<https://stackify.com/php-try-catch-php-exception-tutorial/>

<https://www.tutorialspoint.com/php/php_error_handling.htm>

Error handling is the process of catching errors raised by your program and then taking appropriate action. If you would handle errors properly then it may lead to many unforeseen consequences.

Its very simple in PHP to handle an errors.

Using die() function

While writing your PHP program you should check all possible error condition before going ahead and take appropriate action when required.

Try following example without having **/tmp/test.xt** file and with this file.

<?php

if(!file\_exists("/tmp/test.txt")) {

die("File not found");

}else {

$file = fopen("/tmp/test.txt","r");

print "Opend file sucessfully";

}

// Test of the code here.

?>

This way you can write an efficient code. Using above technique you can stop your program whenever it errors out and display more meaningful and user friendly message.

Defining Custom Error Handling Function

You can write your own function to handling any error. PHP provides you a framework to define error handling function.

This function must be able to handle a minimum of two parameters (error level and error message) but can accept up to five parameters (optionally: file, line-number, and the error context) −

Syntax

error\_function(error\_level,error\_message, error\_file,error\_line,error\_context);

|  |  |
| --- | --- |
| **Sr.No** | **Parameter & Description** |
| 1 | **error\_level**  Required - Specifies the error report level for the user-defined error. Must be a value number. |
| 2 | **error\_message**  Required - Specifies the error message for the user-defined error |
| 3 | **error\_file**  Optional - Specifies the file name in which the error occurred |
| 4 | **error\_line**  Optional - Specifies the line number in which the error occurred |
| 5 | **error\_context**  Optional - Specifies an array containing every variable and their values in use when the error occurred |

Possible Error levels

These error report levels are the different types of error the user-defined error handler can be used for. These values cab used in combination using **|**operator

|  |  |  |
| --- | --- | --- |
| **Sr.No** | **Constant & Description** | **Value** |
| 1 | **.E\_ERROR**  Fatal run-time errors. Execution of the script is halted | 1 |
| 2 | **E\_WARNING**  Non-fatal run-time errors. Execution of the script is not halted | 2 |
| 3 | **E\_PARSE**  Compile-time parse errors. Parse errors should only be generated by the parser. | 4 |
| 4 | **E\_NOTICE**  Run-time notices. The script found something that might be an error, but could also happen when running a script normally | 8 |
| 5 | **E\_CORE\_ERROR**  Fatal errors that occur during PHP's initial start-up. | 16 |
| 6 | **E\_CORE\_WARNING**  Non-fatal run-time errors. This occurs during PHP's initial start-up. | 32 |
| 7 | **E\_USER\_ERROR**  Fatal user-generated error. This is like an E\_ERROR set by the programmer using the PHP function trigger\_error() | 256 |
| 8 | **E\_USER\_WARNING**  Non-fatal user-generated warning. This is like an E\_WARNING set by the programmer using the PHP function trigger\_error() | 512 |
| 9 | **E\_USER\_NOTICE**  User-generated notice. This is like an E\_NOTICE set by the programmer using the PHP function trigger\_error() | 1024 |
| 10 | **E\_STRICT**  Run-time notices. Enable to have PHP suggest changes to your code which will ensure the best interoperability and forward compatibility of your code. | 2048 |
| 11 | **E\_RECOVERABLE\_ERROR**  Catchable fatal error. This is like an E\_ERROR but can be caught by a user defined handle (see also set\_error\_handler()) | 4096 |
| 12 | **E\_ALL**  All errors and warnings, except level E\_STRICT (E\_STRICT will be part of E\_ALL as of PHP 6.0) | 8191 |

All the above error level can be set using following PHP built-in library function where level cab be any of the value defined in above table.

int error\_reporting ( [int $level] )

Following is the way you can create one error handling function −

<?php

function handleError($errno, $errstr,$error\_file,$error\_line) {

echo "<b>Error:</b> [$errno] $errstr - $error\_file:$error\_line";

echo "<br />";

echo "Terminating PHP Script";

die();

}

?>

Once you define your custom error handler you need to set it using PHP built-in library **set\_error\_handler** function. Now lets examine our example by calling a function which does not exist.

