {  
7     echo $e->getMessage() . PHP_EOL;  
8 } finally {  
9     echo 'Exécution du bloc finally' . PHP_EOL;  
10 }  
11  
12 echo 'Exécution du reste du code' . PHP_EOL;  
13  
14 function getProfit($sales, $charges)  
15 {  
16     if ($sales < $charges) {  
17         throw new Exception('Attention le résultat est négatif<br>');  
18     }  
19  
20     return $sales - $charges;  
21 }  
22
```

L'exécution de ce code affichera dans l'ordre :

Exécution du bloc try

Attention le résultat est négatif

Exécution du bloc finally

Exécution du reste du code

Complément

Plusieurs exceptions peuvent être attrapées dans l'instruction `catch` en les séparant par le caractère `|`.

Ce sera l'une ou l'autre qui sera attrapée, et non les deux simultanément. Car il est impossible qu'il y ait deux exceptions lancées en même temps, le lancement de la première interrompant l'exécution du reste du script.

```
1 catch(DivisionByZeroError | ParseError $e) {}
```

Syntaxe À retenir

- Une exception est attrapée si elle est levée dans le code exécuté au sein du bloc `try`.
- Le bloc `catch` permet de traiter cette exception.
- Le bloc `finally` permet quant à lui d'exécuter du code systématiquement.

Complément

`try`, `catch`, `finally`¹

IX. Exercice : Appliquez la notion

Le code suivant doit permettre de supprimer un objet de type `User`.

On considère qu'il n'est pas possible de supprimer un utilisateur s'il est administrateur, auquel cas une exception est levée.

```
1 <?php
2
3 function removeUser(User $user)
4 {
5     if ($user->isAdmin()) {
6         throw new Exception('Vous ne pouvez pas supprimer un administrateur');
7     }
8
9     // code métier qui supprime un utilisateur
10    return true;
11 }
12
13 $user1 = new User('Anthony', [User::ROLE_ADMIN]);
14 $user2 = new User('Camille', [User::ROLE_USER]);
15
16 $usersToRemove = [$user1, $user2];
17
18 foreach ($usersToRemove as $user) {
19     removeUser($user);
20 }
21
22 class User
23 {
24     public $name;
25
26     public $roles = [];
27
28     public const ROLE_ADMIN = 'ROLE_ADMIN';
29     public const ROLE_USER = 'ROLE_USER';
30
31     public function __construct(string $name, array $roles)
32     {
33         $this->name = $name;
34         $this->roles = $roles;
```

¹ <https://www.php.net/manual/fr/language.exceptions.php>

```

35     }
36
37     public function isAdmin()
38     {
39         return in_array(self::ROLE_ADMIN, $this->roles);
40     }
41 }
42

```

Question

[solution n°4 p.28]

Mettez en place la syntaxe permettant d'attraper l'exception levée lorsque la fonction `removeUser` est appelée. Que l'utilisateur ait pu ou non être supprimé, vous afficherez le message "Utilisateur traité".

X. Throwable, Exception et Error

Objectifs

- Connaître l'interface Throwable
- Connaître l'objet Exception
- Connaître l'objet Error

Mise en situation

Que se passe-t-il techniquement parlant, quand une exception est lancée ?

PHP fournit une ligne directrice avec une interface dédiée et un panel d'objets prédéfinis nous permettant de gérer bon nombre des exceptions possibles. Nous allons voir ce que contiennent ces objets.

Méthode L'interface Throwable

Cette interface est le point de départ du gestionnaire d'exceptions en PHP.

Elle fournit les principaux éléments des classes qui vont l'implémenter. Parmi eux :

- `getMessage()` : permet d'afficher le message associé à l'exception lancée
- `getCode()` : permet d'obtenir le code de l'exception
- `getFile()` : récupère le fichier où l'exception est lancée
- `getLine()` : récupère le numéro de la ligne où l'exception est lancée
- `getTrace()` : récupère l'enchaînement des appels (chemin/fichier.php et nom de la fonction) ; très utile selon le niveau de profondeur du code

Fondamental L'objet Exception

Exception est la classe de base pour toutes les exceptions de type logique métier. Elle implémente l'interface Throwable et définit donc les méthodes vues précédemment.

PHP prédéfinit des classes qui étendent Exception, comme :

- `InvalidArgumentException` (étend `LogicException` et donc `Exception`) : exception lancée si le type d'un argument transmis à une fonction n'est pas le bon.
- `OutOfBoundsException` (étend `RuntimeException` et donc `Exception`) : exception lancée à l'exécution quand l'index d'un tableau n'existe pas.

Exemple

```
1 <?php
2
3 try {
4     displayUser(20);
5 } catch (InvalidArgumentException $e) {
6     echo $e->getMessage();
7 }
8
9 function displayUser($user) {
10     if (!is_string($user)) {
11         throw new InvalidArgumentException('Le paramètre doit être une chaîne de caractères');
12     }
13
14     echo "Vous êtes $user";
15 }

