

La gestion des erreurs

Table des matières

I. Contexte	3
II. Les erreurs en PHP (à partir de PHP 7)	3
A. Définition d'une erreur en PHP et comportement par défaut.....	3
B. Gérer « proprement » les erreurs	5
C. Exercice : Quiz	11
III. PHP moderne : gérer les erreurs avec les Exceptions	12
A. Les exceptions, les lancer et les attraper.....	12
B. Les différents types d'Exception	16
C. Exercice : Quiz	17
IV. Essentiel	18
V. Auto-évaluation	18
A. Exercice	18
B. Test	20
Solutions des exercices	21

I. Contexte

Durée : 1 h

Prérequis : maîtriser les bases de PHP

Environnement de travail : PHP (>v7), IDE Visual Studio Code ou terminal et éditeur de texte standard

Contexte

Les programmes, comme les êtres humains, doivent faire face aux imprévus. L'exécution d'un programme, contrairement à sa conception, a lieu dans le monde réel : un programme tourne et utilise les ressources d'une machine, d'un réseau, échange avec d'autres programmes, etc. Il y aura toujours des imprévus (une donnée incomplète, une panne, le serveur d'une base de données qui met trop de temps à répondre, etc.). Nous devons donc vivre avec, mais surtout les anticiper et nous en protéger, nous et nos clients. Une erreur mal gérée peut conduire votre service web à tomber en panne, ou encore révéler par mégarde des informations sensibles (fuite d'informations confidentielles, informations sur votre système exploitables par des personnes malveillantes, etc.). La gestion des erreurs et des exceptions est donc un sujet important que vous devez comprendre et maîtriser avant de déployer vos projets en production. Les erreurs et les exceptions sont justement des outils mis à votre disposition par PHP pour vous permettre d'anticiper et de réagir en cas d'événements imprévus.

Dans ce cours, nous aborderons le fonctionnement des erreurs et des exceptions en PHP. C'est un sujet qui peut porter à confusion, car depuis PHP 7, et pour des raisons historiques, la différence entre erreur et exception est souvent mal comprise. Nous allons éclaircir la différence entre les deux et apprendre comment les utiliser à votre avantage pour programmer de manière défensive, afin que vos programmes échouent avec grâce. S'il y a un message à retenir de ce module, c'est le suivant : « *En PHP, gérez les erreurs et utilisez les exceptions* ».

II. Les erreurs en PHP (à partir de PHP 7)

A. Définition d'une erreur en PHP et comportement par défaut

Attention Version de PHP

Ce module couvre la gestion des erreurs à partir de PHP 7 !

Aujourd'hui (depuis PHP 5 et surtout PHP 7) on utilise les exceptions plutôt que les erreurs. Mais vous devez quand même comprendre comment les erreurs fonctionnent et surtout comment les gérer, car toutes les fonctions fournies par PHP les utilisent.

PHP 7 a changé la manière dont la plupart des erreurs (mais pas toutes !) sont gérées par le moteur de PHP. La plupart des erreurs sont aujourd'hui reportées en lançant un objet de type **Error**, à la manière des exceptions. Comprendre le nouveau système d'erreurs de PHP vous aidera donc à comprendre celui des exceptions, abordées dans la seconde partie de ce module.

Définition Les erreurs PHP

Une erreur se déclenche lorsque votre script PHP ne peut pas s'exécuter comme prévu (par exemple, un script à charger n'existe pas, une autorisation de lecture de fichier est refusée, l'espace disque est insuffisant, une classe n'existe pas, etc.).

Une erreur est caractérisée par 4 valeurs :

- **Un niveau d'erreur.** Chaque niveau est défini par une constante PHP dont le nom commence par **E_**. Par exemple, une erreur de niveau 1 est une erreur fatale définie par la constante **E_ERROR**. Une erreur de niveau 2 est un avertissement défini par la constante **E_WARNING**. Le niveau d'erreur indique au moteur PHP quoi faire : arrêter ou continuer l'exécution du programme. Par exemple, une erreur de niveau 1 (Fatal Error) arrête l'exécution du programme.

- Un message d'erreur.
- Le nom du fichier PHP dans lequel l'erreur s'est produite.
- Le numéro de ligne du fichier où l'erreur s'est produite.

Lorsqu'une erreur est émise, un rapport est généré par le système de gestion d'erreurs de PHP. Par défaut, lorsqu'aucune politique de gestion n'est mise en place, le rapport est simplement affiché sur la sortie standard. Un rapport est ainsi constitué pour chaque erreur rencontrée. Chaque rapport fournit les quatre informations mentionnées ci-dessus, suivies de la trace d'appels (pile d'exécution au moment de l'erreur, ou *stack trace*). Une fois la génération du rapport terminée, si le niveau d'erreur le permet, l'exécution du programme continue. Sinon, le programme s'arrête.

Attention Vos rapports d'erreur ne doivent pas tomber entre toutes les mains

Ces rapports d'erreur sont à double tranchant : en environnement de développement, ils sont très utiles pour déboguer votre programme. En production, en plus de nuire à votre image, ils peuvent nuire à la sécurité de votre système en livrant des informations précieuses à des personnes mal intentionnées. On prendra soin d'ajuster la politique de gestion d'erreurs en fonction de l'environnement d'exécution. Nous y reviendrons.

Comportement à la génération d'une erreur

Il peut donc se passer plusieurs choses en fonction du niveau de l'erreur :

- Soit l'erreur est fatale (**Fatal Error**, de niveau 1) : le programme reporte l'erreur (par défaut sur la sortie standard) et le programme s'arrête immédiatement (*crash*).
- Soit l'erreur est une alerte (**Warning**, de niveau 2) : le programme reporte l'erreur et continue son exécution.
- Soit l'erreur n'en est peut-être pas une, c'est une simple remarque (**Notice**, de niveau 8) : le programme reporte l'erreur et continue son exécution.
- Etc.

Exemple

Prenez ce programme. Il inclut un script `foo.php` pour charger la classe `Foo`. Malheureusement, le fichier `foo.php` n'est pas accessible sur le serveur ! Que va-t-il se passer ? On se doute que cela va mal se passer, mais de quelle manière exactement ?

```
1 <?php
2 require 'foo.php';
3 echo 'Tout va bien' . PHP_EOL;
```

Voici la sortie de ce programme :

```
1 php index.php
2 PHP Warning: require(foo.php): Failed to open stream: No such file or directory in index.php
  on line 2
3 PHP Fatal error: Uncaught Error: Failed opening required 'foo.php'
  (include_path='.:usr/share/php') in index.php:2
```

Le programme a généré deux rapports d'erreurs : le premier rapport nous indique une erreur de type **Warning**, généré par `require`. En effet, PHP n'a pas eu accès au script `foo.php`. Comme c'est seulement un **Warning**, le programme continue. Le deuxième rapport nous indique cette fois qu'il y a eu une erreur fatale qui a mis fin à l'exécution du script. On voit que cette erreur est toujours générée à la ligne 2, par le *construct* `require`. L'instruction `echo 'Tout va bien' . PHP_EOL;` n'est jamais exécutée.

Que s'est-il passé ? Dans la documentation, on vous dit que `require`, en plus d'émettre un **Warning**, émet également une erreur **E_COMPILE_ERROR**, qui est une erreur fatale mettant fin à l'exécution. Nous avons donc bien sous les yeux les deux rapports d'erreur générés par `require`.

Générer (ou non) des erreurs est un choix de design

Qui détermine la nature de l'erreur ? Qui décide qu'une erreur est fatale ou un simple avertissement ? La personne qui a écrit la fonction !

Générer une erreur est un choix de design. Si la personne chargée de l'écriture de la fonction juge que, dans ce cas, sa fonction ne peut pas travailler sereinement, elle fait le choix d'arrêter son exécution et de générer une erreur pour vous prévenir. C'est ce qu'on appelle la programmation offensive.

Attention Préférez toujours `require` à `include`

Lorsque vous souhaitez inclure un autre script dans le script courant, pour charger une classe par exemple, vous avez le choix entre les deux *constructs* PHP `include` et `require`. La seule différence entre les deux est que `include` lève un simple **Warning** (erreur de niveau 2) si le fichier n'a pas pu être chargé, alors que `require` lèvera en plus une erreur `E_COMPILE_ERROR`, une erreur fatale qui mettra fin immédiatement au programme.

Il est donc recommandé d'utiliser `require` (ou `require_once`) au lieu de `include` (ou `include_once`).

Complément La différence entre `require` et `require_once`

Le construct `require_once` est identique à `require`, sauf qu'il demande à PHP de vérifier si le fichier a déjà été inclus. Si c'est le cas, `require_once` ne l'inclut pas une seconde fois. Cela permet d'éviter des erreurs de redéfinition (classes, fonctions, etc.). La contrepartie est une plus grande consommation de la mémoire du script pour maintenir la liste des fichiers déjà inclus. Il en va de même pour `include` et `include_once`.

