Exceptions et PHP5

par Guillaume Affringue (http://guillaume-affringue.developpez.com)

Date de publication : 13/09/2006

Dernière mise à jour : 24/01/2007

Ce cours a la prétention de vous apprendre comment utiliser les exceptions avec PHP5 et quelle est leur place naturelle au sein d'un script, notamment par rapport au système de gestion d'erreur de PHP

- Présentation

I - Présentation

Les exceptions sont pour beaucoup de personnes un système élaboré de gestion d'erreur. Ce n'est pas le cas. Et même si dans la pratique, on peut, comme nous le verrons, établir des liens entre ces deux concepts, leurs buts sont totalement différents. Une erreur est dans la majorité des cas assimilable à un bug, c'est à dire une réaction non désirée. Une exception est un traitement qui sera appelé lorsqu'un cas particulier est détecté pendant le déroulement du programme.

Pour synthétiser, une erreur est le résultat un fonctionnement anormal alors qu'une exception est un fonctionnement normal, mais exceptionnel.

I-A - Mots-clés [throw] [try] [catch]

Comme tous les mots-clés de programmation, ces trois mots proviennent de l'anglais et signifient

- **throw** : lancer
- **try** : essayer
- **catch** : attraper

La signification de ces 3 mots-clé est très représentative du mécanisme utilisé et doit toujours être gardé à l'esprit. En effet, les exceptions fonctionnent de la même façon dans tous les langages. Dans un bloc défini (**try**), l'environnement d'exécution envoie un signal (**throw**) au gestionnaire d'exception qui diffusera l'information aux différents blocs de traitement (catch). Le bloc adéquat au signal émis attrappera alors le signal (**catch**) et exécutera un code adapté.

La mise en place d'une exception se présente comme ceci

Mécanisme simple d'exception

```
try
{
    throw new Exception ('ceci est faux');
    echo 'Code non executé';
}
catch(Exception $myException)
{
    echo $myException->getMessage();
}
```

Le déroulement de ce code est le suivant

1 On entre dans le bloc try
2 On lance une exception (on dit aussi "soulever une exception").
3 On quitte le bloc try (sans exécuter le code restant du bloc try)
4 Le catch attrape l'exception
5 On execute le contenu du bloc catch

Le code précédent affichera ainsi :

Affichage

Affichage
```
Ceci est faux
```

Il s'agit ici du cas le plus simple car l'exception est au même niveau que le bloc try/catch et nous n'avons traité qu'un cas d'exception, en utilisant la classe native de PHP. Nous allons voir qu'il est possible de personnaliser ce mécanisme par l'écriture de classe d'Exception propre à un contexte, et comment utiliser au mieux ce mécanisme, en comprennant bien le rôle et la place du throw et des blocs try/catch

I-B - Profondeur de throw

Un bloc try peut ne pas contenir directement de throw. Les throw sont en effet répartis dans le code en fonction des besoins. On peut ainsi lancer une exception dans une fonction, et placer l'appel à cette fonction dans un bloc try.

Utilisation des exceptions dans une fonction
```php
function verif_text( $text )
{
    if (empty($text))
    {
        throw new Exception ('Le texte est vide');
    }
    else
    {
        return $text;
    }
}


try
{
    verif_text( '' );
}
catch(Exception $myException)
{
    echo $myException->getMessage();
}
```

Cet exemple lancera bien l'exception et affichera le message 'Le texte est vide'.

Le principe est exactement le même dans les méthodes d'une classe.

On a vu au I-A que le code suivant un throw dans un bloc try n'était pas éxecuté. Il en est de même ici, même à l'intérieur de la fonction. Ainsi, ceci :

Exemple d'interruption de fonction par une exception
```php
function myFct()
{
    throw new Exception("Voici l'exception");
    echo "Une exception vient d'être levée";
}


try
{
    myFct();
}
catch(Exception $e)
{
    echo $e->getMessage();
```

Exemple d'interruption de fonction par une exception

```
    }
```

affichera uniquement

Affichage

```
Voici l'exception
```

Il est donc inutile de mettre un die, exit ou return directement après un throw, la fonction est de toute manière interrompue.

