PHP Error Handling
Introduction

• How to write an error handler.
• Normally, it displays a message indicating the cause of the error and may also terminate script execution when a PHP script encounters an error.
• Now, while this behaviour is acceptable during the development phase, it cannot continue once a PHP application has been released to actual users.
• It is more professional to intercept these errors and either resolve them (if resolution is possible), or notify the user with an easily-understood error message (if not).
There are 3 basic types of runtime errors in PHP:

i. **Notices**: These are trivial, non-critical errors that PHP encounters while executing a script. By default, such errors are not displayed to the user at all – we can change this default behaviour.

ii. **Warnings**: These are more serious errors - for example, attempting to include() a file which does not exist. By default, these errors are displayed to the user, but they do not result in script termination.

iii. **Fatal errors**: These are critical errors - for example, instantiating an object of a non-existent class, or calling a non-existent function. These errors cause the immediate termination of the script, and PHP's default behaviour is to display them to the user when they take place.
• It should be noted that a syntax error in a PHP script – (EG: missing brace or semi-colon) is treated as a fatal error and results in script termination (stop execution).

• PHP errors can be generated by:
  i. Zend engine
  ii. PHP built-in functions
  iii. user-defined functions

They may occur at startup, at parse-time, at compile-time or at run-time.

• During the debug phase:
  ➢ Use E_ALL type to see all fatal and non-fatal errors generated by our script.
  ➢ Use the new E_STRICT error type to view errors that affect the forward compatibility of our code in PHP 5.
Early Warning

- Example 1
  - we get a non-fatal error (E_WARNING)
  - it still get executed but with non-fatal error

- Example 2
  - it call non-existent function
  - generate a fatal error
  - immediately stops script execution
Example 1

```php
// initialize the $string variable
$string = 'a string';

// explode() a string
// this will generate a warning or E_WARNING because the number of arguments to explode() is incorrect
explode($string);
```

• Output

**Warning**: Wrong parameter count for explode() in C:\wamp\www\test2\part12\eg1.php on line 8
Example 2

```php
// call a non-existent function
// this will generate a fatal error (E_ERROR)
callMeJoe();

?>

• Output

Fatal error: Call to undefined function callMeJoe() in C:\wamp\www\test2\part12\eg2.php on line 5
```
• Example 3
  ➢ control which errors are displayed to the user (built-in PHP function called error_reporting())
  ➢ this function tells the script to report only errors that match that type
  ➢ "hide" non-fatal errors

• Example 4
  ➢ Use a similar technique to turn off the display of fatal errors
Example 3

```php
<?php

// report only fatal errors
error_reporting(E_ERROR);

// initialize the $string variable
$string = 'string';

// attempt to explode() a string
// this will not generate a warning because only fatal errors are reported
explode($string);

?>
```

- Output
  - No output
Example 4

```php
// report no fatal errors
error_reporting(~E_ERROR);

// call a non-existent function
callMeJoe();

?>
```

- Output
  No output
• Although the script above will not display a visible error message, script execution will still stop at the point of error and statements subsequent to that point will not be executed.

• error_reporting() gives you control over which errors are displayed; it doesn't prevent the errors themselves.
Rolling Your Own

- Changing the way errors are handled.
- Function called set_error_handler(), it allows to divert all PHP errors to a custom function that are defined, instead of sending them to the default handler.
- This custom function must be capable of accepting a minimum of two mandatory arguments:
  i. error type
  ii. corresponding descriptive message
- and up to three additional arguments
  i. the file name
  ii. line number where the error occurred
  iii. dump of the variable space at the time of error
• Example 5
  ➢ The `set_error_handler()` function tells the script that all errors are to be routed to my user-defined oops() function
  ➢ This function is set up to accept five arguments:
    i. error type
    ii. message
    iii. file name
    iv. line number
    v. context
  ➢ These arguments are then used to create an error page that is friendlier and more informative than PHP's standard one-line error message
Example 5

```php
// define a custom error handler
set_error_handler('oops');

// initialize the $string variable
$string = 'a string';

// explode() a string
// this will generate a warning because the number of arguments to explode() is incorrect
// the error will be caught by the custom error handler
explode($string);
```
// custom error handler
function oops($type, $msg, $file, $line, $context) {
    echo "<h1>Error!</h1>"
    echo "An error occurred while executing this script. Please contact the <a href=mailto:webmaster@somedomain.com>webmaster</a> to report this error.";
    echo "<p />";
    echo "Here is the information provided by the script:";
    echo "<hr><pre>"
    echo "Error code: $type<br/>";
    echo "Error message: $msg<br/>";
    echo "Script name and line number of error: $file:$line<br/>";
    $variable_state = array_pop($context);
    echo "Variable state when error occurred: ";
    print_r($variable_state);
    echo "</pre><hr>";
}