B. Gérer « proprement » les erreurs

Le bubbling, quand les erreurs remontent à la surface

Lorsqu'une erreur est lancée par une fonction ou un *construct* PHP, comme `require` dans l'exemple précédent, le flot d'exécution du programme est mis en pause, et PHP remonte la pile d'appels jusqu'à rencontrer quelque chose qui veuille bien s'en occuper. L'erreur, comme une bulle immergée dans un liquide, remonte vers la surface. On appelle ça le *bubbling*, et c'est le même mécanisme qu'utilisent les exceptions.

Il existe deux façons d'attraper une erreur afin de la gérer :

- **Localement, avec un bloc `try/catch (Error)`** . L'erreur est prise dans le filet d'un `try/catch` et stoppe sa remontée. L'exécution reprend ensuite son cours normal après le bloc.
- **Globalement, avec un gestionnaire d'erreurs global**. Si la bulle est passée entre les mailles de vos filets (blocs `try/catch`), le gestionnaire global l'attrape à la toute dernière minute, avant qu'elle n'atteigne la surface.

Gestion locale avec le bloc `try/catch`

Pour s'occuper d'une erreur localement, PHP vous met à disposition un bloc `try/catch (Error)` . Vous devez placer l'instruction qui peut émettre une erreur dans le bloc `try`, et écrire le code à exécuter en cas d'erreur dans le bloc `catch (Error)` . Toute instruction placée après une instruction qui a lancé une erreur n'est pas exécutée. Le flot d'exécution saute directement au bloc `catch`, via le mécanisme de *bubbling*. Il se poursuit ensuite normalement.

Exemple

Reprenons l'exemple précédent, mais cette fois-ci avec une gestion locale de l'erreur.

```
1 <?php
2 try {
3     require 'foo.php';
4     echo 'Cette instruction ne sera pas exécutée si require émet une erreur' . PHP_EOL;
5 } catch (Error $e) {
6     echo "L'erreur " . $e->getMessage() . " est gérée localement" . PHP_EOL;
7 }
8 echo 'Le programme continue ici' . PHP_EOL;
9 ?>
```

La fonction **require** lance une erreur de type **Error**, que l'on attrape dans le bloc **catch(Error \$e) {}**. **Error**¹ est un type (une classe) fourni par PHP et qui possède, comme toute classe, des méthodes et des propriétés. Vous remarquerez que l'erreur fatale émise par **require** ne stoppe plus l'exécution du programme, car elle a été attrapée par le bloc. Le programme va donc continuer, pour le meilleur et pour le pire. Il sera toujours de votre responsabilité de choisir ce qu'il faut faire en cas d'erreur (arrêter ou continuer le programme).

Exemple

Précédemment, nous avons attrapé immédiatement l'erreur, mais on aurait pu l'attraper à un endroit situé plus haut dans la pile. Prenez cet exemple, qui illustre bien le mécanisme interne de *bubbling*.

```
1 <?php
2 /**
3  * Charge un modèle (la classe Foo), mais foo.php n'existe pas
4  * @throws Error - Si le fichier foo.php n'arrive pas à être lu
5  */
6 function loadLModelFoo()
7 {
8     require 'foo.php';
9 }
10
11 /**
12  * Charge un modèle (la classe Bar), mais bar.php n'existe pas
13  * @throws Error - Si le fichier bar.php n'arrive pas à être lu
14  */
15 function loadLModelBar()
16 {
17     require 'bar.php';
18 }
19
20 /**
21  * Charge les modèles
22  * @throws Error - Si un modèle ne peut pas être chargé
23  */
24 function loadModels()
25 {
26     loadLModelFoo();
27     loadLModelBar();
28 }
29
30 try {
31     loadModels();
32     echo 'Cette instruction ne sera pas exécutée si require émet une erreur' . PHP_EOL;
```

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

```

33 } catch (Error $e) {
34     echo "L'erreur " . $e->getMessage() . " est gérée localement" . PHP_EOL;
35 }
36 echo 'Le programme continue ici' . PHP_EOL;

```

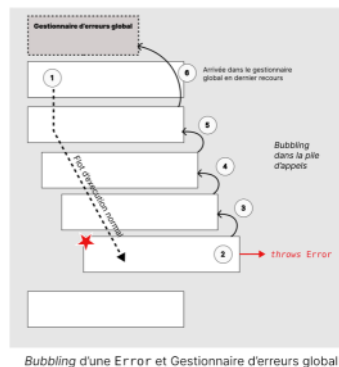
C'est la fonction `loadModelFoo` qui est susceptible d'émettre une erreur (avec `require`). Mais nous pouvons attraper l'exception plus haut dans la pile, à l'appel de la fonction `loadModels`. Comme ça, si `loadModelBar` émet aussi une erreur, les deux cas seront gérés en un seul endroit. Nous pouvons faire cela grâce au mécanisme de *bubbling* des erreurs.

Attention Ne lancez pas des erreurs, gérez-les !

Aujourd'hui, il n'est pas recommandé d'utiliser les erreurs PHP lorsqu'on souhaite émettre une erreur, mais plutôt les exceptions. Les erreurs doivent être uniquement gérées pour les fonctions natives de PHP (les fonctions système, en quelque sorte, de votre application).

Gestion globale avec le gestionnaire d'erreurs global

On peut déclarer un gestionnaire d'erreurs global pour gérer globalement les erreurs émises par nos fonctions PHP. Un gestionnaire d'erreurs global se déclare avec la fonction PHP `set_error_handler($callable, int $error_levels): ?callable`. Il vous permet de personnaliser la génération de vos rapports en enregistrant une fonction `$callable`. Cette fonction sera appelée lorsqu'une erreur sera émise dans votre code. Vous pouvez ainsi, comme dans les blocs `catch`, exécuter du code personnalisé (par exemple, envoyer un e-mail à l'administrateur du site, puis tuer le programme).



Bubbling d'une Error et Gestionnaire d'erreurs global

La fonction `$callable` récupère 4 arguments, fournis par PHP. Vous les connaissez déjà : le niveau d'erreur, le message d'erreur, le fichier PHP à l'origine de l'erreur, et le numéro de la ligne où l'erreur s'est produite.

Essayons de personnaliser notre rapport d'erreur. On voudrait gérer proprement notre erreur pour afficher un message afin d'arrêter le programme de manière contrôlée et avec grâce.

```

1 /**
2  * Définition du gestionnaire d'erreurs global
3  */
4  set_error_handler(function ($errno, $errstr, $errfile, $errline) {
5      echo "Nous sommes désolés, un problème vient de survenir :/ \nNous vous invitons à revenir
6      plus tard." . PHP_EOL;
7      // Si c'est une erreur de niveau 2 (avertissement), l'exécution peut en principe continuer
8      // mais on décide, par précaution, d'arrêter là les frais
9      if ($errno === E_WARNING)
10         die;
11 });

```

```
11
12 require 'foo.php';
13 echo 'Tout va bien' . PHP_EOL;
```

À présent, au lieu d'afficher le rapport d'erreur à l'utilisateur et d'exposer des informations sensibles sur notre système, nous affichons un message d'erreur plus sympathique et engageant grâce à la fonction enregistrée auprès de notre gestionnaire d'erreurs.

Comme notre programme lève un simple **Warning** (erreur de niveau 2), PHP va tout de même chercher à poursuivre l'exécution. Mais nous préférons la stopper ici nous-mêmes avec une instruction **die**. Le *construct* PHP **die** met fin immédiatement, et sans poser de questions, à l'exécution du programme. Martin Fowler appelle cette façon de gérer l'erreur « *l'équivalent logiciel d'un suicide si vous manquez un vol* ». Mais si le coût de l'arrêt est faible et que l'utilisateur est tolérant, arrêter un programme est une solution acceptable. Ce qui est souvent le cas sur le Web.

C'est ce que l'on appelle échouer gracieusement (*fails gracefully*) : notre programme souffre et nous ne pouvons rien y faire lors de l'exécution, mais nous pouvons au moins décider de sa chute.

Attention Le gestionnaire d'erreurs global ne peut pas traiter les erreurs fatales !

Attention cependant, le gestionnaire d'erreurs global à lui seul ne permet pas de gérer toutes les erreurs, notamment les erreurs fatales ! Celles-ci peuvent être gérées localement par des blocs `try/catch`, mais soyez prudents si vous décidez de poursuivre l'exécution tout de même. Une erreur fatale indique que le bon fonctionnement du programme n'est plus garanti.