L'intéret est de distinguer les instructions throw et try/catch. L'écriture d'une classe ou d'une fonction ne devrait contenir que des throw, et rarement de try/catch que l'on utilise plus lors des appels de ces classes/fonctions.

I-C - Intérêt des exceptions

L'intérêt des exceptions par rapport aux erreurs est multiple

- Pouvoir interrompre un bloc de code si on detecte la moindre exception.
- Pouvoir générer des exceptions pour les erreurs systèmes, mais aussi pour les erreurs utilisateurs ou tout autre évènement défini par le développeur.
- Personnaliser le traitement de l'exception dans le bloc catch et ainsi appliquer un correctif ou une alerte adapté.

On pourrait alors se dire que les exceptions ne sont qu'un système d'erreur évolué. Pas du tout ! Il faut bien distinguer les erreurs des exceptions. Une erreur est en général une réponse du système à problème rencontré. Les exceptions sont des alertes dont l'emplacement et le comportement sont définis par le développeur. Les exceptions permettent de gérer les erreurs, la réciproque n'a pas de sens.

Les exceptions répondent à un besoin, qui est de laisser la gestion des erreurs utilisateurs aux mécanismes adaptés pour ceci. Vous pensez que c'est ce que vous faites? Qui n'a jamais écrit une fonction de ce type?

Exemple de fonction renvoyant valeur et erreur

```
<?php
function testLogin( $login )
{
    if ( ! ($res = mysql_query("SELECT 1 FROM membres WHERE
 login='".mysql_real_escape_string($login)."'")) )
    {
        return FALSE;
    }
    else
    {
        $row = mysql_fetch_assoc($res);
        if ($row['login'] == $login)
        {
            return TRUE;
        }
        else
        {
            return FALSE;
        }
    }
}
?>
```

Ceci est une horreur. Que peut-on conclure si la fonction retourne un false ? Absolument rien, peut-être y a-t-il eu une erreur, peut-être pas...

Par définition, une fonction doit toujours renvoyer une valeur attendue et jamais une erreur (et encore moins les deux en même temps). Imaginez votre voiture, vous mettez de l'essence en entrée, vous obtenez une vitesse en sortie. Lorsqu'un problème intervient, avec le système incorrect ci-dessus, vous obtienteriez que votre vitesse est nulle, la voiture ne fonctionne plus, et c'est tout. Or, le problème en question n'est qu'un feu stop cassé. Avec les exceptions, la voiture continue de rouler, vous lisez la vitesse sur le compteur, et un petit voyant s'est allumé sur le tableau de bord indiquant qu'un feu stop est cassé.

En renvoyant erreurs et valeurs, des collisions peuvent se produire car false == 0

Il faudra donc typer les comparaisons pour distinguer les erreurs des valeurs

Par exemple

```
if ( countMembre() === false ) { // c'est une erreur...

if ( countMembre() === 0) { // Il y a 0 membre...
```

II - Héritage de la classe d'Exception

La classe d'exception que nous avons utilisée jusqu'ici est la classe native de PHP5. Nous allons maintenant définir nos propres classes d'exception adaptées à nos besoins, héritant de cette classe mère. Nous allons commencer par étudier les propriétés de la classe native puis comment créer et/ou utiliser les classes filles.

II-A - Méthode et attributs de la classe Exception

II-A-1 - Attributs

La classe Exception possède 4 attributs protected (ce qui nous permet d'y avoir accès depuis une classe fille).

Attributs de la classe Exception

```
protected $message = 'exception inconnu'; // message
protected $code = 0;      // code erreur défini par l'utilisateur
protected $file;       // nom du fichier source de l'exception
protected $line;       // ligne de la source de l'exception
```

II-A-2 - Méthodes

La classe possède un seul constructeur :

Constructeur de la classe Exception

```
function __construct(string $message=NULL, int code=0);
```

Ce constructeur est le seul moyen d'affecter des valeurs aux attributs $message et $code. Ces deux attributs doivent donc naturellement être définis par l'utilisateur. Pour instancier la classe et ainsi définir les valeurs de $message et $code, on utilise l'instruction throw vue ci-dessus.

La classe définit ensuite plusieurs accesseurs à ces attributs.