1 // Affichera : Le paramètre doit être une chaîne de caractères.
```

Ici, la fonction `displayUser` attend une chaîne de caractères, or c'est un entier qui est passé en paramètre. L'exception `InvalidArgumentException` est donc attrapée.

Fondamental L'objet Error

`Error` est la classe permettant de traduire les erreurs internes de PHP (*notice*, *warning*, *fatal*) en exceptions. Elle implémente également l'interface `Throwable`. PHP prédéfinit des classes qui étendent `Error`, comme :

- `DivisionByZeroError` (étend `ArithmeticError` et donc `Error`) : exception lancée lors d'une tentative de division par zéro.
- `ArgumentCountError` (étend `TypeError` et donc `Error`) : exception lancée quand il manque des arguments en paramètres d'une fonction.

Exemple

```
1 <?php
2 try {
3     displayUser('Doe');
4 } catch (ArgumentCountError $e) {
5     echo $e->getMessage();
6 }
7
8 function displayUser(string $lastname, string $firstname) {
9     echo "Vous êtes Doe";
10 }
11 ?>

1 //Affichera : Too few arguments to function displayUser(), 1 passed in
  /home/runner/erreurs/index.php on line 3 and exactly 2 expected
```

Ici, la fonction `displayUser` attend deux paramètres : lors de l'appel de la fonction, un seul lui est fourni, donc l'exception est attrapée.

Remarque

Contrairement aux exceptions de la classe `Exception`, qu'il faut lancer dans le code, les exceptions de classe `Error` concernant des erreurs fatales sont lancées nativement par PHP. Dans ce cas précis, l'erreur n'a pas été générée manuellement, mais c'est PHP qui s'en est chargé pour nous.

Par contre, il faut tout de même prévoir le bloc `catch` pour attraper l'erreur.

Syntaxe **À retenir**

- Les exceptions implémentent l'interface **Throwable**.
- Cette interface permet de récupérer des détails relatifs à l'exception levée.
- PHP met à disposition nativement la gestion de certaines erreurs. Leur utilisation est possible avec les classes **Error**, **Exception** et les autres classes qui étendent ces deux-là.

Complément

Throwable¹

Exception²

Error³

XI. Exercice : Appliquez la notion

Le code suivant doit permettre de supprimer un objet de type `User`.

On considère qu'il n'est pas possible de supprimer un utilisateur s'il est administrateur, auquel cas une exception est levée.

```
1 <?php
2
3 function removeUser(User $user)
4 {
5     if ($user->isAdmin()) {
6         throw new Exception('Vous ne pouvez pas supprimer un administrateur');
7     }
8
9     // code métier qui supprime un utilisateur
10    return true;
11 }
12
13 $user1 = new User('Anthony', [User::ROLE_ADMIN]);
14 $user2 = new User('Camille', [User::ROLE_USER]);
15
16 $usersToRemove = [$user1, $user2];
17
18 foreach ($usersToRemove as $user) {
19     try {
20         removeUser($user);
21     } catch (Exception $exception)
22     {
23         echo $exception->getMessage() . PHP_EOL;
```

1 <https://www.php.net/manual/fr/class.throwable.php>

2 <https://www.php.net/manual/fr/class.exception.php>

3 <https://www.php.net/manual/fr/class.error.php>

```

24     } finally {
25         echo 'Utilisateur traité' . PHP_EOL;
26     }
27 }
28
29 class User
30 {
31     public $name;
32
33     public $roles = [];
34
35     public const ROLE_ADMIN = 'ROLE_ADMIN';
36     public const ROLE_USER = 'ROLE_USER';
37
38     public function __construct(string $name, array $roles)
39     {
40         $this->name = $name;
41         $this->roles = $roles;
42     }
43
44     public function isAdmin()
45     {
46         return in_array(self::ROLE_ADMIN, $this->roles);
47     }
48 }
49

```

Question

[solution n°5 p.29]

Lorsqu'une exception est levée, on souhaite afficher :

- Le message de l'exception
- Le code erreur
- Le fichier et la ligne concernée
- La stack trace (ou trace d'appels)

Sous la forme :

"[Code]-Message:Fichier ligne"

Stack trace

XII. Créer des exceptions

Objectif

- Créer une exception personnalisée

Mise en situation

Selon les besoins, les exceptions de base fournies par PHP peuvent ne pas correspondre à nos besoins.

Il peut s'avérer nécessaire de réécrire certaines exceptions afin d'y ajouter des fonctionnalités. Cette réécriture peut répondre au besoin d'identifier plus facilement des exceptions métier.

Grâce à l'héritage, nous pourrions créer nos propres classes d'exceptions, sur la base de celles existantes.

Méthode Implémenter ses propres exceptions

Ces nouvelles classes devront obligatoirement implémenter l'interface **Throwable**.

Il n'est pas possible de l'implémenter directement, il faudra donc étendre les classes **Exception** ou **Error**.

L'héritage impose quelques règles.

Ainsi, il est possible d'utiliser les attributs protégés des classes parents **Exception** ou **Error** :

- `$message` : message de l'exception,
- `$code` : code de l'exception,
- `$file` : fichier d'où est lancée l'exception,
- `$line` : ligne d'où est lancée l'exception.

Il est également possible de réécrire les méthodes `__construct()` et `__toString()`.

Toutes les autres méthodes étant déclarées avec le mot-clé `final` ne pourront être réécrites, seulement utilisées telles quelles.

Exemple

Imaginons une exception `ParamException` qui serait déclenchée pour toutes les erreurs concernant les paramètres d'une fonction.