Méthode Installation de l'IDE VS Code

L'IDE VS Code (Visual Studio Code) est utilisé pour les démonstrations vidéo de ce module. VS Code est un IDE multiplateforme (Windows, mac OS X, GNU/Linux) très populaire basé sur les technologies du Web et développé par Microsoft. Vous pouvez le télécharger en vous rendant sur le site officiel¹. L'extension Intelephense² est également recommandée pour PHP. Elle fournit notamment l'autocomplétion des fonctions fournies par PHP et permet d'accéder à leurs documentations, ainsi qu'à votre code source commenté, directement depuis l'IDE au survol des fonctions. Vous pouvez l'installer directement depuis VS Code, via le panneau d'extension, ou en suivant les indications fournies sur la page web de l'extension.

Méthode Configuration du système de génération des rapports d'erreurs

Vous pouvez également configurer le système de génération de rapports (le *reporting* de PHP) de manière globale. Vous pouvez par exemple dire au système de gestion d'erreurs de ne reporter que des erreurs d'un certain niveau et d'ignorer les autres. Pour ce faire, utilisez la fonction PHP `error_reporting()` ou la directive `error_reporting` dans votre fichier de configuration `php.ini`. Le système de rapport d'erreurs peut être ajusté comme vous le souhaitez, du silence le plus complet (déconseillé !) à la sensibilité la plus extrême.

Voici les règles que vous devriez suivre :

- Ne désactivez jamais votre rapport d'erreur.
- Afficher les erreurs dans l'environnement de développement, mais n'affichez pas les erreurs dans un environnement de production.
- Faites toujours un *log* (enregistrement horodaté dans un fichier dédié) de vos erreurs, en développement comme en production. Cela vous fournira des informations précieuses pour comprendre ce qui s'est mal passé et déboguer votre code.
- Mettez en place un système d'alerte où vos erreurs (Warning, Exceptions, etc.) sont redirigées vers les administrateurs de votre système, via des alertes SMS ou mails, ou vers des applications web de *monitoring* en temps réel.

1 <https://code.visualstudio.com/>

2 <https://marketplace.visualstudio.com/items?itemName=bmewburn.vscode-intelephense-client>

Voici une configuration type de système *reporting*, à l'aide de directives PHP, pour un environnement de développement sur un serveur.

```
1;Afficher les erreurs
2display_startup_errors = On
3display_errors = On
4;Rapporter toutes les erreurs
5error_reporting = -1
6;Activer le log d'erreurs
7log_errors = On
```

Et une configuration typique pour un environnement de production :

```
1;N'afficher SURTOUT PAS les erreurs
2display_startup_errors = Off
3display_errors = Off
4;Rapporter toutes les erreurs SAUF les notice (pour ne pas polluer vos logs)
5error_reporting = E_ALL & ~E_NOTICE
6;Activer le log d'erreurs
7log_errors = On
```

Vous pouvez retrouver l'ensemble des directives PHP disponibles dans la documentation officielle de PHP.¹

Attention Inspectez les codes d'erreur avant de réagir

Lorsque vous personnalisez votre système de rapport d'erreurs, gardez à l'esprit ceci : PHP va envoyer toutes vos erreurs vers votre gestionnaire d'erreur global, même celles que vous avez exclues de votre configuration (par exemple, les **Notice**). Vous revient donc la charge d'inspecter vous-même les codes d'erreur et d'agir en conséquence.

```
1set_error_handler(function ($errno, $errstr, $errfile, $errline) {
2
3    //Filtre des erreurs par Masquage de bits
4    if (!(error_reporting() & $errno)) {
5        //L'erreur n'est pas spécifiée dans la configuration du
6        //système de rapport d'erreur, donc on l'ignore
7        return;
8    }
9
10   echo "Nous sommes désolés, un problème vient de survenir :/ \Nous vous invitons à revenir
11   plus tard." . PHP_EOL;
12   die;
13});
```

Méthode Enregistrez toujours un gestionnaire d'erreurs global, même en PHP 8 !

Le système d'erreur de PHP est le système le plus ancien, contrairement aux exceptions qui sont arrivées avec PHP 5. Si la plupart des erreurs ont été transformées aujourd'hui en lancement d'objets de type **Error**, à la manière des exceptions attrapables par un bloc **try/catch(Error)**, certaines fonctions comme **fopen** (pour ouvrir un accès à un fichier) déclenchent toujours des erreurs à l'ancienne, sans lancer d'objets **Error**.

Celles-ci ne seront donc pas attrapées par un bloc **try/catch**, mais uniquement par le système de rapport d'erreurs. D'où la nécessité de toujours enregistrer un système de rapport d'erreurs traditionnel, même en PHP 8.

Pour vous en convaincre, essayez donc le code suivant.

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

```
1 set_error_handler(function ($errno, $errstr, $errfile, $errline) {
2     echo 'Par contre, fopen, comme tout le monde, reporte bien ses erreurs auprès du
    gestionnaire d\'erreurs global.' . PHP_EOL;
3 });
4 try {
5     fopen('foo.php', 'r');
6 } catch (Error $e) {
7     echo 'Vous ne verrez jamais ce message car fopen n\'émet pas d\'erreurs de type Error !
    L\'erreur est déclenchée de manière classique' . PHP_EOL;
8 }
```

La sortie vous montre que **fopen** ne lance pas d'**Error**. Pourtant, l'erreur qu'elle génère est bien reportée auprès du gestionnaire d'erreurs global. Un des vestiges historiques de PHP encore visible aujourd'hui.

Si tout cela vous rend confus, c'est normal, et vous n'êtes pas seul ! Retenez donc simplement ceci : il faut toujours enregistrer un gestionnaire d'erreur global, même si celui-ci ne peut pas traiter toutes les erreurs, notamment les erreurs fatales.

Conseil Restaurer le gestionnaire par défaut

Il est recommandé de toujours restaurer le gestionnaire d'erreur global précédent (s'il y'en a un) une fois votre code exécuté. Cela peut être utile notamment lorsque vous utilisez un *framework* ou si avez un besoin spécifique de rapport d'erreurs sur une partie précise de votre code. Vous pouvez restaurer le gestionnaire précédent avec la fonction **restore_error_handler()**.

```
1 set_error_handler(function ($errno, $errstr, $errfile, $errline) {
2
3     if (!(error_reporting() & $errno)) {
4         //L'erreur n'est pas spécifiée dans la configuration du
5         //système de rapport d'erreur, donc on l'ignore
6         return;
7     }
8
9     echo "Nous sommes désolés, un problème vient de survenir :/ \nNous vous invitons à revenir
    plus tard." . PHP_EOL;
10    die;
11 });
12
13 //Votre code...
14
15 //Restauration du gestionnaire d'erreur précédent
16 restore_error_handler();
```

Fondamental

Dans cette première partie, nous avons vu que PHP dispose d'un système de gestion d'erreurs interne. Depuis PHP 7, la quasi-majorité des erreurs (mais pas toutes, pensez à la fonction **fopen** !) est gérée par le lancement d'objets de type **Error**. Lorsqu'une erreur est lancée, un rapport est généré par défaut. Vous pouvez configurer le système de rapport d'erreurs dans votre fichier de configuration **php.ini**, personnaliser le contenu du rapport et exécuter du code lorsque votre code rencontre une erreur afin de stopper votre programme en douceur, de manière sécurisée, et dans le respect de vos utilisateurs.

Nous avons vu qu'une erreur, une fois émise (*thrown*), remonte la pile d'exécution comme une bulle (*bubbling* de l'erreur) jusqu'à être attrapée par un bloc **try/catch(Error)** ou par le gestionnaire d'erreurs global, qui agit comme un garde-fou de dernier recours (sauf pour les erreurs fatales qu'il ne peut pas gérer).

Aujourd'hui, les erreurs (objets de type **Error**) sont réservées et utilisées par les fonctions natives de PHP (pour des raisons historiques). Vous devez donc savoir les gérer. En revanche, votre code applicatif ne devrait pas en émettre (n'écrivez jamais d'instructions **throw new Error()**, sauf si, vraiment, vous avez une bonne raison de le faire). Comme le fait toute la communauté PHP aujourd'hui (*frameworks*, bibliothèques, etc.), utilisez plutôt les exceptions.

C. Exercice : Quiz

[solution n°1 p.23]

Question 1

Parmi les propositions suivantes, trouvez celle qui est correcte :

- ☐ Les erreurs ne devraient jamais arriver dans un programme bien écrit
- ☐ L'arrivée d'erreurs et leur gestion font partie intégrante de la vie d'un programme en cours d'exécution
- ☐ Si une erreur se produit, on ne peut rien y faire

Question 2

Compléter la proposition suivante : « En PHP, les erreurs [...] » :

- ☐ Font partie du cœur du système. La plupart des erreurs ont été réécrites en PHP 7 sous la forme d'objets de type **Error** qui sont lancés par le programme et remontent la pile d'appels. Il est alors possible de les attraper avec un bloc **try/catch**
- ☐ Sont désuètes et ne sont plus utilisées
- ☐ Doivent être utilisées et émises par notre code lorsque notre programme rencontre un problème

Question 3

Complétez la proposition suivante : « Lorsqu'une erreur est lancée, si on ne l'attrape pas [...] ».