Méthodes de la classe Exception

```
final function getMessage();             // message de l'exception
final function getCode();               // code de l'exception
final function getFile();               // nom du fichier source
final function getLine();               // ligne du fichier source
final function getTrace();               // un tableau de backtrace()
final function getTraceAsString();       // chaîne formattée de trace
```

Les 4 premières méthodes renvoient les attributs respectifs. La méthode getTrace renvoie un tableau de toutes les exceptions lancées. En effet, comme vous choisissez vous même le comportement d'un traitement d'exception, vous n'êtes pas obligé d'interrompre le script comme le ferait une erreur PHP.

Le tableau renvoyé est de la forme

- **line** - La ligne où s'est déroulée l'erreur
- **file** - Le fichier où s'est déroulée l'erreur
- **function** - La fonction concernée
- **class** - La classe concernée

- **type** - Type d'appel de la méthode (statique ou dynamique)
- **args** - Les arguments de la méthode

Enfin, la denière méthode renvoie, comme la précédente, l'ensemble des exceptions lancées, mais sous forme de string.

Il faut noter que ces fonctions sont finales, elles ne peuvent donc pas être surchargées dans une classe fille.

II-B - Exemple d'héritage

La classe Exception de PHP5 est utilisable dans sa forme native, mais le but des exceptions est d'effectuer un traitement adapté en réponse à une alerte particulière. PHP5 permet donc d'étendre cette classe par héritage.

Voyons un exemple :

Exemple de classe fille de la classe Exception

```php
<?php
class MyException extends Exception
{
    public function __construct($msg=NULL, $code=0)
    {
        parent::__construct($msg, $code);
    }

    public function addLog()
    {
        // Ajoute un log dans la table
        mysql_query("INSERT INTO log (text) VALUES ('".$this->getMessage()."')");
    }

    public function showError()
    {
        return '<div style="color:red">'.$this->getMessage().'</div>';
    }
}


try
{
    throw new MyException('ceci est faux');
}
catch(MyException $myException)
{
    $myException->addLog();
    $myException->showError();
}
?>
```

Ici, on utilise la classe MyException, qui va insérer un log dans une table, puis afficher un message formaté.

On voit donc qu'il est possible de créer une gestion complète en fonction du type d'exception soulevée. Les classes d'Exception permettent de travailler sur un contexte plutôt que sur un script complet. Ainsi, dans un même script, on peut avoir une classe d'exception pour les erreurs liées au SGBD, une classe pour les erreurs XML, une classe pour les erreurs système, une classe pour les erreurs utilisateur.....

Cascade d'exceptions

Cascade d'exceptions

```php
try
{
    if ( $x )
    {
        throw new MyException('ceci est faux');
    }
    $xml = new DomDocument();
    $sql = new SQLiteDatabase( $filename );
}
catch(MyException $myException)
{
    $myException->addLog();
    $myException->showError();
}
catch(DOMException $myDOMException)
{
    echo 'Erreur XML';
}
catch(SQLiteException $mySQLIteException)
{
    echo 'erreur SQL';
}
```

Si x == false, nous lancerons l'exception MyException comme nous l'avons défini manuellement. Mais nous pouvons aussi attraper 2 exceptions que nous ne connaissons pas. Ces exceptions sont définies respectivement dans les extensions DOM et SQLITE. En effet, certaines extensions possèdent leur propre classe d'Exception, facilitant le travail du développeur en lui évitant de se demander comment accéder aux données à afficher en cas d'erreur. Ceci reflète ce qui a été dit au I-B, c'est-à-dire que les fonctions et classes lancent elles-mêmes leurs exceptions, qui sont attrapées par le développeur dans le code plutôt que dans les fonctions/classes elles-mêmes.

III - Exceptions ou erreurs ?

Dans ce paragraphe, nous allons comparer les 2 systèmes (exceptions/erreurs). En fait, nous allons nous concentrer sur les exceptions qui est le sujet principal, car on remarquera vite que les inconvénients de l'un sont les avantages de l'autre et réciproquement.