<?php

error\_reporting( E\_ERROR );

function handleError($errno, $errstr,$error\_file,$error\_line) {

echo "<b>Error:</b> [$errno] $errstr - $error\_file:$error\_line";

echo "<br />";

echo "Terminating PHP Script";

die();

}

//set error handler

set\_error\_handler("handleError");

//trigger error

myFunction();

?>

Exceptions Handling

PHP 5 has an exception model similar to that of other programming languages. Exceptions are important and provides a better control over error handling.

Lets explain there new keyword related to exceptions.

* **Try** − A function using an exception should be in a "try" block. If the exception does not trigger, the code will continue as normal. However if the exception triggers, an exception is "thrown".
* **Throw** − This is how you trigger an exception. Each "throw" must have at least one "catch".
* **Catch** − A "catch" block retrieves an exception and creates an object containing the exception information.

When an exception is thrown, code following the statement will not be executed, and PHP will attempt to find the first matching catch block. If an exception is not caught, a PHP Fatal Error will be issued with an "Uncaught Exception ...

* An exception can be thrown, and caught ("catched") within PHP. Code may be surrounded in a try block.
* Each try must have at least one corresponding catch block. Multiple catch blocks can be used to catch different classes of exceptions.
* Exceptions can be thrown (or re-thrown) within a catch block.

Example

Following is the piece of code, copy and paste this code into a file and verify the result.

<?php

try {

$error = 'Always throw this error';

throw new Exception($error);

// Code following an exception is not executed.

echo 'Never executed';

}catch (Exception $e) {

echo 'Caught exception: ', $e->getMessage(), "\n";

}

// Continue execution

echo 'Hello World';

?>

In the above example $e->getMessage function is used to get error message. There are following functions which can be used from **Exception** class.

* **getMessage()** − message of exception
* **getCode()** − code of exception
* **getFile()** − source filename
* **getLine()** − source line
* **getTrace()** − n array of the backtrace()
* **getTraceAsString()** − formated string of trace

Creating Custom Exception Handler

You can define your own custom exception handler. Use following function to set a user-defined exception handler function.

string set\_exception\_handler ( callback $exception\_handler )

Here **exception\_handler** is the name of the function to be called when an uncaught exception occurs. This function must be defined before calling set\_exception\_handler().

Example

<?php

function exception\_handler($exception) {

echo "Uncaught exception: " , $exception->getMessage(), "\n";

}

set\_exception\_handler('exception\_handler');

throw new Exception('Uncaught Exception');

echo "Not Executed\n";

?>

These are functions dealing with error handling and logging. They allow you to define your own error handling rules, as well as modify the way the errors can be logged. This allows you to change and enhance error reporting to suit your needs.

Using these logging functions, you can send messages directly to other machines, to an email, to system logs, etc., so you can selectively log and monitor the most important parts of your applications and websites.

## Installation

The error and logging functions are part of the PHP core. There is no installation needed to use these functions.

## Runtime Configuration

The behaviour of these functions is affected by settings in php.ini. These settings are defined below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Default** | **Changeable** | **Changelog** |
| error\_reporting | NULL | PHP\_INI\_ALL |  |
| display\_errors | "1" | PHP\_INI\_ALL |  |
| display\_startup\_errors | "0" | PHP\_INI\_ALL | Available since PHP 4.0.3. |
| log\_errors | "0" | PHP\_INI\_ALL |  |
| log\_errors\_max\_len | "1024" | PHP\_INI\_ALL | Available since PHP 4.3.0. |
| ignore\_repeated\_errors | "0" | PHP\_INI\_ALL | Available since PHP 4.3.0. |
| ignore\_repeated\_source | "0" | PHP\_INI\_ALL | Available since PHP 4.3.0. |
| report\_memleaks | "1" | PHP\_INI\_ALL | Available since PHP 4.3.0. |
| track\_errors | "0" | PHP\_INI\_ALL |  |
| html\_errors | "1" | PHP\_INI\_ALL | PHP\_INI\_SYSTEM in PHP <= 4.2.3. Available since PHP 4.0.2. |
| docref\_root | "" | PHP\_INI\_ALL | Available since PHP 4.3.0. |
| docref\_ext | "" | PHP\_INI\_ALL | Available since PHP 4.3.2. |
| error\_prepend\_string | NULL | PHP\_INI\_ALL |  |
| error\_append\_string | NULL | PHP\_INI\_ALL |  |
| error\_log | NULL | PHP\_INI\_ALL |  |
| warn\_plus\_overloading | NULL |  | This option is no longer available as of PHP 4.0.0 |