- ☐ L'exécution du programme s'arrête immédiatement
- ☐ Un rapport est généré (par défaut sur la sortie standard) et le programme s'arrête immédiatement
- ☐ Un rapport est généré (par défaut sur la sortie standard) et, en fonction du niveau de l'erreur, le programme continue son exécution ou s'arrête

Question 4

Il n'est pas nécessaire ni important de définir un gestionnaire d'erreurs global.

- ☐ Vrai
- ☐ Faux

Question 5

Parmi les propositions suivantes, trouvez celle qui est correcte.

- ☐ La politique de gestion des erreurs doit être la même en environnement de développement qu'en environnement de production
- ☐ La politique de gestion des erreurs doit être adaptée à l'environnement d'exécution
- ☐ PHP gère automatiquement pour nous la gestion des erreurs en fonction de l'environnement d'exécution

III. PHP moderne : gérer les erreurs avec les Exceptions

A. Les exceptions, les lancer et les attraper

Fondamental L'objet de type Exception

Les exceptions ont été introduites en PHP 7. Elles sont l'évolution orientée objet du système de gestion d'erreur de PHP. Aujourd'hui, toute la communauté utilise les exceptions dans le code applicatif au lieu des erreurs, et vous devez faire de même.

Une exception est une instance de la classe **Exception** fournie par PHP. Comme les objets **Error** vus précédemment, une **Exception** est lancée (*thrown*) quand une situation imprévue se présente (par exemple, lorsqu'une requête à une base de données n'aboutit pas). Dans ces situations exceptionnelles, les exceptions sont utilisées offensivement pour déléguer la responsabilité des actions à mener à l'utilisateur du code. Elles sont également utilisées défensivement : en tant qu'utilisateur d'un code tiers (une bibliothèque, par exemple), nous devons anticiper les exceptions lancées.

Pour instancier une **Exception**, il faut utiliser l'opérateur **new** comme pour n'importe quel objet. Le constructeur d'exception attend trois arguments, tous optionnels :

- Un message qui décrit ce qui s'est mal passé.
- Un code qui permet d'ajouter du contexte sur l'imprévu.
- L'exception à l'origine de celle-ci, si elle existe, afin de pouvoir accéder à la chaîne des exceptions à n'importe quel niveau de gestion dans votre programme.

```
1 <?php
2 $exception = new Exception('Attention, le programme rencontre une situation imprévue', 42);
```

Pour accéder au message et au code, vous pouvez utiliser les méthodes **getMessage()** et **getCode()** prévue à cet effet :

```
1 <?php
2 $exception = new Exception('Attention, le programme rencontre une situation imprévue', 42);
3 $exception->getMessage(); // Attention, le programme rencontre une situation imprévue
4 $exception->getCode(); // 42
```

Complément Le code permet d'ajouter du contexte et de documenter votre système

Même si le code d'une exception (une valeur entière quelconque) est optionnel, il peut s'avérer utile. Vous pouvez créer une table de codes d'exception (que vous devrez documenter et maintenir à jour) à laquelle vous pouvez vous référer, ou que vous pouvez afficher côté utilisateur pour faciliter l'intervention en cas de problème.

C'est une pratique très courante dans l'électroménager, par exemple. Lorsque votre machine à laver cesse de fonctionner et vous affiche un code d'erreur, vous pouvez ouvrir le manuel et aller directement voir sa signification, et ce que vous pouvez faire pour la remettre en état.

Lancer des exceptions

On peut instancier une exception comme on vient de le faire. Mais les exceptions sont faites pour être lancées. Si vous écrivez du code pour d'autres programmeurs (vos collègues, un *framework*, une bibliothèque, etc.), vous devez utiliser les exceptions de manière offensive : vous devez lancer des exceptions lorsque votre code rencontre une situation où les conditions ne sont pas réunies pour faire un travail correct.

C'est là l'avantage des exceptions (ou des instances de classe **Erreur** introduites par PHP 7) : vous déléguiez la gestion de la situation exceptionnelle à l'utilisateur de votre code. En effet, c'est à lui de savoir ce qu'il y a de mieux à faire dans le cas de son application. Vous n'avez aucune raison d'imposer aux autres une manière de la gérer, comme d'arrêter l'exécution du programme.

Pour lancer une exception, on utilise le mot-clé **throw**.

```

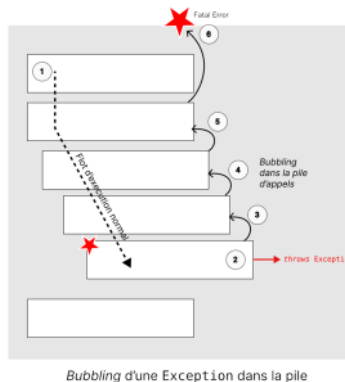
1 <?php
2 throw new Exception('Attention, le programme rencontre une situation imprévue');
3 //Ce code ne sera pas exécuté
4 $foo = 42;

```

Attention Seuls les objets de type Throwable peuvent être « lancés »

Le mot-clé **throw** ne peut être utilisé que sur des instances de classe implémentant l'interface **Throwable**. **Throwable** est l'interface de base pour tout objet qui peut être lancé, comme les objets de type **Error** et **Exception**. L'interface **Throwable** est réservée au fonctionnement interne de PHP, vous ne pouvez pas la faire implémenter par vos classes. Si vous voulez créer des objets « jetables », ils doivent être de type **Exception** (donc leur classe doit étendre la classe **Exception**).

Lorsqu'une **Exception** est lancée, le mécanisme de *bubbling* se met en place : le flot d'exécution normal est suspendu, et l'**Exception** remonte la pile d'appels jusqu'à ce qu'elle soit interceptée par un bloc **try/catch(Exception \$e)** ou par un gestionnaire d'exception global, exactement comme pour nos objets **Error** vus précédemment. Si une **Exception** remonte la pile jusqu'en haut, et que personne ne l'a attrapée, elle explose avec perte et fracas en générant une **Fatal Error**, ce qui mettra fin brusquement à votre programme.



Méthode Attraper des exceptions

Lors de la phase de *bubbling*, vous pouvez attraper l'exception avec un bloc **try/catch(Exception \$e)**. Cela vous permet de gérer l'exception à l'endroit qui vous convient le mieux, de manière locale, et de reprendre l'exécution de votre programme selon vos propres conditions (ou de l'arrêter de manière gracieuse). Pour attraper une exception, il faut placer la fonction susceptible de lancer une exception dans un bloc **try{ }**. Il est de votre responsabilité d'attraper les exceptions, personne ne le fera pour vous. C'est ce qu'on appelle programmer de manière défensive : vous devez vous assurer que vous protégez votre code contre les situations exceptionnelles.

Le code suivant montre une tentative de connexion à une base de données **MySQL** qui échoue. Au lieu de récupérer la *stack trace* et d'afficher des détails sur notre système à l'utilisateur, on affiche un message *user-friendly* :

```

1 <?php
2 try {
3     //Tentative de connexion à une base de données MySQL avec le module PDO (PHP Data Object)
4     $pdo = new PDO('mysql://host=foo;dbname=bar');
5 } catch (Exception $e) {
6     //Inspector l'exception
7     $code = $e->getCode();
8     $message = $e->getMessage();
9     //Afficher un message user-friendly
10    echo "Mmmm, impossible d'accéder à la base de données. Veuillez réessayer plus tard." .
    PHP_EOL;

```

```
11 }
12 //L'execution continue ici
```

Complément Le mécanisme d'exception n'est pas sans inconvénients

Le mécanisme de *bubbling* est pratique, car il vous permet de gérer l'exception au niveau qui vous convient dans votre code. Cependant, il n'est pas sans inconvénient : au lancement d'une exception, le flot d'exécution est altéré et saute d'appel en appel jusqu'à être attrapé par un bloc `catch`. Le flot d'exécution devient ainsi plus difficile à suivre. Une politique de gestion des exceptions trop lourde peut rapidement devenir une source de complexité supplémentaire et indésirable dans votre code.

Le gestionnaire d'exceptions global, votre dernier rempart

Comme nous l'avons déjà dit, une fois lancées, les exceptions remontent la pile d'appel comme des bulles vers la surface. On parle de *bubbling*. Ce mécanisme est une alternative au « *suicide logiciel* » de Martin Fowler, « *c'est l'équivalent logiciel de se jeter de la fenêtre d'un immeuble en espérant que quelqu'un nous attrape dans un filet avant de toucher le sol* » (*PHP in Action*, de David Reiersol, Marcus Baker, Chris Shiflett, Manning Publication 2007).

