Error Handling

Disclaimer: вы смотрите просто запись лекции,
это **НЕ** специально подготовленный видеокурс!

Intro

While PHP always tries to continue script execution (even with notices and/or warnings) it is a “must have” to analyze any situation for possible errors and to handle those potential errors.

Error Handling Functions: error_reporting

It is a good idea to keep error_reporting option set to 0 in php.ini and/or to use error_reporting(0) in production environment.

```
<?php

// Turns off all error messages (for this particular script execution)
error_reporting(0);

// No more error message here
$someData = file('non_existing_file');
```

See error_reporting option in php.ini for the list of values and samples of those values combinations.

Error Handling Functions: trigger_error (alias: user_error)

This function is useful while creating some frameworks and/or libraries. Still you may also use exceptions (see later).

```php
<?php

$someArray = [1, 2, 3];

if (!isset($someArray[999])) {
    trigger_error('No 999th element in array!', E_USER_ERROR);
}
```

Error Handling Functions: error_get_last and error_clear_last

These are another functions useful in frameworks and/or libraries. In most cases there are more efficient ways to analyze an error, but still…

```php
<?php

$someArray = [1, 2, 3];
echo $someArray[999];
print_r(error_get_last());

/*
Array
(
    [type] => 2
    [message] => Undefined array key 999
    [file] => D:\PWD_Code_Samples\03 07 - Error Handling\03_error_get_last.php
    [line] => 4
)
*/

error_clear_last();
print_r(error_get_last()); // Empty result
```

If an error stops script execution, you'll never get to this line ☹.

Error Handling Functions: debug_backtrace and debug_print_backtrace

These functions are useful if you are creating something really huge and complex and need to provide your own debug mechanism to end-users:

```php
<?php

function someFunctionOne(): void
{
    someFunctionTwo('Test');
}

function someFunctionTwo(string $msg): void
{
    print_r(debug_backtrace());
    debug_print_backtrace();
}

someFunctionOne();
```

```
/*
Array
(
    [0] => Array
        (
            [file] => D:\PWD_Code_Samples\03 07 - Error Handling\04_debug_backtrace.php
            [line] => 5
            [function] => someFunctionTwo
            [args] => Array
                (
                    [0] => Test
                )

        )

    [1] => Array
        (
            [file] => D:\PWD_Code_Samples\03 07 - Error Handling\04_debug_backtrace.php
            [line] => 14
            [function] => someFunctionOne
            [args] => Array
                (
                )

        )

)
#0 D:\PWD_Code_Samples\03 07 - Error Handling\04_debug_backtrace.php(5): someFunctionTwo('Test')
#1 D:\PWD_Code_Samples\03 07 - Error Handling\04_debug_backtrace.php(14): someFunctionOne()
*/
```

Error Handling Functions: error_log

While in most cases you may create your own efficient logging mechanism (see later) there is a standard approach:

```php
<?php

$someArray = [1, 2, 3];

if (!isset($someArray[999])) {
    // Write a message to a log set be error_log option in php.ini
    error_log('No 999th element in array!', 0);

    // Sand a e-mail to administrator
    error_log('No 999th element in array!', 1, "admin@site.com");

    // Write a message to some log file
    error_log('No 999th element in array!', 3, 'D:/our_error_log.txt');
}
```

Usually, this will just print a message to the screen.

E-mail sending mechanism should be set for this to work.

There will not be any ‘decorations’, this will just write this string ‘as is’.

Error Handling Functions: set_error_handler and restore_error_handler

These functions (along with set_exception_handler and restore_exception_handler) are usually most efficient (see a big sample soon):

```php
<?php

function customErrorHandler($errno, $errstr, $errfile, $errline)
{
    echo 'Error #   : ' . $errno . "\n";
    echo 'Error is  : ' . $errstr . "\n";
    echo 'Error file: ' . $errfile . "\n";
    echo 'Error line: ' . $errline . "\n";
}

$oldErrorHandler = set_error_handler('customErrorHandler');
echo $someVariable;
/*
Error #   : 2
Error is  : Undefined variable $someVariable
Error file: D:\PWD_Code_Samples\03 07 - Error Handling\06_set_error_handler.php
Error line: 12
*/

restore_error_handler();
echo $someVariable;
/* Warning: Undefined variable $someVariable in D:\PWD_Code_Samples\03 07 - Error Handling\06_set_error_handler.php on line 21 */
```