## PHP Error and Logging Constants

**PHP** − indicates the earliest version of PHP that supports the constant.

You can use any of the constant while configuring your php.ini file.

|  |  |  |
| --- | --- | --- |
| **Value** | **Constant & Description** | **PHP** |
| 1 | **E\_ERROR**  Fatal run-time errors. Errors that cannot be recovered from. Execution of the script is halted |  |
| 2 | **E\_WARNING**  Non-fatal run-time errors. Execution of the script is not halted |  |
| 4 | **E\_PARSE**  Compile-time parse errors. Parse errors should only be generated by the parser |  |
| 8 | **E\_NOTICE**  Run-time notices. The script found something that might be an error, but could also happen when running a script normally |  |
| 16 | **E\_CORE\_ERROR**  Fatal errors at PHP startup. This is like an E\_ERROR in the PHP core | 4 |
| 32 | **E\_CORE\_WARNING**  Non-fatal errors at PHP startup. This is like an E\_WARNING in the PHP core | 4 |
| 64 | **E\_COMPILE\_ERROR**  Fatal compile-time errors. This is like an E\_ERROR generated by the Zend Scripting Engine | 4 |
| 128 | **E\_COMPILE\_WARNING**  Non-fatal compile-time errors. This is like an E\_WARNING generated by the Zend Scripting Engine | 4 |
| 256 | **E\_USER\_ERROR**  Fatal user-generated error. This is like an E\_ERROR set by the programmer using the PHP function trigger\_error() | 4 |
| 512 | **E\_USER\_WARNING**  Non-fatal user-generated warning. This is like an E\_WARNING set by the programmer using the PHP function trigger\_error() | 4 |
| 1024 | **E\_USER\_NOTICE**  User-generated notice. This is like an E\_NOTICE set by the programmer using the PHP function trigger\_error() | 4 |
| 2048 | **E\_STRICT**  Run-time notices. PHP suggest changes to your code to help interoperability and compatibility of the code | 5 |
| 4096 | **E\_RECOVERABLE\_ERROR**  Catchable fatal error. This is like an E\_ERROR but can be caught by a user defined handle (see also set\_error\_handler()) | 5 |
| 8191 | **E\_ALL**  All errors and warnings, except of level E\_STRICT | 5 |