Comme pour les **Error** vues dans la partie « **Les erreurs en PHP (à partir de PHP 7)** », il est possible de définir une politique de manière globale pour attraper les exceptions grâce à un gestionnaire d'exceptions global. Ainsi, si vous avez oublié d'attraper toutes les exceptions possibles dans votre code (et cela arrivera !), vous aurez la certitude qu'au moins un composant de votre programme y veille, et attrape toutes celles qui auraient échappé à vos filets. Comme vous devez toujours définir un gestionnaire d'erreurs global, vous devez toujours définir un gestionnaire d'exceptions global (oui, les deux !).

Un gestionnaire d'exceptions global se définit avec la fonction PHP `set_exception_handler`, et prend en argument un *callback* qui sera appelé lorsque votre **Exception** sera interceptée. Rien de nouveau, c'est exactement le même principe que pour le gestionnaire d'erreurs global.

```
1 <?php
2 //Définition d'un gestionnaire d'exception global
3 set_exception_handler(function(Exception $e){
4     //Effectuer des actions avant de mettre fin à l'exécution.
5     //Gérer l'exception ici et "échouez gracieusement"
6     echo 'Une Exception a été détectée. Nous mettons fin au programme.' . PHP_EOL;
7 });
8 throw new Exception();
9 //Ce code ne sera jamais exécuté
10 $foo = 42;
```

Conseil Restaurer aussi le gestionnaire global d'exception par défaut

De la même manière que pour le gestionnaire global d'erreurs, il est de bon usage de restaurer le gestionnaire d'exceptions global précédent, si jamais il y en avait un, une fois votre travail terminé. Pour ce faire, utilisez la fonction `restore_exception_handler()` à la fin de votre programme.

```
1 <?php
2 //Définition d'un gestionnaire d'exception global
3 set_exception_handler(function(Exception $e){
4     //Effectuer des actions avant de mettre fin à l'exécution.
5     //Gérer l'exception ici et "échouez gracieusement"
6     echo 'Une Exception a été détectée. Nous mettons fin au programme.' . PHP_EOL;
7 });
8 //Votre code...
9
10 //A la fin
```

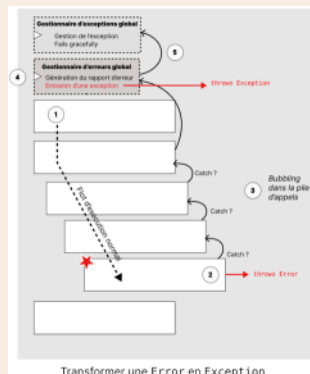
```
11 restore_exception_handler();
```

Complément Transformer les Error en Exception

Mais que faire de toutes ces fonctions PHP qui génèrent encore des erreurs ? Et que faire des erreurs fatales ? Eh bien, comme on l'a vu, depuis PHP 7, la plupart d'entre elles lancent des objets de type **Error**. Une manière populaire de gérer les objets **Error** et **Exception** est de récupérer l'objet **Error** et de le relancer en tant que **Exception**. Cette conversion permet de pouvoir travailler principalement avec l'interface des exceptions uniquement.

Où transformer les **Error** en **Exception** ? Dans votre gestionnaire d'erreurs global, bien sûr !

```
1 <?php
2 //Définition d'un gestionnaire d'exceptions global
3 set_exception_handler(function (Throwable $e) {
4     echo 'Une Exception a été détectée. Nous mettons fin au programme.' . PHP_EOL;
5 });
6
7 //Définition d'un gestionnaire d'erreurs global
8 set_error_handler(function ($errno, $errstr, $errfile, $errline) {
9     //Transformer l'erreur en Exception avec la classe dédiée ErrorException
10    echo 'Une erreur a été détectée. On la relance sous forme d\'Exception !' . PHP_EOL;
11    throw new ErrorException($errstr, $errno, 0, $errfile, $errline);
12 });
13
14 //Test : lancer une erreur avec l'une des deux instructions suivantes
15 // throw new Error();
16 trigger_error('Oups !', E_USER_WARNING);
17 //Ce code ne sera jamais exécuté...
18 echo '42 !' . PHP_EOL;
```



Attention Toutes les erreurs ne peuvent pas être transformées en exceptions !

Toutes les erreurs ne peuvent pas être transformées en **Exception** ! Ces erreurs incluent **E_ERROR**, **E_PARSE**, **E_CORE_ERROR**, **E_CORE_WARNING**, **E_COMPILE_ERROR**, **E_COMPILE_WARNING** et la plupart des **E_STRICT**. Pourquoi ? Car ces erreurs sont des erreurs fatales et ne peuvent pas être gérées par le gestionnaire d'erreurs global vu précédemment. Donc vous ne pouvez pas les relancer comme des exceptions ! Pour des raisons historiques et une volonté de rétrocompatibilité entre les versions, le système de gestion d'erreurs de PHP peut dérouter... C'est pour cette raison que, dans notre gestionnaire d'exceptions global, on passe à notre fonction de *callback* un type **Throwable**. Ce type est à la base de tout ce qui peut être lancé en PHP : les types **Error** et **Exception** sont des sous-types de **Throwable** (leurs classes en héritent). Cela nous assure d'attraper toutes les erreurs lancées dans notre gestionnaire d'exceptions global, y compris celles qui sont fatales.

B. Les différents types d'Exception

Un type d'exception pour chaque contexte

PHP vous fournit, en plus de la classe de base, tout un ensemble de sous-classes d'**Exception** pour apporter davantage de contexte aux erreurs rencontrées. En voici quelques exemples :

- **ErrorException** : représente une instance d'**Error** en **Exception**.
- **DomainException** : représente une erreur dans logique métier de votre application (par exemple, une valeur non définie pour le domaine métier).
- **LogicException** : représente une erreur dans la logique de votre programme, pour vous aider à le déboguer.
- Etc.

Chaque classe existe pour une situation donnée et vous permet d'ajouter toujours plus de contexte à l'erreur rencontrée. Ajouter du contexte (où s'est produite l'erreur, dans quelle situation, dans quel état de votre programme, pour quelles raisons précises, etc.) vous permet de mieux comprendre les erreurs et de corriger ou d'améliorer votre programme plus facilement. Il vous permet aussi de mettre en place différentes stratégies de gestion d'erreur en fonction du contexte (actions à mener, log, envoi d'e-mails, etc.).

Méthode Attraper différents types d'Exception

Il est possible d'enchaîner plusieurs blocs **catch** à la suite d'un bloc **try** pour attraper les différents types d'**Exception** et réaliser un traitement particulier pour chacun d'eux. Attention, le premier bloc **catch** qu'une **Exception** rencontre et qui correspond à son type gèrera l'**Exception** ! Vous pouvez également utiliser un bloc **finally** pour exécuter du code après n'importe quelle exception attrapée.

```
1 <?php
2 try {
3     //On lance une exception du type de base Exception
4     throw new Exception('Pas une Exception de type PDO !');
5     //Lève une exception PDOException si la connexion à la base échoue
6     $pdo = new PDO('mysql://host=foo;dbname=bar');
7 } catch (PDOException $e) {
8     //L'exception est attrapée par ce premier bloc qui match l'Exception
9     echo 'PDOException attrapée. Traitement.' . PHP_EOL;
10 } catch (Exception $e) {
11     echo 'Exception attrapée. Traitement.' . PHP_EOL;
12 } finally {
13     echo 'On exécute toujours ce bloc de code, peu importe l\'exception attrapée.' . PHP_EOL;
14 }
```

Méthode Créer ses propres exceptions

Comme **Exception** est une classe et qu'elle n'est pas **finale**, elle peut être étendue pour créer vos propres classes d'**Exception**, avec leurs propriétés et leurs méthodes propres. Il est cependant recommandé de conserver une signature du constructeur identique à celle de la classe mère **Exception**.

```
1 class MyException extends Exception
2 {
3     public __construct(string $message = "", int $code = 0, ?Throwable $previous = null)
4     {
5         parent::__construct($message, $code, $previous);
6         //Votre logique...
7     }
8 }
9 throw new MyException();
```


En général, les différentes exceptions fournies par PHP, ainsi que les messages et codes d'erreur, fournissent assez de contexte pour couvrir vos besoins. Mais vous êtes libre d'implémenter les vôtres. Si vous le faites, demandez-vous : « *Pourquoi est-ce que je lance ce type d'exception ?* » et documentez votre choix.

C. Exercice : Quiz

[solution n°2 p.24]

Question 1

Lorsqu'une Exception est lancée, elle...

- ☐ Suspend immédiatement le flot d'exécution du programme et remonte la pile d'appels jusqu'à être attrapée
- ☐ Génère automatiquement une erreur
- ☐ Arrête l'exécution du programme immédiatement

Question 2

Une Exception se caractérise au moins par...