Error Handling: set_exception_handler and restore_exception_handler

These functions (along with set_error_handler and restore_error_handler) are usually most efficient (see a big sample soon):

```php
<?php

function customExceptionHandler($exception)
{
    echo 'Exception: ' . $exception->getMessage() . "\n";
}

set_exception_handler('customExceptionHandler');
throw new Exception('TEST');

/*
Exception: TEST
*/

// As we've thrown an exception, the script will be terminated,
// and we'll never get to this line:
restore_exception_handler();
throw new Exception('TEST');
```

Error Handling: remember “magic constants”

`__LINE__`	The current line number of the file.
`__FILE__`	The full path and filename of the file with symlinks resolved.
`__DIR__`	The directory of the file.
`__FUNCTION__`	The function name, or {closure} for anonymous functions.
`__CLASS__`	The class name (including the namespace if it was declared).
`__TRAIT__`	The trait name (including the namespace if it was declared).
`__METHOD__`	The class method name.
`__NAMESPACE__`	The name of the current namespace.
`ClassName::class`	The fully qualified class name.

Useful for logging, debugging, complex frameworks linking.

Error Handling: remember useful pre-defined constants

See https://www.php.net/manual/en/reserved.constants.php:

```php
<?php

echo PHP_VERSION . "\n";            // 8.1.0
echo PHP_MAJOR_VERSION . "\n";      // 8
echo PHP_MINOR_VERSION  . "\n";     // 1
echo PHP_MAXPATHLEN . "\n";         // 2048
echo PHP_OS . "\n";                 // WINNT
echo PHP_OS_FAMILY . "\n";          // Windows
echo PHP_SAPI . "\n";               // cli
echo PHP_EOL . "\n";                // \r\n
echo PHP_INT_MAX . "\n";            // 9223372036854775807
echo PHP_INT_MIN  . "\n";           // -9223372036854775808
// ... and so on.
```

Error Handling: big sample (maybe not so big, but useful ☺)

```php
<?php

function customErrorHandler($errno, $errstr, $errfile, $errline)
{
   // We just 'convert' any error to an exception
   throw new Exception('Error [' . $errstr . '] (number [' . $errno .']) in [' .
                                    $errfile . '] at line [' . $errline . ']');
}

function customExceptionHandler($exception)
{
   // Here we may use any additional data to make the message more informative:
   error_log(date('Y.m.d H:i:s') . ' Exception: ' . $exception->getMessage() . "\n", 0);
}

set_error_handler('customErrorHandler');
set_exception_handler('customExceptionHandler');

echo $someVariable;
// 2022.01.05 16:20:32 Exception: Error [Undefined variable $someVariable]
// (number [2]) in [D:\PWD_Code_Samples\03 07 - Error Handling\08_big_sample.php] at line [19]
```

Exceptions

Exceptions are error situations that may not (or should not) be processed by the code that has detected the situation.

The main idea is to inform the calling code about the situation leaving all further decisions to that code. If you may handle error situation locally, you should not throw an exception.

Let’s see an example…

Exceptions: where to use

Imagine, you are creating some framework/library:

```php
<?php

// Let's imagine that this function is in some library you are creating...
function copyFile($sourceFileName, $destinationFileName) : bool
{
    $finalResult = false;

    if (!is_file($sourceFileName)) {
        throw new Exception('Source file [' . $sourceFileName . '] does not exist!');
    }

    // ... Here goes a lot of useful code ...

    return $finalResult;
}
```

You have absolutely no idea what to do next, so you just raise an exception telling the calling code to deal with this situation.

```php
<?php

// And this is how someone will use your library...
try {
    copyFile('c:/non_existing_file.ext', 'c:/copy.ext');
} catch (Exception $e) {
    // Some reaction here...
}
```

And the calling code should catch the exception and react accordingly.