```
1 <?php
2
3 class ParamException extends Exception
4 {
5     public function __construct($message, $code=0)
6     {
7         parent::__construct($message, $code);
8     }
9 }
10
11 <?php
12
13 try {
14     displayUser(20);
15 } catch (ParamException $e) {
16     echo $e->getMessage();
17 }
18
19 function displayUser($user) {
20     if (!is_string($user)) {
21         throw new ParamException ('Le paramètre doit être une chaîne de caractères');
22     }
23
24     echo "Vous êtes $user";
25 }
26 ?>
```

L'exception `ParamException` peut maintenant être utilisée.

Syntaxe À retenir

- Pour écrire sa propre classe d'exception, il faut obligatoirement étendre les classes de PHP `Exception` ou `Error`.

XIII. Exercice : Appliquez la notion

Le code suivant doit permettre de supprimer un objet de type `User`.

On considère qu'il n'est pas possible de supprimer un utilisateur s'il est administrateur, auquel cas une exception est levée.

Actuellement, lorsqu'une exception est levée, celle-ci est formatée de façon spécifique.

```

1 <?php
2
3 function removeUser(User $user)
4 {
5     if ($user->isAdmin()) {
6         throw new Exception('Vous ne pouvez pas supprimer un administrateur');
7     }
8
9     // code métier qui supprime un utilisateur
10    return true;
11 }
12
13 $user1 = new User('Anthony', [User::ROLE_ADMIN]);
14 $user2 = new User('Camille', [User::ROLE_USER]);
15
16 $usersToRemove = [$user1, $user2];
17
18 foreach ($usersToRemove as $user) {
19     try {
20         removeUser($user);
21     } catch (Exception $exception) {
22         echo sprintf("[%s] - %s: %s ligne %s", $exception->getCode(), $exception-
>getMessage(), $exception->getFile(), $exception->getLine());
23     } finally {
24         echo 'Utilisateur traité' . PHP_EOL;
25     }
26 }
27
28 class User
29 {
30     public $name;
31
32     public $roles = [];
33
34     public const ROLE_ADMIN = 'ROLE_ADMIN';
35     public const ROLE_USER = 'ROLE_USER';
36
37     public function __construct(string $name, array $roles)
38     {
39         $this->name = $name;
40         $this->roles = $roles;
41     }
42
43     public function isAdmin()
44     {
45         return in_array(self::ROLE_ADMIN, $this->roles);
46     }
47 }
48

```

Question

[solution n°6 p.30]

Afin de pouvoir réutiliser le formatage qui a été mis en place, implémentez une nouvelle exception de type `UserAdminException`.

Une méthode `displayDetails` permettra d'afficher les détails de l'exception de façon formatée.

Le code sera modifié en conséquence.

XIV. Auto-évaluation**A. Exercice final****Exercice 1**

[solution n°7 p.31]

Exercice

Sur un serveur de production, il est fortement conseillé...

- ☐ D'afficher directement les erreurs à l'utilisateur final
- ☐ De masquer les erreurs à l'utilisateur final

Exercice

Grâce à quel mot-clé une exception est-elle déclenchée ?

- ☐ `throw`
- ☐ `trigger`
- ☐ `try`

Exercice

Lorsqu'une exception n'est pas attrapée, quel type d'erreur est déclenché ?

- ☐ Une erreur de type *Notice*
- ☐ Une erreur de type *Warning*
- ☐ Une erreur fatale

Exercice

Quel ordre faut-il respecter pour attraper une exception ?

- ☐ `catch/try/finally`
- ☐ `finally/try/catch`
- ☐ `try/catch/finally`

Exercice

Quel est le rôle du bloc `finally` ?

- ☐ Exécuter du code si une erreur a été attrapée
- ☐ Exécuter du code si une erreur n'a pas été attrapée
- ☐ Exécuter du code qu'importe la situation

Exercice

Si une erreur n'a pas été attrapée et qu'une erreur fatale est levée, le code exécuté dans le bloc `finally`...

- ☐ Sera exécuté
- ☐ Ne sera pas exécuté

Exercice

Si le code suivant est exécuté et qu'une erreur est levée, qu'advient-il du code situé après le déclenchement de l'erreur ?