III-A - Avantages des exceptions

III-A-1 - Lisibilité

Comme je l'ai déjà écrit plusieurs fois, les exceptions ne sont pas identiques aux erreurs dans le sens ou elles représentent un fonctionnement normal, mais exceptionnel. Nous allons maintenant voir un exemple dans lequel les exceptions nous aident grandement et montrent clairement que le mécanisme de gestion d'erreurs n'a rien à voir là-dedans. Considérons une page de traitement de formulaire d'inscription à un site. Nous avons le pseudo, le mot de passe et la confirmation ainsi que l'adresse mail qui arrivent du formulaire. Il faudra vérifier que le pseudo n'existe pas, que le mot de passe et la confirmation sont identiques et que le mail est correct.

On pourrait y arriver avec une avalanche de if, mais ça deviendrait vite suffisamment compliqué pour qu'une erreur s'y glisse. Par exemple :

Page d'inscription utilisant une avalanche de if

```
$result = mysql_query("
    SELECT 1 FROM membres
    WHERE login='".mysql_real_escape_string($_POST['login'])."'");

if (mysql_num_rows($result) == 0)
{
    // le pseudo n'est pas pris, on continue
    if ($_POST['pass'] == $_POST['confirmation'])
    {
        // Le mot de passe et la confirmation coïncident
        if (isValide($_POST['mail']))
        {
            // Le mail est valide, on insère le mebre
            insere_membre($_POST['login'], $_POST['pass'], $_POST['mail']);
        }
        else
        {
            affiche_formulaire('mail_invalide');
        }
    }
    // Le mot de passe et la confirmation ne coïncident pas
    else
    {
        affiche_formulaire('error_mdp');
    }
}
// Le pseudo est pris, on affiche le formulaire
else
{
    affiche_formulaire('error_login');
}
```

Imaginez si on doit effectuer une vérification sur 200 champs de fomulaire. C'est sincèrement ingérable à cause de la profondeur des if.

Grâce aux exceptions, on ne s'enfonce plus dans une telle profondeur de if, et on n'appelle plus qu'une seule fois la fonction affiche_formulaire. Ainsi, le code précédent devient :

Page d'inscription utilisant les exceptions

```
try
{
    $result = mysql_query("
        SELECT 1 FROM membres
        WHERE login='".mysql_real_escape_string($_POST['login'])."'");

    if (mysql_num_rows($result) > 0)
    {
        throw new Exception('error_login');
    }

    if ($_POST['pass'] != $_POST['confirmation'])
    {
        throw new Exception('error_mdp');
    }

    if ( ! isValide($_POST['mail']))
    {
        throw new Exception('mail_invalide');
    }
}
catch(Exception $myException)
{
    affiche_formulaire($myException->getMessage());
}
```

III-A-2 - Fiabilité

Les exceptions permettent de ne pas stopper net un système ayant généré une erreur. Ceci permet d'effectuer un traitement adapté au problème rencontré, voire parfois de le réparer. Ainsi, certains langages comme l'ADA ont intégré entièrement les exceptions dans leur langage pour gérer les erreurs liées au système. C'est pourquoi l'ADA est très utilisé en milieu militaire et aérospatial. En effet, même dans ces systèmes coutânt des milliards, il y a des bugs, des erreurs... Il vaut donc mieux être sûr que même en cas d'erreur, le système ne va pas simplement s'arréter. Java, bien que possédant une gestion des erreurs, est très orienté exception, ce qui a contribué (entre autre) à son succès dans le monde industriel. Une classe Java annonce quelles exceptions elle lance. Ainsi, un code source Java ne traitant pas ces exceptions n'est simplement pas compilé. Tous les autres langages utilisent soit les erreurs uniquement, soit un répartition équilibrée entre exception et erreur.

Evidemment, pour augmenter réellement la fiabilité des systèmes, il est important de bien réfléchir à la cohérence des classes d'exception et de bien les capturer. Sinon, le système ne correspondra qu'à une gestion des erreurs traditionnelles.