## List of Functions

**PHP** − indicates the earliest version of PHP that supports the function.

|  |  |  |
| --- | --- | --- |
| **Sr.No** | **Advanced & Description** | **PHP** |
| 1 | [**debug\_backtrace()**](https://www.tutorialspoint.com/php/php_function_debug_backtrace.htm)  Generates a backtrace | 4 |
| 2 | [**debug\_print\_backtrace()**](https://www.tutorialspoint.com/php/php_function_debug_print_backtrace.htm)  Prints a backtrace | 5 |
| 3 | [**error\_get\_last()**](https://www.tutorialspoint.com/php/php_function_error_get_last.htm)  Gets the last error occurred | 5 |
| 4 | [**error\_log()**](https://www.tutorialspoint.com/php/php_function_error_log.htm)  Sends an error to the server error-log, to a file or to a remote destination | 4 |
| 5 | [**error\_reporting()**](https://www.tutorialspoint.com/php/php_function_error_reporting.htm)  Specifies which errors are reported | 4 |
| 6 | [**restore\_error\_handler()**](https://www.tutorialspoint.com/php/php_function_restore_error_handler.htm)  Restores the previous error handler | 4 |
| 7 | [**restore\_exception\_handler()**](https://www.tutorialspoint.com/php/php_function_restore_exception_handler.htm)  Restores the previous exception handler | 5 |
| 8 | [**set\_error\_handler()**](https://www.tutorialspoint.com/php/php_function_set_error_handler.htm)  Sets a user-defined function to handle errors | 4 |
| 9 | [**set\_exception\_handler()**](https://www.tutorialspoint.com/php/php_function_set_exception_handler.htm)  Sets a user-defined function to handle exceptions | 5 |
| 10 | [**trigger\_error()**](https://www.tutorialspoint.com/php/php_function_trigger_error.htm)  Creates a user-defined error message | 4 |
| 11 | [**user\_error()**](https://www.tutorialspoint.com/php/php_function_user_error.htm)  Alias of trigger\_error() | 4 |

# **PHP Try Catch: Basics & Advanced PHP Exception Handling Tutorial**

When PHP version 5 was released, it incorporated a built-in model to catch errors and exceptions. Handling errors in PHP with try catch blocks is almost the same as handling errors in other programming languages.

When a PHP exception is thrown, the PHP runtime looks for a catch statement that can handle that type of exception. It will continue checking the calling methods up the stack trace until a catch statement is found. If one is not found, the exception is handed to the global exception handler that we will also cover in this article.

### Simple PHP try catch example

Here is an example of a basic PHP try catch statement.

try {

// run your code here

}

catch (exception $e) {

//code to handle the exception

}

finally {

//optional code that always runs

}

### PHP error handling keywords

The following keywords are used for PHP exception handling.

* **Try:**The try block contains the code that may potentially throw an exception. All of the code within the try block is executed until an exception is potentially thrown.
* **Throw:**The throw keyword is used to signal the occurrence of a PHP exception. The PHP runtime will then try to find a catch statement to handle the exception.
* **Catch:**This block of code will be called only if an exception occurs within the try code block. The code within your catch statement must handle the exception that was thrown.
* **Finally:** In PHP 5.5, the finally statement is introduced. The finally block may also be specified after or instead of catch blocks. Code within the finally block will always be executed after the try and catch blocks, regardless of whether an exception has been thrown, and before normal execution resumes. This is useful for scenarios like closing a database connection regardless if an exception occurred or not.

### PHP try catch with multiple exception types

PHP supports using multiple catch blocks within try catch. This allows us to customize our code based on the type of exception that was thrown. This is useful for customizing how you display an error message to a user, or if you should potentially retry something that failed the first time.

try {

// run your code here

}

catch (Exception $e) {

echo $e->getMessage();

}

catch (InvalidArgumentException $e) {

echo $e->getMessage();

}

### When to use try catch-finally

In PHP version 5.5, the finally block was added. Sometimes in your [PHP error handling](https://stackify.com/php-error-handling-guide/)code, you will also want to use a finally section. Finally is useful for more than just exception handling, it is used to perform cleanup code such as closing a file, closing a database connection, etc.

**The finally block always executes when the try catch block exits.** This ensures that the finally block is executed even if an unexpected exception occurs.

Example for try catch-finally:

try {

print "this is our try block n";

throw new Exception();