```
1 <?php
2
3 function removeUser(User $user)
4 {
5     if ($user->isAdmin()) {
6         throw new Exception('Vous ne pouvez pas supprimer un administrateur');
7         echo 'Une erreur est survenue';
8     }
9
10    return true;
11 }
```

- ☐ Il sera tout de même exécuté
- ☐ À partir du moment où une erreur est lancée, le code situé après ne sera pas exécuté

Exercice

Que se passera-t-il si l'on exécute le code suivant ?

```
1 <?php
2
3 function removeUser(string $username)
4 {
5     if ('John' === $username) {
6         throw new Exception('Vous ne pouvez pas supprimer un administrateur');
7     }
8
9     return true;
10 }
11
12 $username = 'John';
13
14 try {
15     removeUser($username);
16 } catch (InvalidArgumentException $exception) {
17     echo $exception->getMessage();
18 }
19
```

- ☐ Il sera affiché "Vous ne pouvez pas supprimer un administrateur"
- ☐ Une erreur fatale sera levée, car une exception n'a pas été interceptée
- ☐ Le code sera exécuté et la fonction retournera `true`

Exercice

Selon le code suivant, par quelle fonction l'exception sera-t-elle interceptée ?

```
1 <?php
2
3 function removeUser(string $username)
4 {
5     if ('John' === $username) {
6         throw new Exception('Vous ne pouvez pas supprimer un administrateur');
7     }
8
9     return true;
10 }
11
12 function manageUser(string $username, string $action)
13 {
14     if ('remove' === $action) {
15         try {
16             removeUser($username);
17         } catch (InvalidArgumentException $exception) {
18             echo $exception->getMessage();
19         }
20     }
21 }
22
23 function processData()
24 {
25     $username = 'John';
26
27     try {
28         manageUser($username, 'remove');
29     } catch (Exception $exception) {
30         echo $exception->getMessage();
31     }
32 }
33 }
34
35 processData();
36
```

- ☐ removeUser
- ☐ manageUser
- ☐ processData
- ☐ L'exception ne sera pas interceptée

Exercice

Pour créer une exception personnalisée, il est nécessaire...

- ☐ D'étendre la classe Exception, qui implémente l'interface Throwable
- ☐ D'étendre la classe Throwable, qui implémente l'interface Exception

B. Exercice : Défi

"Never trust user input"

Vous disposez du formulaire de création d'un utilisateur, ainsi que de son script de traitement. Actuellement, ce script permet de créer un objet "utilisateur" à partir des données saisies dans le formulaire.

Ces données ne sont pas vérifiées, on considère que la personne qui remplit le formulaire est bienveillante et qu'elle connaît toutes les règles de gestion qui font un bon utilisateur.

Vous allez devoir remettre cette confiance en question et implémenter les vérifications adéquates au niveau du traitement du formulaire.

À terme, l'un de vos collègues sera chargé d'implémenter certaines vérifications côté HTML, mais là n'est pas le sujet.

```

1 <!DOCTYPE html>
2 <html lang="fr">
3 <head>
4     <meta charset="utf-8" />
5 </head>
6 <body>
7 <form method="post" action="action.php">
8     <fieldset>
9         <legend>Création de votre compte</legend>
10        <label for="name">Nom d'utilisateur</label>
11        <p class="help-text">
12            Le nom d'un utilisateur ne peut contenir plus de 20 caractères et ne doit
13            contenir que des lettres.
14        </p>
15        <input type="text" name="name" id="name" required placeholder="Saisissez votre nom">
16
17        <label for="email">Adresse e-mail</label>
18        <p class="help-text">Il doit s'agir d'un e-mail valide</p>
19        <input type="text" name="email" id="email" required placeholder="Saisissez votre
20        email">
21
22        <label for="password">Votre mot de passe</label>
23        <p class="help-text">
24            Le mot de passe ne peut contenir que des chiffres et des lettres. Il doit disposer
25            d'au moins 6 caractères.
26        </p>
27        <input type="password" name="password" id="password" required placeholder="Saisissez
28        votre mot de passe">
29
30        <label for="birthDate">Votre date de naissance</label>
31        <p class="help-text">
32            L'utilisateur doit être une personne majeure
33        </p>
34        <input type="date" name="birthDate" id="birthDate" required placeholder="Saisissez
35        votre date de naissance">
36
37        <label for="comment">Commentaire</label>
38        <p class="help-text">
39            Le commentaire ne peut excéder plus de 200 caractères
40        </p>
41        <textarea name="comment" id="comment" cols="30" rows="10"></textarea>
42
43        <button type="submit">Soumettre le formulaire</button>
44    </fieldset>
45 </form>
46 </body>
47 <style>

```