III-B - Inconvénients des Exceptions

III-B-1 - Implémentation "partielle"

Dans PHP5, le mécanisme d'exception est totalement opérationnel, mais ne constitue qu'un apport pour le développeur et non pour le langage. C'est à dire que PHP5 n'utilise pas les exceptions. L'ensemble des fonctions de PHP génèrent toujours des erreurs PHP lorsqu'elles rencontrent un problème, seules quelques extensions ont adopté le modèle des exceptions. Cette faible présence des exceptions dans PHP5 s'explique par :

- La programmation objet en PHP n'a de réel sens que depuis PHP5 or, pour utiliser au mieux les exceptions, il est nécessaire pour un langage de possèder un bon modèle objet.
- Lorsqu'une exception est soulevée, le contexte local contenu dans un bloc try est sauvegardé, ceci peut être amplifié par l'imbrication de bloc try. L'abus de l'utilisation des exceptions peut donc diminuer considérablement les performances.
- Par choix. La plupart des langages choisissent de ne pas supprimer la gestion des erreurs au profit des exceptions pour des raisons de performances et de style de programmation (les exceptions sont équivalentes pour certains programmateurs à l'affreux goto, de plus les exceptions augmentent considérablement le nombre de lignes de code). D'autres langages, comme l'ADA, n'utilisent que les exceptions.

III-B-2 - Boucle infinie

Les exceptions permettent un traitement personnalisé d'une erreur. Imaginons un système d'exception unique pour tout un script, que le développeur utilise pour fiabiliser l'accès à un fichier, la connexion à une BDD... Dans cette classe d'exception, il décide de logguer toutes les exceptions soulevées afin de garder une trace. Pour cela, il utilise une BDD. Si la première connexion à la base de donnée ne se fait pas, on soulève une exception, qui tentera de se connecter à la base de donnée, ce qui soulèvera une exception... On obtient des comportements plutôt imprévus, pouvant aller jusqu'à la boucle infinie si le script implémente la fonction set_exception_handler. Il est donc très important d'éviter ce genre de scénario. La meilleure méthode pour éviter ceci est de bien distinguer les différentes classes d'exception dont le script a besoin. En créant ici une classe d'Exception spécifique à la base de données, on aurait forcement compris que l'on ne peut pas utiliser la base de données, vu que l'appel même de l'exception signifie que la connexion à la base de données a échoué. On utilisera alors un fichier. De même, on logguera les exceptions soulevées lors d'un accès fichier dans la BDD et non dans un fichier.

En réalité, PHP empêche ceci. Si il détecte une exception lancée à l'intérieur du traitement d'une autre exception (que ce soit avec catch ou set_exception_handler), il stoppe l'éxecution et affichera un message d'erreur. Ainsi, le code suivant :

Exception lancée dans le code de traitement d'une autre exception

```php
<?php

class MyException extends Exception
{
    public function __construct($msg=null, $code=0)
    {
        parent::__construct($msg, $code);
    }

    // 3 ) La fonction addLog va essayer de se connecter à MySQL, or cette fonction est
    //  justement appelée car une précédente tentative de connexion a échouée.
    public function addLog()
    {
        // La tentative échoue encore
        if ( ! @mysql_connect($server, $user, $pass) )
        {
            // 4) On lance une exception non capturée (on pourait la capturer,
            //  l'erreur serait identique)
            throw new MyException('Impossible de se connecter à MySQL dans la classe MyException');
        }
    }
}

// 5 ) On attrape l'exception lancée dans la méthode addLog
//      et on ressaie d'insérer un log...
//    On voit ici la boucle infinie qui se crée entre l'appel de
//      cette fonction exception_handler et la méthode addLog.
function exception_handler($myException)
{
```

Exception lancée dans le code de traitement d'une autre exception

```
        $myException->addLog();
    }

    set_exception_handler('exception_handler');

    // 1) Le commencement, on ouvre une connexion à MySQL,
            qui bien sûr ne va pas fonctionner
    try
    {
        if ( ! @mysql_connect($server, $user, $pass) )
        {
            // On lance donc l'exception
            throw new MyException('Impossible de se connecter à Mysql dans le script');
        }
    }
    // 2) On capture l'exception et on applique le traitement,
    //      c'est à dire insérer un log
    catch(MyException $e)
    {
        $e->addLog();
    }


    ?>
```

Affichera l'erreur suivante.