?>

Output

Error!
• An error occurred while executing this script. Please contact the webmaster to report this error. Here is the information provided by the script:

• Error code: 2
  Error message: Wrong parameter count for explode()
  Script name and line number of error: C:\wamp\www\test2\part12\eg5.php:12
  Variable state when error occurred: a string
• Example 6
  ➢ Can use this custom error handler to alter the error message the user sees, on the basis of the error type

• Note that certain error types can't be handled in this way.
• For example, a fatal E_ERROR will prevent the PHP script from continuing.
• So, it can never reach a user-created error-handling mechanism.
Example 6

```php
// define a custom error handler
set_error_handler('oops');

// initialize $string variable
$string = 'a string';

// this will generate a warning
explode($string);

// custom error handler
function oops($type, $msg, $file, $line, $context) {
    switch ($type) {
        case E_NOTICE:
            // do nothing
            break;
    }
}
```
// warnings
case E_WARNING:
    // report error
    print "Non-fatal error on line $line of $file: $msg <br />
break;

    // other
    default:
    print "Error of type $type on line $line of $file: $msg <br />
break;

?>

• **Output**
   Non-fatal error on line 10 of C:\wamp\www\test2\part12\eg6.php: Wrong parameter count for explode()
Pulling the Trigger

- PHP allows you to use its built-in error handling system to raise your own custom errors as well.
- This is accomplished via a function named `trigger_error()`, which allows you to raise any of the three error types reserved for users: `E_USER_NOTICE`, `E_USER_WARNING` and `E_USER_ERROR`.
- When these errors are triggered, PHP's built-in handler will automatically wake up to handle them.
- Example(`triggererror6.php`):
Example(triggerror6.php):

```php
<?php

// function to test a number
// generates E_USER_WARNING if number is a float
// generates E_USER_ERROR is number is negative
function testNumber($num) {
    // float
    // trigger a warning
    if (is_float($num)) {
        trigger_error("Number $num is not an integer", E_USER_WARNING);
    }

    // negative
    // trigger a fatal error
    if ($num < 0) {
        trigger_error("Number $num is negative", E_USER_ERROR);
    }
}

// test the function with different values
testNumber(100);
testNumber(5.6);
testNumber(-8);

?>

----output-----
Warning: Number 5.6 is not an integer in C:\wamp\www\test2\part12\triggerror6.php on line 10
Fatal error: Number -8 is negative in C:\wamp\www\test2\part12\triggerror6.php on line 16
```
• The next example rewrites the previous script to use a user-defined error handler to have a custom error handler to handle your custom errors.