```

43  /**
44      Attention, ce CSS est là uniquement pour rendre le formulaire "agréable" à la lecture
45      sans que vous n'ayez
46      à récupérer deux fichiers distincts.
47      Dans un cas d'usage "réel", ces éléments doivent être externalisés
48      */
49  body {
50      font-family: Calibri, serif;
51  }
52  form {
53      max-width: 50%;
54  }
55
56  form label {
57      display: block;
58      font-weight: bold;
59      margin-bottom: 10px;
60  }
61
62  fieldset {
63      border: 1px solid lightgray;
64      background-color: rgba(225, 233, 255, 0.25);
65  }
66
67  legend {
68      font-style: italic;
69      font-size: 1.1em;
70      padding: 5px;
71  }
72
73  form input, form textarea {
74      display: inline-block;
75      margin-bottom: 10px;
76      padding: 10px;
77      width: 80%;
78  }
79
80  button[type=submit] {
81      padding: 10px;
82      margin-top: 15px;
83      width: auto;
84  }
85
86  .help-text {
87      color: #2b2b2b;
88      font-size: 12px;
89  }
90 </style>
91 </html>
92
1  <?php
2
3  processForm();
4
5
6  function processForm()
7  {

```

```
8     $user = new User();
9     $formFields = ['name', 'birthDate', 'email', 'password', 'comment'];
10
11     foreach ($formFields as $field) {
12         $user->{'set' . ucfirst($field)}($_POST[$field]);
13     }
14
15     echo 'Utilisateur complet';
16 }
17
18 /**
19  * Class User
20  */
21 class User
22 {
23     private $name;
24
25     private $birthDate;
26
27     private $email;
28
29     private $password;
30
31     private $comment;
32
33     /**
34      * @param string $name
35      */
36     public function setName(string $name)
37     {
38         $this->name = $name;
39     }
40
41     /**
42      * @param string $birthDate
43      */
44     public function setBirthDate(string $birthDate)
45     {
46         $this->birthDate = $birthDate;
47     }
48
49     /**
50      * @param string $email
51      */
52     public function setEmail(string $email)
53     {
54         $this->email = $email;
55     }
56
57     /**
58      * @param string $password
59      */
60     public function setPassword(string $password)
61     {
62         $this->password = $password;
63     }
64
65 }
```



```

66  /**
67   * @param string|null $comment
68   */
69  public function setComment(?string $comment)
70  {
71      $this->comment = $comment;
72  }
73
74  /**
75   * @return bool
76   */
77  public function isFull()
78  {
79      return !empty($this->name) && !empty($this->password) && !empty($this->birthDate) &&
80      !empty($this->email);
81  }
82 }
83

```

Question

[solution n°8 p.34]

La création d'un utilisateur doit dépendre d'un certain nombre de règles de gestion. Vous êtes chargé de vérifier ces données côté PHP.

Ainsi :

- Le nom d'un utilisateur ne peut contenir plus de 20 caractères
- Ce nom ne doit contenir que des lettres
- La date de naissance doit être une date valide
- L'utilisateur doit être majeur, ainsi il doit avoir au moins 18 ans
- Son mot de passe ne peut contenir que des chiffres et des lettres
- Un mot de passe doit avoir au moins 6 caractères
- L'e-mail doit être au bon format
- Un commentaire ne peut contenir plus de 200 caractères

Une Exception devra être levée dès lors que l'une de ces conditions n'est pas respectée.

Chacune des propriétés est définie grâce à son `setter`. C'est là que vous effectuerez ces vérifications. Vous définirez les exceptions que vous jugerez nécessaires.

Pour vous guider, il existe des fonctions PHP pour vérifier si une chaîne de caractères est constituée uniquement de lettres, de lettres ou de chiffres, ou de chiffres.

Par une recherche Internet, vous trouverez facilement le nom de ces fonctions, et comment les utiliser.

Pour vérifier le format de la date fournie par l'utilisateur, vous pourrez utiliser la fonction `date_create`. Cette fonction formate une date passée en paramètre pour en faire un objet `DateTime`. Si le paramètre transmis n'est pas une date (donc n'est pas convertible vers l'objet `DateTime`), la fonction `date_create` retournera `false`.

Pour vérifier le format d'un e-mail, vous pouvez utiliser le filtre de validation `FILTER_VALIDATE_EMAIL`. Son appel se fait grâce à la méthode `filter_var`, de la façon suivante : `filter_var($email, FILTER_VALIDATE_EMAIL)`, où `$email` est la variable dont on veut valider le format. Cette fonction retournera un booléen.

Solutions des exercices

p.4 Solution n°1

Les trois erreurs seront les suivantes :

- `$defaultRole = DEFAULT_ROLE;` déclenchera un warning, car on essaie d'assigner à la variable une constante qui n'a pas été définie. Cela ne provoquera pas d'arrêt du script.
- `echo $city;` déclenchera quant à elle un notice, car `$city` n'est pas définie.
- enfin, une erreur fatale sera déclenchée lors de l'appel `greetings($name)` ; , car l'un des paramètres requis n'a pas été instancié.

p.7 Solution n°2