} catch (Exception $e) {

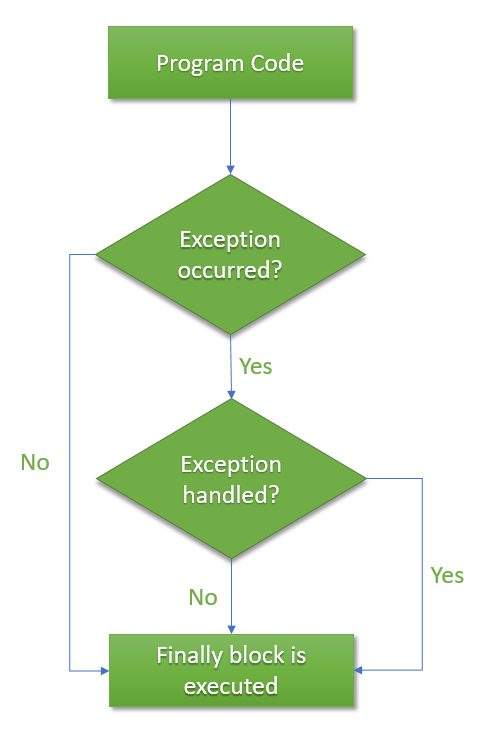
print "something went wrong, caught yah! n";

} finally {

print "this part is always executed n";

}

Below is the diagram showing how the program works.



### Creating custom PHP exception types

PHP also allows creating custom exception types.This can be useful for creating custom exceptions in your application that you can have special exception handling around.

To create a custom exception handler, we must create a special class with functions that can be called when an exception occurs. The class must be an extension of the exception class.

class DivideByZeroException extends Exception {};

The custom exception class inherits the properties from PHP’s **Exception** class and you can add custom functions to it. You may not want to display all the details of an exception to the user or you can display a user-friendly message and log the error message internally for monitoring.

The sample below uses a custom PHP exception with multiple catch statements.

class DivideByZeroException extends Exception {};

class DivideByNegativeException extends Exception {};

function process\_divide($denominator)

{

try

{

if ($denominator == 0)

{

throw new DivideByZeroException();

}

else if ($denominator < 0)

{

throw new DivideByNegativeException();

}

else

{

echo 100 / $denominator;

}

}

catch (DivideByZeroException $ex)

{

echo "Divide by zero exception!";

}

catch (DivideByNegativeException $ex)

{

echo "Divide by negative number exception!";

}

catch (Exception $x)

{

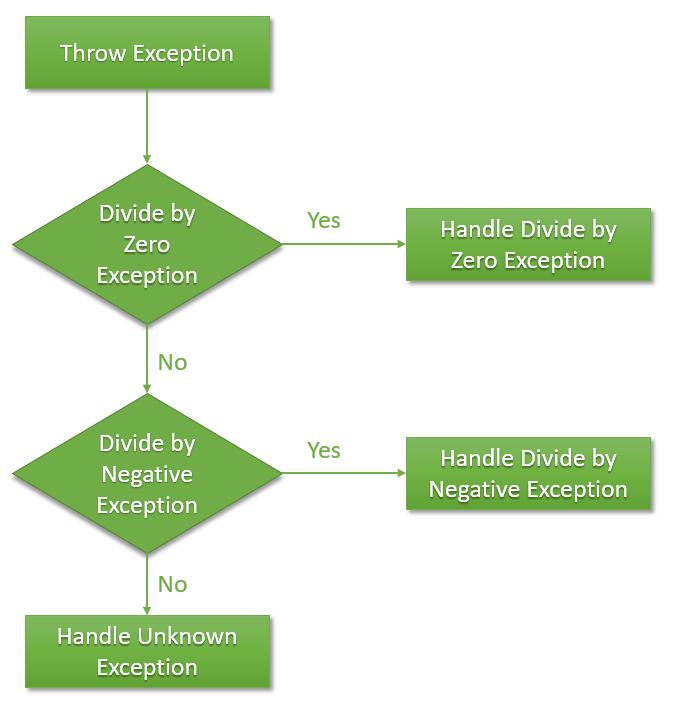
echo "UNKNOWN EXCEPTION!";

}

}

The code above throws an exception and catches it with a custom exception class. The DivideByZeroException() and DivideByNegativeException() classes are created as extensions of the existing Exception class; this way, it inherits all methods and properties from the Exception class. The “try” block is executed and an exception is thrown if the denominator is zero or negative number. The “catch” block catches the exception and displays the error message.