- ☐ Un type (**Exception** ou classe étendant la classe **Exception**), un message, un code et une *stack trace*
- ☐ Un code uniquement
- ☐ Un message uniquement

Question 3

Quelle est la sortie produite par le code suivant ?

```
1 <?php
2 try {
3     throw new PDOException();
4 } catch (Exception $e) {
5     echo 'Exception attrapée. Traitement.' . PHP_EOL;
6 } catch (PDOException $e) {
7     echo 'PDOException attrapée. Traitement.' . PHP_EOL;
8 } finally {
9     echo 'On execute toujours ce bloc de code, peu importe l\'exception attrapée.' . PHP_EOL;
10 }
```

- ☐ PDOException attrapée. Traitement.
On execute toujours ce bloc de code, peu importe l\'exception attrapée.
- ☐ PDOException attrapée. Traitement.
Exception attrapée. Traitement.
On execute toujours ce bloc de code, peu importe l\'exception attrapée.
- ☐ Exception attrapée. Traitement.
On execute toujours ce bloc de code, peu importe l\'exception attrapée.

Question 4

À quoi sert un gestionnaire d'exceptions global ?

- ☐ À attraper automatiquement les exceptions qui, en remontant la pile d'appels, n'ont rencontré aucun bloc **try/catch**
- ☐ À rien, car toutes les exceptions sont attrapées dans des blocs **try/catch**
- ☐ À rien si on a déjà défini un gestionnaire d'erreurs global : les deux font exactement la même chose

Question 5

Qu'est-ce que la programmation défensive ?

- ☐ Émettre des exceptions quand votre programme rencontre une situation qu'il ne peut pas traiter avec certitude
- ☐ Attraper les exceptions émises par votre code ou celui d'autres personnes pour être sûr de parer aux pires situations
- ☐ Tester votre code en amont

IV. Essentiel

En tant que programmeur, il est essentiel de savoir gérer les imprévus et de concevoir des programmes capables d'atterrir en douceur plutôt que de les laisser se crasher, sous peine de froisser ou perdre la confiance de vos utilisateurs et de vos clients, ou d'exposer des informations de votre système à des personnes malveillantes.

Dans ce module, nous avons vu comment gérer les erreurs rencontrées par votre programme au cours de son exécution. Nous avons vu que le système de PHP de gestion d'erreurs (depuis la version 7) était principalement basé sur des objets de type **Error**. Ces objets sont lancés lorsqu'une fonction PHP rencontre une situation où elle ne peut pas s'exécuter correctement. Une fois lancée, une **Error** donne lieu à l'établissement d'un rapport, puis elle remonte la pile d'appels jusqu'à être attrapée. Il est possible de configurer le système de *reporting* de notre système d'erreur avec des directives dans le fichier de configuration **php.ini**.

Les erreurs sont utilisées par les fonctions et les *constructs* fournies par PHP, mais vous ne devez pas les utiliser pour votre code applicatif : vous devez utiliser les exceptions à la place. Gérez seulement les **Error** proprement, grâce à des blocs **try/catch**, un gestionnaire d'erreurs global et un système de *reporting* adapté à votre environnement d'exécution.

Les **Exceptions** fonctionnent comme les objets **Error** : lorsqu'elles sont lancées, elles suspendent le flot d'exécution et remontent la pile d'appel jusqu'à être attrapées. Vous devez utiliser le système des exceptions dans votre code en cas d'imprévus : soit en émettre, soit les attraper. Nous avons vu qu'il était possible d'attraper les exceptions avec des blocs **try/catch** afin de gérer localement le problème rencontré et de poursuivre l'exécution du programme, mais également avec le gestionnaire d'exceptions global, au cas où des **Exceptions** échappent à vos filets, sous peine de récupérer une **Fatal Error**. Vous pouvez également définir vos exceptions propres à votre application au besoin.

Que ce soit dans votre gestionnaire d'erreurs ou dans votre gestionnaire d'exception, pensez à toujours écrire des *logs* de vos erreurs, en environnement de développement ou de production.

V. Auto-évaluation

A. Exercice

Vous êtes développeur et devez écrire un programme PHP pour l'entreprise Strategy, en vue d'améliorer un processus de publication vers le Web. L'entreprise Strategy est dans le domaine de la veille stratégique. Elle recense et traite quotidiennement des centaines de publications dans différents domaines : académique, juridique, commercial, etc. Depuis peu, elle récupère des résumés (*abstracts*) de publications au format Markdown. Elle aimerait pouvoir les transformer automatiquement au format HTML pour les publier sur un site web.

Question 1

[solution n°3 p.26]

Votre programme doit récupérer chaque *abstract* depuis une URL puis l'enregistrer dans un fichier. Votre programme récupère une liste d'*abstracts* sous la forme suivante :

```

1 $abstracts = [
2     [
3         'id' => 1,
4         'name' => 'git',
5         'url' => 'http://git.test/abstract'
6     ],
7     [
8         'id' => 2,
9         'name' => null,
10        'url' => 'http://php.test/abstract'
11    ],
12    [
13        'id' => 3,
14        'name' => 'handling-errors-best-practices',
15        'url' => 'http://programming.test/abstract/handling-errors/best-practices'
16    ],
17    [
18        'id' => 4,
19        'name' => 'geocities',
20        'url' => 'geocities.test/abstract'
21    ],
22 ];

```

Où :

- **id** est un identifiant unique de l'*abstract*,
- **name** est le nom du fichier sous lequel l'*abstract* va être enregistré sur votre serveur,
- **url** est l'URL de la ressource pour récupérer le contenu de l'*abstract* au format Markdown.

La liste contient parfois des erreurs : elle ne contient pas toujours le nom de la ressource, ou l'URL est parfois mal définie. C'est le cas de deux items dans cette liste (*id*=2 où le nom est manquant et *id*=4 où l'URL est invalide). Vous devez écrire un programme qui filtre les ressources *abstract* invalides (données incomplètes ou mal formées). Il doit afficher sur la sortie standard à la fin du traitement :

- Le nombre d'abstracts valides / nombre d'abstracts total ("**Nombre d'abstracts valides : 2/4** ")
- La liste des id des abstracts invalides ("**Liste des abstracts invalides (id): 2, 4**")

Les *abstracts* invalides doivent être signalés par le lancement d'une exception de type **DomainException**, attrapée dans un bloc **try/catch**. Cela permettra de créer une politique de *log* efficace ultérieurement. Une URL valide contient nécessairement le schéma '**http://**', un nom valide ne doit pas être **null** (absence de nom).

Question 2

[solution n°4 p.27]

Définissez un gestionnaire d'exceptions global dans votre programme. Celui-ci doit écrire un *log* horodaté de l'erreur incluant : la date et l'heure au format **y-m-d h:m:s**, le message d'erreur, le code d'erreur, le nom du fichier où l'*Exception* a été levée et le numéro de ligne. Testez-le en lançant une exception avec le message '**Oups !**' et le code 400 dans l'espace global.

Indice :

Pour écrire un fichier de *log*, utilisez la fonction `error_log`¹.

¹ <https://www.php.net/manual/en/function.error-log.php>

B. Test

Exercice 1 : Quiz

[solution n°5 p.28]

Question 1

Le fichier mylib.php n'existe pas dans le dossier courant du script. Quelle est la sortie du code suivant ?

```
1 <?
2 require 'mylib.php';
3 echo 'mylib loaded ! ' . PHP_EOL;
```

- ☐ PHP Warning: require(mylib.php): Failed to open stream: No such file or directory in index.php on line 1
PHP Fatal error: Uncaught Error: Failed opening required 'mylib.php' (include_path=!.:/usr/share/php) in index.php:1
- ☐ mylib loaded !
- ☐ PHP Warning: require(mylib.php): Failed to open stream: No such file or directory in index.php on line 1
mylib loaded !

Question 2

PHP nous permet de définir nos propres exceptions.

- ☐ Vrai
- ☐ Faux

Question 3

Quelle est la sortie du code suivant ?

```
1 <?php
2 /**
3  * Retourne l'âge de la personne s'il est censé
4  * @throws DomainException Si l'âge a une valeur insensée
5  */
6 function acceptedAge(int $age): int
7 {
8     if ($age < 0 || $age > 150)
9         throw new DomainException("L'âge n'est pas valide !");
10    return $age;
11 }
12
13 try {
14     acceptedAge(200);
15     acceptedAge(12);
16 } catch (Exception $e) {
17     echo $e->getMessage() . PHP_EOL;
18 } catch (DomainException $e) {
19     echo 'Erreur dans le domaine : ' . $e->getMessage() . PHP_EOL;
20 }
```

- ☐ L'âge n'est pas valide !
- ☐ Erreur dans le domaine : L'âge n'est pas valide !
- ☐ L'âge n'est pas valide !
12

Question 4

Une Exception qui n'est pas attrapée remonte la pile d'appels jusque dans le contexte global, et finit par générer une Fatal Error.