Erreur générée

```
Fatal error: Exception thrown without a stack frame in Unknown on line 0
```

Pour résumer : il faut bien être conscient de la cause d'une exception, de façon à ne pas appeler de nouveau dans son traitement la portion de code responsable de sa levée. Le mieux est d'établir une hiérarchie des classes d'exception. Par exemple, si on lance une exception dans le script, le traitement pourra tenter d'écrire dans la base de données, s'il n'y parvient pas, il lance une exception qui tentera d'écrire dans un fichier. Si ce traitement est aussi un échec, une nouvelle exception tentera simplement d'afficher à l'ecran, si rien ne fonctionne, on lance une erreur simple, ou on ne fait rien du tout. Le meilleur moyen d'éviter cette erreur consiste à limiter les opérations à risque dans le code de traitement des exceptions.

IV - Amélioration du système d'exception

IV-A - Capture automatique des exceptions

La présence de la gestion des exceptions dans certaines extensions de PHP, généralement non utilisée car non connue des développeurs, ainsi que les possibles risques d'avoir un throw égaré dans les profondeurs d'une classe impliquent que certaines exceptions soulevées peuvent ne pas être attrapées. Nous obtenons alors un message peu agréable de la sorte :

```
Fatal error: Uncaught exception 'Exception' with message 'Message non capturé'
in C:\wamp\www\exception\index.php:4
Stack trace: #0 C:\wamp\www\exception\index.php(7): showUncaughtException() #1 {main}
thrown in C:\wamp\www\exception\index.php on line 4
```

Dans le cas de PDO par exemple, une exception non gérée affiche toute la chaîne de connexion incluant les **données de connexion** à la base de données.

```
Exception de connexion à la BDD avec PDO a écrit :
Fatal error: Uncaught exception 'PDOException' with message 'SQLSTATE[42000] [1049]
Unknown database 'jeu_de_role'' in
c:\program files\apache group\www\dvp\tests\pdo.php:22
Stack trace: #0 c:\program files\apache group\www\dvp\tests\pdo.php(22):
PDO->__construct('mysql:host=loca...', 'root', '1234', Array) #1 {main}
thrown in c:\program files\apache group\www\dvp\tests\pdo.php on line 22
```

Pour remédier à ce problème évident, PHP définit la fonction set_exception_handler permettant de définir une fonction de callback appelée à chaque exception soulevée et non capturée

Exemple d'utilisation de la fonction set_exception_handler

```php
function exception_handler($myException)
{
    echo 'Exception non capturée : '.$myException->getMessage();
}

set_exception_handler('exception_handler');

throw new Exception('voici une exception');
// Affichera l'exception non capturé : voici une exception
```

IV-B - Bascule erreur->exception

Comme nous l'avons vu précédemment, PHP5 n'utilise pas les exceptions, il se contente de les proposer au développeur. Parallèlement, PHP permet de redéfinir le support des erreurs système. Merveilleux, nous avons tout pour faire notre propre gestion des erreurs en utilisant les exceptions.

Exemple de redéfintion de la gestion d'erreur

```php
function exception_handler($code, $msg, $file, $line)
{
    throw new Exception($msg, $code);
}

set_error_handler('exception_handler');
```

Exemple de redéfintion de la gestion d'erreur

```
try
{
    fopen();
}
catch (Exception $myException)
{
    echo '<div style="color:red">'.$myException->getMessage().'</div>';
}
```

Cet exemple est trop simple pour être utilisable tel quel. Il retire beaucoup d'informations par rapport à ce qu'apporterait une erreur. Pour l'étendre, nous pouvons d'une part créer une classe fille MyPHPException, chargée de ne traiter que ce type d'erreur et nous laissant libre d'étendre nos exceptions ailleurs, d'autre part, nous pouvons ajouter les informations de l'erreur dans l'exception. Par exemple, nous pouvons obtenir la ligne et le fichier dans lequel s'est produite l'erreur, ainsi que le contexte à ce stade du script. Nous pouvons ajouter aussi des informations apportées par l'exception. En effet, nous avons aussi accès au contexte lors de la levée d'exception. Ainsi, nous pouvons connaître la dernière fonction appelée, dans laquelle s'est déroulée l'erreur. Nous avons aussi accès aux tableau super globaux et aux variables locales.