```

1 <?php
2 // On désactive l'affichage des erreurs
3 ini_set('display_errors', 'off');
4
5 // On vérifie qu'on a bien deux variables de définies
6 if (isset($_GET['nombre1']) && isset($_GET['nombre2'])) {
7     // On vérifie qu'il s'agit de numériques
8     if (is_numeric($_GET['nombre1']) && is_numeric($_GET['nombre2'])) {
9         // On vérifie qu'on n'essaie pas de faire une division par zéro
10        if ((int)$_GET['nombre2'] !== 0) {
11            $division = $_GET['nombre1'] / $_GET['nombre2'];
12
13            echo $division;
14        } else {
15            echo 'Vous ne pouvez pas diviser un nombre par zéro';
16        }
17    } else {
18        echo 'Vous devez fournir des nombres';
19    }
20 } else {
21     echo 'Vous devez définir deux paramètres';
22 }
```

p.10 Solution n°3

```

1 <?php
2
3 // On vérifie qu'on a bien deux variables de définies
4 if (isset($_GET['nombre1']) && isset($_GET['nombre2'])) {
5     // On vérifie qu'il s'agit de numériques
6     if (is_numeric($_GET['nombre1']) && is_numeric($_GET['nombre2'])) {
7         // On vérifie qu'on n'essaie pas de faire une division par zéro
8         if ((int)$_GET['nombre2'] !== 0) {
9             $division = $_GET['nombre1'] / $_GET['nombre2'];
10
11             echo $division;
12         } else {
13             throw new Exception('Vous ne pouvez pas diviser un nombre par zéro');
14         }
15     } else {
16         throw new Exception('Vous devez fournir des chiffres');
```

```

17     }
18 } else {
19     throw new Exception('Vous devez définir deux nombres');
20 }
21
22
23

```

p. 13 Solution n°4

```

1 <?php
2
3 function removeUser(User $user)
4 {
5     if ($user->isAdmin()) {
6         throw new Exception('Vous ne pouvez pas supprimer un administrateur');
7     }
8
9     // code métier qui supprime un utilisateur
10    return true;
11 }
12
13 $user1 = new User('Anthony', [User::ROLE_ADMIN]);
14 $user2 = new User('Camille', [User::ROLE_USER]);
15
16 $usersToRemove = [$user1, $user2];
17
18 foreach ($usersToRemove as $user) {
19     try {
20         removeUser($user);
21     } catch (Exception $exception)
22     {
23         echo $exception->getMessage() . PHP_EOL;
24     } finally {
25         echo 'Utilisateur traité' . PHP_EOL;
26     }
27 }
28
29 class User
30 {
31     public $name;
32
33     public $roles = [];
34
35     public const ROLE_ADMIN = 'ROLE_ADMIN';
36     public const ROLE_USER = 'ROLE_USER';
37
38     public function __construct(string $name, array $roles)
39     {
40         $this->name = $name;
41         $this->roles = $roles;
42     }
43
44     public function isAdmin()
45     {
46         return in_array(self::ROLE_ADMIN, $this->roles);
47     }

```

48 }
49

p. 16 Solution n°5

```

1 <?php
2
3 function removeUser(User $user)
4 {
5     if ($user->isAdmin()) {
6         throw new Exception('Vous ne pouvez pas supprimer un administrateur');
7     }
8
9     // code métier qui supprime un utilisateur
10    return true;
11 }
12
13 $user1 = new User('Anthony', [User::ROLE_ADMIN]);
14 $user2 = new User('Camille', [User::ROLE_USER]);
15
16 $usersToRemove = [$user1, $user2];
17
18 foreach ($usersToRemove as $user) {
19     try {
20         removeUser($user);
21     } catch (Exception $exception) {
22         echo sprintf("[%s] - %s: %s ligne %s", $exception->getCode(), $exception->
23 >getMessage(), $exception->getFile(), $exception->getLine());
24         echo '<pre>';
25         foreach ($exception->getTrace() as $item) {
26             var_dump($item);
27         }
28     } finally {
29         echo 'Utilisateur traité' . PHP_EOL;
30     }
31 }
32
33 class User
34 {
35     public $name;
36
37     public $roles = [];
38
39     public const ROLE_ADMIN = 'ROLE_ADMIN';
40     public const ROLE_USER = 'ROLE_USER';
41
42     public function __construct(string $name, array $roles)
43     {
44         $this->name = $name;
45         $this->roles = $roles;
46     }
47
48     public function isAdmin()
49     {
50         return in_array(self::ROLE_ADMIN, $this->roles);
51     }
52 }

```

p. 19 Solution n°6