The flowchart below summarizes how our sample code above works for custom types of exceptions.



### Global PHP exception handling

In a perfect world, your code will do proper exception handling. As a best practice, you should also configure a global PHP exception handler. It will be called in case an unhandled exception occurs that was not called in a proper PHP try catch block.

To configure a global PHP exception handler, we will use the set\_exception\_handler() function to set a user-defined function to handle all uncaught exceptions:

function our\_global\_exception\_handler($exception) {

//this code should log the exception to disk and an error tracking system

echo "Exception:" . $exception->getMessage();

}

set\_exception\_handler(‘our\_global\_exception\_handler’);

### How to properly log exceptions in your PHP try catch blocks

Logging is usually the eyes and ears for most developers when it comes to troubleshooting application problems. Logging exceptions so you can find them after they happen is a really important part of [PHP error handling best practices](https://stackify.com/php-error-handling-guide/).

Error logs are crucial during development because it allows developers to see warnings, errors, notices, etc. that were logged while the application is running. That you can handle them appropriately through the PHP exception handling techniques try catch we just learned.

Depending on the [PHP framework](https://stackify.com/php-frameworks-development/) you are using, whether Laravel, Codeigniter, Symfony, or others, they may provide built-in logging frameworks. You can also use [Monolog](https://stackify.com/php-monolog-tutorial/), which is a standard PHP logging library. Regardless of the logging framework you are using, you want to always log important exceptions being thrown in your code.

Here is a sample of a try/catch that logs errors with Monolog:

require\_once(DIR.'/vendor/autoload.php');

use MonologLogger;

use MonologHandlerStreamHandler;

$logger = new Logger('channel-name');

$logger->pushHandler(new StreamHandler(DIR.'/app.log', Logger::DEBUG));

try {

// Code does some stuff

// debug logging statement

$logger->info('This is a log! ^\_^ ');

}

catch (Exception $ex) {

$logger->error('Oh no an exception happened! ');

}

### How to use try catch with MySQL

The PHP libraries for MySQL, PDO, and mysqli, have different modes for error handling. If you do not have exceptions enables for those libraries, you can’t use try catch blocks. This makes error handling different and perhaps more complicated.

#### **PDO**

In PDO, you must enable ERRMODE\_EXCEPTION when creating the connection.

// connect to MySQL

$conn = new PDO('mysql:host=localhost;dbname=stackifydb;charset=utf8mb4', 'username', 'password');

//PDO error mode to exception

$conn->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);

Learn more about [PDO attributes](http://php.net/manual/en/pdo.setattribute.php) from the PHP docs.

#### **MySQL**

For mysqli, you must do something similar:

mysqli\_report(MYSQLI\_REPORT\_ERROR | MYSQLI\_REPORT\_STRICT);

Learn more from the [MySQL docs](https://dev.mysql.com/doc/apis-php/en/apis-php-mysqli-driver.report-mode.html).

### How to view all PHP exceptions in one place

Proper exception handling in PHP is very important. As part of that, you don’t want to simply log your exceptions to a log file and never know they occurred.

The solution is to use an error tracking solution like Stackify’s [Retrace](https://stackify.com/retrace/). All [errors are recorded](https://stackify.com/retrace-error-monitoring/) together with some important details about that error, including the time it occurred, the method that caused it, and the exception type.

Retrace provides many important error tracking and monitoring features. Including the ability to see all except in one place, identifying unique errors, quicking finding new errors after a deployment, email notifications about new errors, and much more.

Stackify’s error and [log management tool](https://stackify.com/retrace-log-management/) can help you easily monitor and troubleshoot your application.

**

### ****Summary****

In this tutorial, we showed how to use PHP try catch blocks. We also covered some advanced uses with multiple exception types and even how to log all of your errors to a logging library. Good [error handling best practices](https://stackify.com/php-error-handling-guide/) are critical to any PHP application. [Retrace](https://stackify.com/retrace/) can help you quickly find and troubleshoot all of the exceptions being thrown in your code.