Voici notre nouveau script amélioré :

Gestion complète des erreurs par les exceptions

```
/**
* @desc Notre classe d'exception pour les erreurs PHP
*/
class MyPHPException extends Exception
{
    /**
    * @desc Constructeur
    */
    public function __construct($msg, $code, $file, $line, $context)
    {
        $this->message = $msg;
        $this->code = $code;
        $this->line = $line;
        $this->file = $file;
        $this->context = $context;

        parent::__construct($msg, $code);
    }

    /**
    * @desc Affichage de l'erreur
    */
    public function showError()
    {
        echo 'PHP a généré l\'erreur système suivante : ['.$this->code.' | '.
            $this->getMessage().'] à la ligne '.
            $this->line.' du fichier '.$this->file;

        // $Mytrace contient le contexte de l'exception
        // $this->context contient le contexte de l'erreur

        $Mytrace = $this->getTrace();
        //print_r($Mytrace);
        if ( ! empty($Mytrace['1']['function'] ))
        {
            echo ' sur la fonction '.$Mytrace['1']['function'];
        }
        echo '<br /><br/>Contexte lors de l\'erreur :<br/><pre>';
        print_r($this->context);
        echo '</pre>';
```

Gestion complète des erreurs par les exceptions

```
        }
    }

    /**
    * @desc la fonction de callback, chargée de lancer l'exception
    */
    function errorToException($code, $msg, $file, $line, $context)
    {
        throw new MyPHPException($msg, $code, $file, $line, $context);
    }

    // redéfinition de la gestion d'erreur
    set_error_handler('errorToException');

    // Enfin, notre script
    try
    {
        fopen();
    }
    catch (MyPHPException $myPHPException)
    {
        echo '<div style="color:red">'.$myPHPException->showError().'</div>';
    }
```

Ceci nous affichera à l'écran :

```
PHP a généré l'erreur système suivante : [2 | fopen() expects at least 2 parameters, 0 given]
à la ligne 44 du fichier .C:\wamp\www\exceptions\index.php sur la fonction fopen

Contexte lors de l'erreur :

Array
(
    [GLOBALS] => Array
 *RECURSION*
    [_ENV] => Array
        (
            [ALLUSERSPROFILE] => C:\Documents and Settings\All Users
            [CommonProgramFiles] => C:\Program Files\Fichiers communs
            [ComSpec] => C:\WINDOWS\system32\cmd.exe
            [FP_NO_HOST_CHECK] => NO
            [NUMBER_OF_PROCESSORS] => 1
            [OS] => Windows_NT
            ....
            ...
```

V - Emulation d'exception avec PHP4

Le principe des exceptions est d'envoyer un signal défini par le développeur en fonction d'un stimuli choisi par le développeur qui impliquera un traitement écrit par le développeur. Les exceptions sont ainsi totalement définies par le développeur. Les erreurs sont à l'inverse totalement gérées par PHP. Cependant, le langage PHP définit plusieurs fonctions permettant au développeur de personnaliser cette gestion. Nous avons déjà vu set_error_handler, qui permet de redéfinir la fonction appelée lors d'une erreur (qui correspondrait à l'intérieur de notre bloc catch). Il nous reste à trouver comment "soulever" une erreur comme on soulevait des exceptions. Pour cela, on utilise la fonction trigger_error qui, lorsqu'elle est appelée, génère un message d'erreur.

Emulation d'exception en PHP4

```
function MyErrorHandler($code, $msg, $file, $line)
{
    echo 'div style="color:red;">'.$msg.'</div>';
}

set_error_handler('MyErrorHandler');

if ( ! @fopen() )
{
    trigger_error('Impossible d\'ouvrir le fichier', E_USER_ERROR);
}
```

On peut préciser en second paramètre un niveau d'erreur. Les choix possibles sont E_USER_NOTICE, E_USER_WARNING, E_USER_ERROR. Par défaut, l'erreur sera de type E_USER_NOTICE.

Conclusion

Les exceptions sont une bonne alternative à la gestion des erreurs traditionnelles. Cependant, comme nous l'avons vu, il faut plus les voir comme un complément. PHP nous permet en effet d'utiliser les exceptions pour le langage lui-même, mais ne le fait pas nativement, car celles-ci sont bien trop coûteuses en ressources et en lisibilité. Elles peuvent être utilisés dans deux cas.

- A la place du gestionnaire d'erreur, quand la fiabilité prime sur les performances
- En complément, pour assurer une sécurité d'exécution de parties de code sensibles.