```

1 <?php
2
3 function removeUser(User $user)
4 {
5     if ($user->isAdmin()) {
6         throw new UserAdminException('Vous ne pouvez pas supprimer un administrateur');
7     }
8
9     // code métier qui supprime un utilisateur
10    return true;
11 }
12
13 $user1 = new User('Anthony', [User::ROLE_ADMIN]);
14 $user2 = new User('Camille', [User::ROLE_USER]);
15
16 $usersToRemove = [$user1, $user2];
17
18 foreach ($usersToRemove as $user) {
19     try {
20         removeUser($user);
21     } catch (UserAdminException $exception) {
22         echo $exception->displayDetails();
23     } finally {
24         echo 'Utilisateur traité' . PHP_EOL;
25     }
26 }
27
28 class UserAdminException extends Exception {
29     public function __construct($message, $code=0)
30     {
31         parent::__construct($message, $code);
32     }
33
34     public function displayDetails()
35     {
36         echo sprintf("[%s] - %s: %s ligne %s", $this->getCode(), $this->getMessage(), $this->getFile(), $this->getLine());
37     }
38 }
39
40 class User
41 {
42     public $name;
43
44     public $roles = [];
45
46     public const ROLE_ADMIN = 'ROLE_ADMIN';
47     public const ROLE_USER = 'ROLE_USER';
48
49     public function __construct(string $name, array $roles)
50     {
51         $this->name = $name;
52         $this->roles = $roles;


```

```
53     }  
54  
55     public function isAdmin()  
56     {  
57         return in_array(self::ROLE_ADMIN, $this->roles);  
58     }  
59 }  
60
```

Exercice p. 19 Solution n°7

Exercice

Sur un serveur de production, il est fortement conseillé...

- ☐ D'afficher directement les erreurs à l'utilisateur final
- ☒ De masquer les erreurs à l'utilisateur final
-  En mode production, c'est-à-dire en utilisation réelle, les erreurs ne devraient pas être affichées à l'utilisateur. Celles-ci pourraient permettre à un utilisateur mal intentionné de découvrir et d'exploiter des informations sensibles liées à l'application.

Exercice

Grâce à quel mot-clé une exception est-elle déclenchée ?

- ☒ throw
- ☐ trigger
- ☐ try

Exercice

Lorsqu'une exception n'est pas attrapée, quel type d'erreur est déclenché ?

- ☐ Une erreur de type *Notice*
- ☐ Une erreur de type *Warning*
- ☒ Une erreur fatale

Exercice

Quel ordre faut-il respecter pour attraper une exception ?

- ☐ catch/try/finally
- ☐ finally/try/catch
- ☒ try/catch/finally

Exercice

Quel est le rôle du bloc `finally` ?

- ☐ Exécuter du code si une erreur a été attrapée
- ☐ Exécuter du code si une erreur n'a pas été attrapée
- ☒ Exécuter du code qu'importe la situation

Exercice

Si une erreur n'a pas été attrapée et qu'une erreur fatale est levée, le code exécuté dans le bloc `finally`...

- ☒ Sera exécuté
- ☐ Ne sera pas exécuté
- ☐ Le bloc `finally` sera quand même exécuté avant la levée de l'erreur fatale, ce qui peut permettre d'effectuer des opérations avant l'interruption du script.

Exercice

Si le code suivant est exécuté et qu'une erreur est levée, qu'advient-il du code situé après le déclenchement de l'erreur ?

```
1 <?php
2
3 function removeUser(User $user)
4 {
5     if ($user->isAdmin()) {
6         throw new Exception('Vous ne pouvez pas supprimer un administrateur');
7         echo 'Une erreur est survenue';
8     }
9
10    return true;
11 }
```

- ☐ Il sera tout de même exécuté
- ☒ À partir du moment où une erreur est lancée, le code situé après ne sera pas exécuté

Exercice

Que se passera-t-il si l'on exécute le code suivant ?

```
1 <?php
2
3 function removeUser(string $username)
4 {
5     if ('John' === $username) {
6         throw new Exception('Vous ne pouvez pas supprimer un administrateur');
7     }
8
9     return true;
10 }
11
12 $username = 'John';
13
14 try {
15     removeUser($username);
16 } catch (InvalidArgumentException $exception) {
17     echo $exception->getMessage();
18 }
19
```

- ☐ Il sera affiché "Vous ne pouvez pas supprimer un administrateur"
- ☒ Une erreur fatale sera levée, car une exception n'a pas été interceptée
- ☐ Le code sera exécuté et la fonction retournera `true`

- Q Une erreur fatale sera levée, car on lève une Exception tout en n'interceptant que InvalidArgumentException.

Exercice

Selon le code suivant, par quelle fonction l'exception sera-t-elle interceptée ?