Exceptions: where NOT to use

Imagine, you are creating some utility to convert images:

```php
<?php

if ($argc < 3) {
    echo "USAGE: php image_converter.php InputImageFileName InputImageFileName\n";
    exit(-1);
}
```

This is 100% your own code, you absolutely know what to do in this situation, you don’t need to pass this situation handling to any other code.

Exceptions: this is insanity, NEVER do such things!

Just don’t do it:

```php
<?php

$inputImageFileName = '';
$outputImageFileName = '';

// Use 'if' instead!!!
try {
    $inputImageFileName = realpath($argv[1]);
} catch (Exception $e) {
    echo "InputImageFileName is missing!\n";
    echo "USAGE: php image_converter.php InputImageFileName InputImageFileName\n";
    exit(-1);
}

// Use 'if' instead!!!
try {
    $outputImageFileName = realpath($argv[2]);
} catch (Exception $e) {
    echo "OutputImageFileName is missing!\n";
    echo "USAGE: php image_converter.php InputImageFileName InputImageFileName\n";
    exit(-2);
}

// This is ultimate madness. It's like talking in a messenger with yourself... :(
try {
    if (($inputImageFileName == '') || ($outputImageFileName == '')) {
        throw new Exception("USAGE: php image_converter.php InputImageFileName InputImageFileName\n");
    }
} catch (Exception $exception) {
    echo $exception->getMessage();
}
```

Exceptions are for exceptional occasions. If you may handle a potential error with ‘if’, don’t replace it with ‘try… catch’.

Exceptions: general syntax

PHP has an exception model similar to that of other programming languages.

```php
<?php

try {
    // Something that may go wrong...
} catch (ChildException $e) {
    // Reaction to lover-level exception.
} catch (ParentException $e) {
    // Reaction to higher-level exception.
} finally {
    // Some 'final' code that should work
    // no mater, if there was an exception,
    // or if there was not.
}
```

Exceptions: general syntax

Since PHP 8 you may omit variable here:

```php
<?php

try {
    // Something that may go wrong...
} catch (ChildException) {
    // Reaction to lover-level exception.
} catch (ParentException) {
    // Reaction to higher-level exception.
} finally {
    // Some 'final' code that should work
    // no mater, if there was an exception,
    // or if there was not.
}
```

Exceptions: general syntax with sample

```php
<?php

$linksOnThePage = $page->getAllLinks();

foreach ($linksOnThePage as $url) {
    try {
        $page->setInnerObjectsData($url, $crawler->getDataFromURL($url));
    } catch (HttpException_404) {
        $page->setInnerObjectsStatus($url, 404);
    } catch (HttpException_403) {
        $page->setInnerObjectsStatus($url, 403);
    } catch (HttpException) {
        $page->setInnerObjectsStatus($url, -1);
    } finally {
        $page->incrementInnerObjectsProcessedCount();
    }
}
```

How to produce an exception

We’ve already seen this, but – just to remember:

```php
<?php

// General approach:
if ($somethingIsWrong) {
    throw new Exception("Message");
}

// Since PHP 8 this approach is also available:
$someValue = $someArray['offset'] ?? throw new OffsetDoesNotExist('offset');
```

How to create custom exceptions

```php
<?php

class TypeMissmatchException extends Exception{};

class MathException extends Exception{};

function getAverage($numbers)
{
    if (!is_array($numbers)) {
        throw new TypeMissmatchException('Function getAverage expects first argument to be an array!');
    }

    if (sizeof($numbers) === 0) {
        throw new MathException('Function getAverage expects non empty array!');
    }

    // Useful code here :).
}

$test1 = 'Test';
$test2 = [];
$test3 = [1, 2, 3];

try {
    $avg = getAverage($test1);
} catch (TypeMissmatchException $e) {
    var_dump($e);
} catch (MathException $e) {
    var_dump($e);
}
```

Error Handling

Disclaimer: вы смотрите просто запись лекции, это **НЕ** специально подготовленный видеокурс!