- ☐ Vrai
- ☐ Faux

Question 5

Il est impossible de récupérer (poursuivre l'exécution du programme) après une Fatal Error (erreur fatale).

- ☐ Vrai
- ☐ Faux

Solutions des exercices

Exercice p. 11 Solution n°1**Question 1**

Parmi les propositions suivantes, trouvez celle qui est correcte :

- ☐ Les erreurs ne devraient jamais arriver dans un programme bien écrit
- ☒ L'arrivée d'erreurs et leur gestion font partie intégrante de la vie d'un programme en cours d'exécution
- ☐ Si une erreur se produit, on ne peut rien y faire
- ☐ Les erreurs arrivent toujours, même dans un programme « *bien* » écrit et conçu. Tout simplement car votre programme est exécuté sur une machine et qu'il interagit avec le monde extérieur (*hardware*, mémoire, OS, Web, API, autres programmes, etc.). Il faut donc anticiper les erreurs et penser à les gérer afin de réduire l'impact qu'elles peuvent avoir sur le système, sur l'utilisateur de votre code ou sur un client. On peut donc y faire quelque chose : quand l'erreur arrive, il est trop tard, mais la plupart des langages de programmation, dont PHP, nous permettent de décider comment gérer les imprévus et faire échouer notre programme « *avec grâce* ».

Question 2

Compléter la proposition suivante : « *En PHP, les erreurs [...]* » :

- ☒ Font partie du cœur du système. La plupart des erreurs ont été réécrites en PHP 7 sous la forme d'objets de type **Error** qui sont lancés par le programme et remontent la pile d'appels. Il est alors possible de les attraper avec un bloc **try/catch**
- ☐ Sont désuètes et ne sont plus utilisées
- ☐ Doivent être utilisées et émises par notre code lorsque notre programme rencontre un problème
- ☐ Les erreurs en PHP ne sont pas désuètes, elles sont utilisées par toutes les fonctions de PHP. La plupart ont été récemment réécrites à partir de PHP 7 pour fonctionner comme les exceptions : elles sont lancées et arrêtent le flot du programme jusqu'à ce qu'elles soient attrapées. Cependant, les erreurs ne doivent pas être lancées par le code applicatif, il faut préférer les exceptions, qui sont aujourd'hui utilisées par toute la communauté PHP. Il faut donc toujours gérer les erreurs, mais ne pas en émettre soi-même.

Question 3

Complétez la proposition suivante : « *Lorsqu'une erreur est lancée, si on ne l'attrape pas [...]* ».

- ☐ L'exécution du programme s'arrête immédiatement
- ☐ Un rapport est généré (par défaut sur la sortie standard) et le programme s'arrête immédiatement
- ☒ Un rapport est généré (par défaut sur la sortie standard) et, en fonction du niveau de l'erreur, le programme continue son exécution ou s'arrête
- ☐ Par défaut, une erreur génère toujours un rapport. Le programme ne s'arrête pas forcément, cela dépend du niveau de l'erreur. Par exemple, une Fatal Error arrête l'exécution car elle n'est plus considérée comme sûre. Par contre, une erreur de type Notice ou Warning génère un rapport, mais l'exécution se poursuit quand même.

Question 4

Il n'est pas nécessaire ni important de définir un gestionnaire d'erreurs global.

- ☐ Vrai
- ☒ Faux

Q Il faut toujours définir un gestionnaire d'erreurs global dans son code ! Sinon, des erreurs remontent à la surface et peuvent s'afficher sur la sortie standard (et donc sur une page web envoyée au client dans un contexte web) des informations sensibles sur le système ou les utilisateurs. C'est un garde-fou nécessaire pour se protéger, personnaliser le rapport d'erreur et réagir (écrire un *log*, envoyer un e-mail à l'administrateur, supprimer des données temporaires, etc.) avant d'éventuellement mettre un terme à l'exécution du programme. Attention cependant, certains types d'erreurs ne peuvent pas être gérés par le gestionnaire d'erreurs global, comme les erreurs fatales.

Question 5

Parmi les propositions suivantes, trouvez celle qui est correcte.

- ☐ La politique de gestion des erreurs doit être la même en environnement de développement qu'en environnement de production
- ☒ La politique de gestion des erreurs doit être adaptée à l'environnement d'exécution
- ☐ PHP gère automatiquement pour nous la gestion des erreurs en fonction de l'environnement d'exécution

Q PHP ne gère pas pour nous la politique de gestion des erreurs. Il nous fournit les moyens de la mettre en place : en configurant un gestionnaire d'erreurs global avec `set_error_handler` et en paramétrant sa sensibilité avec des directives à renseigner dans le fichier de configuration `php.ini`. La politique de gestion des erreurs doit être adaptée à l'environnement d'exécution. Par exemple, en environnement de développement, on pourra afficher toutes les erreurs sur la sortie standard et les enregistrer dans un *log*. En production, on les enregistre aussi dans un *log*, mais on ne les affichera surtout pas à l'utilisateur !

Exercice p. 17 Solution n°2

Question 1

Lorsqu'une Exception est lancée, elle...

- ☒ Suspend immédiatement le flot d'exécution du programme et remonte la pile d'appels jusqu'à être attrapée
- ☐ Génère automatiquement une erreur
- ☐ Arrête l'exécution du programme immédiatement

Q Lorsqu'une **Exception** est lancée, le flot d'exécution du programme est suspendu, et l'**Exception** remonte la pile d'appels (*bubbling*) jusqu'à rencontrer un bloc `try/catch` ou le gestionnaire d'exceptions global. Si elle est récupérée par un bloc `try/catch`, l'exécution reprend après le bloc attrapant. Si elle n'est pas attrapée, elle génère une **Fatal Error**, ce qui stoppe l'exécution du programme brutalement.

Question 2

Une Exception se caractérise au moins par...

- ☒ Un type (**Exception** ou classe étendant la classe **Exception**), un message, un code et une *stack trace*
- ☐ Un code uniquement
- ☐ Un message uniquement

Q Une **Exception** apporte du contexte à une erreur rencontrée par le programme. Elle est déjà définie par un type **Exception** fourni par PHP, qui peut être étendu pour créer ses propres exceptions. Comme une **Error**, une exception est définie par un message, un code et une *stack trace*.

Question 3

Quelle est la sortie produite par le code suivant ?


```

1 <?php
2 try {
3     throw new PDOException();
4 } catch (Exception $e) {
5     echo 'Exception attrapée. Traitement.' . PHP_EOL;
6 } catch (PDOException $e) {
7     echo 'PDOException attrapée. Traitement.' . PHP_EOL;
8 } finally {
9     echo 'On execute toujours ce bloc de code, peu importe l\'exception attrapée.' . PHP_EOL;
10 }

```

- ☐ PDOException attrapée. Traitement.
On execute toujours ce bloc de code, peu importe l\'exception attrapée.
- ☐ PDOException attrapée. Traitement.
Exception attrapée. Traitement.
On execute toujours ce bloc de code, peu importe l\'exception attrapée.
- ☒ Exception attrapée. Traitement.
On execute toujours ce bloc de code, peu importe l\'exception attrapée.

Q Dans ce code, on lance une exception de type **PDOException**, une classe étendant la classe **Exception** fournie par PHP. L'exception va être attrapée par le premier bloc **catch** qu'elle rencontre avec le type correspondant : comme **PDOException** est un sous-type du type **Exception**, elle est par définition de type **Exception**. Elle sera donc attrapée par le premier bloc **catch**. Enfin, le bloc **finally** est toujours exécuté.

Question 4

À quoi sert un gestionnaire d'exceptions global ?

- ☒ À attraper automatiquement les exceptions qui, en remontant la pile d'appels, n'ont rencontré aucun bloc **try/catch**
- ☐ À rien, car toutes les exceptions sont attrapées dans des blocs **try/catch**
- ☐ À rien si on a déjà défini un gestionnaire d'erreurs global : les deux font exactement la même chose

Q Un gestionnaire d'exceptions global est le dernier rempart contre une exception qui remonte la pile et n'est pas attrapée en cours de route. Il est fort probable qu'en développant votre application et en utilisant des bibliothèques tierces, vous oubliez de gérer toutes les possibilités d'émission d'**Exception**. Un gestionnaire global d'exception doit donc toujours être défini dans votre code ! Enfin, le gestionnaire global d'exception gère les exceptions, pas les erreurs ! Chaque gestionnaire a son rôle, et les deux se complètent dans votre politique de gestion d'erreurs.

Question 5

Qu'est-ce que la programmation défensive ?