```
1 <?php
2
3 function removeUser(string $username)
4 {
5     if ('John' === $username) {
6         throw new Exception('Vous ne pouvez pas supprimer un administrateur');
7     }
8
9     return true;
10 }
11
12 function manageUser(string $username, string $action)
13 {
14     if ('remove' === $action) {
15         try {
16             removeUser($username);
17         } catch (InvalidArgumentException $exception) {
18             echo $exception->getMessage();
19         }
20     }
21 }
22
23 function processData()
24 {
25     $username = 'John';
26
27     try {
28         manageUser($username, 'remove');
29     } catch (Exception $exception) {
30         echo $exception->getMessage();
31     }
32 }
33 }
34
35 processData();
36
```

- ☐ removeUser
- ☐ manageUser
- ☒ processData
- ☐ L'exception ne sera pas interceptée

Exercice

Pour créer une exception personnalisée, il est nécessaire...

- ☒ D'étendre la classe Exception, qui implémente l'interface Throwable
- ☐ D'étendre la classe Throwable, qui implémente l'interface Exception

p. 25 Solution n°8

```

1 <?php
2 try {
3     processForm();
4 } catch (InvalidUserException $exception) {
5     echo $exception->getMessage() . "<br/>";
6 }
7
8
9 function processForm()
10 {
11     $user = new User();
12     $formFields = ['name', 'birthDate', 'email', 'password', 'comment'];
13
14     foreach ($formFields as $field) {
15         try {
16             $user->{'set' . ucfirst($field)}($_POST[$field]);
17         } catch (Exception $exception) {
18             echo $exception->getMessage() . "<br/>";
19         }
20     }
21
22     if (!$user->isFull()) {
23         throw new InvalidUserException('Une erreur est survenue : l\'utilisateur est
24         incomplet');
25     }
26
27     echo 'Utilisateur complet';
28 }
29 /**
30  * Class InvalidUserException
31  *
32  * Exception métier lorsqu'un objet User est invalide
33  */
34 class InvalidUserException extends Exception {
35     public function __construct($message, $code=0)
36     {
37         parent::__construct($message, $code);
38     }
39 }
40
41 /**
42  * Class InvalidContentTypeException
43  *
44  * Exception lorsqu'un type de contenu n'est pas valide
45  */
46 class InvalidContentTypeException extends Exception {
47     public function __construct($message, $code=0)
48     {
49         parent::__construct($message, $code);
50     }
51 }
52
53 /**
54  * Class InvalidDateTimeException
55  *

```

```

56 * Exception lorsqu'une date est invalide
57 */
58 class InvalidDateTimeException extends Exception {
59     public function __construct($message, $code=0)
60     {
61         parent::__construct($message, $code);
62     }
63 }
64
65 /**
66 * Class User
67 */
68 class User
69 {
70     private $name;
71
72     private $birthDate;
73
74     private $email;
75
76     private $password;
77
78     private $comment;
79
80     public const NAME_MAX_LENGTH = 20;
81
82     public const AGE_MIN_VAL = 18;
83
84     public const PASSWORD_MIN_LENGTH = 6;
85
86     public const COMMENT_MAX_LENGTH = 200;
87
88     public function setName(string $name)
89     {
90         if (strlen($name) > self::NAME_MAX_LENGTH) {
91             // PHP implémente nativement certaines exceptions, autant les utiliser
92             throw new LengthException("La propriété name ne peut excéder
93 ".self::NAME_MAX_LENGTH." caractères");
94         }
95
96         if (!ctype_alpha($name)) {
97             throw new InvalidContentTypeException("La propriété name ne peut contenir que des
98 lettres");
99         }
100
101         $this->name = $name;
102     }
103
104     public function setBirthDate(string $birthDate)
105     {
106         if (!date_create($birthDate)) {
107             throw new InvalidDateTimeException('La date de naissance n\'a pas pu être
108 identifiée');
109         }
110
111         $birthDate = date_create($birthDate);
112         $now = date_create();
113
114         if ($birthDate > $now) {

```

```

112         throw new InvalidDateTimeException('La date de naissance ne peut se situer dans
113         le futur');
114     }
115     if (date_diff($birthDate, $now)->y < self::AGE_MIN_VAL) {
116         throw new InvalidDateTimeException('Vous devez être majeur pour valider ce
117         formulaire');
118     }
119     $this->birthDate = $birthDate;
120 }
121
122 public function setEmail(string $email)
123 {
124     if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
125         throw new InvalidContentTypeException("La propriété email est invalide");
126     }
127
128     $this->email = $email;
129 }
130
131 public function setPassword(string $password)
132 {
133     if (strlen($password) < self::PASSWORD_MIN_LENGTH) {
134         throw new LengthException("La propriété password ne peut excéder
135         ".self::PASSWORD_MIN_LENGTH." caractères");
136     }
137
138     if (!ctype_alnum($password)) {
139         throw new InvalidContentTypeException("La propriété password ne peut contenir que
140         des lettres ou des chiffres");
141     }
142
143     $this->password = $password;
144 }
145
146 public function setComment(?string $comment)
147 {
148     if (strlen($comment) > self::COMMENT_MAX_LENGTH) {
149         throw new LengthException("La propriété comment ne peut contenir plus de
150         ".self::COMMENT_MAX_LENGTH." caractères");
151     }
152
153     $this->comment = $comment;
154 }
155
156 public function isFull()
157 {
158     return !empty($this->name) && !empty($this->password) && !empty($this->birthDate) &&
159     !empty($this->email);
160 }
161 }

```