- ☐ Émettre des exceptions quand votre programme rencontre une situation qu'il ne peut pas traiter avec certitude
- ☒ Attraper les exceptions émises par votre code ou celui d'autres personnes pour être sûr de parer aux pires situations
- ☐ Tester votre code en amont

Q Lancer des exceptions est de la programmation offensive : vous "attaquez" le code qui utilise le vôtre pour lui signifier qu'il doit prendre en compte des situations imprévues. La programmation défensive est à la fois une méthode de conception et un état d'esprit : c'est concevoir son programme en s'attendant toujours au pire au

Q moment de l'exécution, et non en amont. Le programme doit pouvoir réagir convenablement à des situations imprévues. Une application de la programmation défensive, notamment en PHP, consiste donc à attraper toutes les exceptions susceptibles d'être émises dans des situations imprévues, soit avec des blocs `try/catch`, soit avec un gestionnaire d'exceptions global.

p. 19 Solution n°3

```

1 /**
2  *Rapporte les abstract invalides
3  *@param array $abstracts Une liste d'abstracts non filtrés et potentiellement invalides
4  *@return array Le rapport: liste des abstracts valides, informations sur les abstracts
   invalides
5  */
6 function reportAbstracts(array $abstracts): array
7 {
8
9     $validAbstracts = [ ];
10
11     $invalidAbstracts = [ ];
12     foreach ($abstracts as $abstract) {
13         try {
14             $validAbstracts[] = validAbstract($abstract);
15         } catch (DomainException $e) {
16             $invalidAbstracts[] = $abstract['id'];
17         }
18     }
19
20     $report = sprintf("Nombre d'abstracts valides : %d/%d\n", count($invalidAbstracts),
21                       count($abstracts));
22     $report .= sprintf("Liste des abstracts invalides (id): %s\n", implode(' ',
23                               $invalidAbstracts));
24
25     return array(
26         'valid' => $validAbstracts,
27         'invalid' => $invalidAbstracts,
28         'summary' => $report
29     );
30 }
31 /**
32  Retourne l'abstract s'il est valide
33
34  *@throws DomainException Si l'abstract n'est pas valide
35  */
36 function validAbstract(array $abstract): array
37 {
38     if (!isset($abstract['id']) || !isset($abstract['name']) || !isset($abstract['url'])) {
39         throw new DomainException("L'abstract n'a pas un format valide");
40     }
41
42     if ($abstract['name'] === null) {
43         throw new DomainException("'Nom Indéfini");
44     }
45
46     if (!str_contains($abstract['url'], 'http://')) {
47         throw new DomainException("'URL invalide");
48     }
49 }

```

```

49
50 return $abstract;
51 }
52
53 $result = reportAbstracts($abstracts);
54 echo $result['summary'];

```

p. 19 Solution n°4

```

1 <?php
2
3 $abstracts = [
4 [
5 'id' => 1,
6 'name' => 'git',
7 'url' => 'http://git.test/abstract'
8 ],
9 [
10 'id' => 2,
11 'name' => null,
12 'url' => 'http://php.test/abstract'
13 ],
14 [
15 'id' => 3,
16 'name' => 'handling-errors-best-practices',
17 'url' => 'http://programming.test/abstract/handling-errors/best-practices'
18 ],
19 [
20 'id' => 4,
21 'name' => 'geocities',
22 'url' => 'geocities.test/abstract'
23 ],
24 ];
25
26 function reportAbstracts(array $abstracts): array
27 {
28     $validAbstracts = [];
29     $invalidAbstracts = [];
30     foreach ($abstracts as $abstract) {
31         try {
32             $validAbstracts[] = validAbstract($abstract);
33         } catch (DomainException $e) {
34             $invalidAbstracts[] = $abstract['id'];
35         }
36     }
37
38     $report = sprintf("Nombre d'abstracts valides : %d/%d\n", count($validAbstracts),
39         count($abstracts));
40     $report .= sprintf("Liste des abstracts invalides (id): %s<br>", implode(',',
41         $invalidAbstracts));
42
43     return array(
44         'valid' => $validAbstracts,
45         'invalid' => $invalidAbstracts,
46         'summary' => $report
47     );
48 }

```

```

47 $needle = 'http://';
48 $url = $abstract[url];
49 function validAbstract(array $abstract): array
50 {
51     $needle = 'http://';
52     if (!isset($abstract['id']) || !isset($abstract['name']) || !isset($abstract['url'])) {
53         throw new DomainException("L'abstract n'a pas un format valide");
54     }
55     if ($abstract['name'] === null) {
56         throw new DomainException("Nom Indéfini");
57     }
58     if (strpos($abstract['url'], $needle) === false) {
59         throw new DomainException("URL invalide");
60     }
61     return $abstract;
62 }
63
64 $result = reportAbstracts($abstracts);
65 echo $result['summary'];
66
67 set_exception_handler(function (Exception $e) {
68     $log = sprintf(
69         "%s %s %s %s %s",
70         date('Y-m-d- h:m:s'),
71         $e->getMessage(),
72         $e->getCode(),
73         $e->getFile(),
74         $e->getLine()
75     );
76     echo $log;
77     error_log($log, 3, 'error.log');
78 });
79 throw new Exception('Oups !', 400);

```

Exercice p. 20 Solution n°5


Question 1

Le fichier mylib.php n'existe pas dans le dossier courant du script. Quelle est la sortie du code suivant ?

```

1 <?
2 require 'mylib.php';
3 echo 'mylib loaded ! ' . PHP_EOL;

```

- ☒ PHP Warning: require(mylib.php): Failed to open stream: No such file or directory in index.php on line 1
PHP Fatal error: Uncaught Error: Failed opening required 'mylib.php' (include_path=.:usr/share/php) in index.php:1
- ☐ mylib loaded !
- ☐ PHP Warning: require(mylib.php): Failed to open stream: No such file or directory in index.php on line 1
mylib loaded !
-  Le *construct* **require** permet d'inclure le contenu d'un script PHP dans un autre. On l'a vu, **require** émet un warning suivi d'une erreur fatale lorsque le fichier à **include** dans le script courant n'est pas accessible. Le *construct* **include** n'aurait émis qu'un warning et l'exécution se serait poursuivie (vers une fin certainement malheureuse !).

Question 2

PHP nous permet de définir nos propres exceptions.


- ☒ Vrai
- ☐ Faux
- ☐ Oui, on peut définir nos propres exceptions en étendant la classe `Exception` fournie par PHP.

Question 3

Quelle est la sortie du code suivant ?

```
1 <?php
2 /**
3  * Retourne l'âge de la personne s'il est censé
4  * @throws DomainException Si l'âge a une valeur insensée
5  */
6 function acceptedAge(int $age): int
7 {
8     if ($age < 0 || $age > 150)
9         throw new DomainException("L'âge n'est pas valide !");
10    return $age;
11 }
12
13 try {
14     acceptedAge(200);
15     acceptedAge(12);
16 } catch (Exception $e) {
17     echo $e->getMessage() . PHP_EOL;
18 } catch (DomainException $e) {
19     echo 'Erreur dans le domaine : ' . $e->getMessage() . PHP_EOL;
20 }
```

- ☒ L'âge n'est pas valide !
- ☐ Erreur dans le domaine : L'âge n'est pas valide !
- ☐ L'âge n'est pas valide !
12

 : La bonne réponse est la première. Le premier appel à `acceptedAge` va lancer une exception, car l'âge a une valeur insensée pour un être humain (erreur métier). L'exception est de type `DomainException`, qui est un sous-type (classe enfant) de `Exception`. Donc `DomainException` est par définition de type `Exception` et sera attrapée par le premier bloc `catch`. L'exécution reprendra donc après le bloc `catch`, et le deuxième appel à `acceptedAge` ne sera jamais effectué.

Question 4

Une Exception qui n'est pas attrapée remonte la pile d'appels jusque dans le contexte global, et finit par générer une Fatal Error.

- ☒ Vrai
- ☐ Faux
- ☐ Vrai. L'exception, si elle n'est pas attrapée par un bloc `catch` ou par le gestionnaire d'exceptions global, finit par provoquer une **Fatal Error** et une interruption du programme. D'où l'importance de toujours déclarer un gestionnaire d'exceptions global.

Question 5

Il est impossible de récupérer (poursuivre l'exécution du programme) après une Fatal Error (erreur fatale).

☐ Vrai

☒ Faux

Q Faux. Vous pouvez continuer l'exécution en plaçant du code dans votre gestionnaire d'erreurs global, et relancer tout un traitement. Cela dit, ce n'est pas parce que vous pouvez le faire que vous devez le faire : une erreur fatale est émise par PHP pour vous indiquer que l'exécution de votre programme ne peut plus être sûre et que son bon fonctionnement n'est plus garanti. On vous invite donc à arrêter avant qu'il ne soit trop tard. Vous devriez plutôt chercher à faire atterrir le programme en douceur (échouer avec grâce).