• Example(triggererror7.php):
Example(triggererror7.php):

```php
// function to test a number
// generates E_USER_WARNING if number is a float
// generates E_USER_ERROR if number is negative
function testNumber($num) {
    // float
    // trigger a warning
    if (is_float($num)) {
        trigger_error("Number $num is not an integer", E_USER_WARNING);
    }

    // negative
    // trigger a fatal error
    if ($num < 0) {
        trigger_error("Number $num is negative", E_USER_ERROR);
    }
}

// custom error handler
function myErrorHandler($type, $msg, $file, $line, $context) {
    switch ($type) {
        // warnings
        case E_USER_WARNING:
            // report error
            print "Non-fatal error on line $line of $file: $msg <br />";
            break;
```
```
// fatals
    case E_USER_ERROR:
        // report error and die()
        die("Fatal error on line $line of $file: $msg <br />");
        break;

    // notices
    default:
        // do nothing
        break;
}
}

// set the name of the custom handler
set_error_handler ('myErrorHandler');

// test the function with different values
testNumber(100);
testNumber(5.6);
testNumber(-8);

?>

-----output-----

Non-fatal error on line 10 of C:\wamp\www\test2\part12\triggererror7.php: Number 5.6 is not an integer
Fatal error on line 16 of C:\wamp\www\test2\part12\triggererror7.php: Number -8 is negative
Catching up

• In the exception-based approach, program code is wrapped in a try() block, and exceptions generated by it are "caught" and resolved by a catch() block.

• Multiple catch() blocks are possible, each one dealing with a different error type; this allows developers to trap different types of errors and execute appropriate exception-handling.

• When PHP encounters code wrapped within a try-catch() block:
  – It first attempts to execute the code within the try() block.
  – If this code is processed without any exceptions being generated, control transfers to the lines following the try-catch() block.
  – If an exception is generated while running the code within the try() block, PHP stops execution of the block at that point and begins checking each catch() block to see if there is a handler for the exception.
  – If a handler is found, the code within the appropriate catch() block is executed; if not, a fatal error is generated.
Catching up (cont.)

- The exceptions themselves are generated via PHP's throw statement.
- The throw statement needs to be passed a descriptive message, and an optional error code.
- When the exception is raised, this description and code will be made available to the exception handler.
- Example(tryncatcherror8.php):
Example (tryncatcherror8.php):

```php
<?php
    // PHP 5
    error_reporting(0);

    // try this code
    try {
        $file = 'somefile.txt';

        // open file
        if (!$fh = fopen($file, 'r')) {
            throw new Exception('Could not open file!');
        }

        // read file contents
        if (!$data = fread($fh, filesize($file))) {
            throw new Exception('Could not read file!');
        }

        // close file
        fclose($fh);
        // print file contents
        echo $data;
    }

    // catch errors if any
    catch (Exception $e) {
        print 'Something bad just happened...';
    }
?>

----output----Something bad just happened...
Catching up (cont..)

- If the file doesn't exist or is unreadable, the throw statement will generate an exception (basically, an instance of PHP's built-in Exception object) and pass it a message describing the error.
- When such an exception is generated, control passes to the first catch() block.
- If the catch() block can handle the exception type, the code within the catch() block is executed.
- If the first catch() block cannot handle the generated exception, control passes to the next one.
<?php
// PHP 5
error_reporting(0);

// try this code
try {
    $file = 'somefile.txt';

    // open file
    if (!$fh = fopen($file, 'r')) {
        throw new Exception('Could not open file!', 12);
    }

    // read file contents
    if (!$data = fread($fh, filesize($file))) {
        throw new Exception('Could not read file!', 9);
    }

    // close file
    fclose($fh);

    // print file contents
    echo $data;
} // catch errors if any
catch (Exception $e) {
    print '<h2>Exception</h2>
    print 'Error message: ' . $e->getMessage() . '<br />';
    print 'Error code: ' . $e->getCode() . '<br />';
    print 'File and line: ' . $e->getFile() . '(' . $e->getLine() . ')<br />';
    print 'Trace: ' . $e->getTraceAsString() . '<br />';
}
?>

----output----
Exception
Error message: Could not open file!
Error code: 12
File and line: C:\wamp\www\test2\part12\tryncatcherror9.php(12)
Trace: #0 {main}
When you run this script, you'll see that the message generated by the exception handler contains:
- the descriptive data sent by throw,
- an error code (also sent by throw),
- the file name and line number where the exception occurred, and a stack trace indicating the exception's progress through the class hierarchy, if there is one.

This data is generated by calling the Exception object's getMessage(), getCode(), getFile(), getLine() and getTraceAsString() methods respectively inside the catch() block.
Adding Some Class

• You can handle different exceptions in different ways, by sub-classing the generic Exception object and using more than one catch() block.

• Example(classerror10.php):
  – Note that because PHP will always use the first catch() block that matches the exception type and because the generic Exception class matches all exceptions, the catch() blocks must be arranged in the order of most specific first.
<?php
// PHP 5
// sub-class the Exception class
class NegativeNumException extends Exception {}
//class OutOfRangeException extends Exception {}
class FloatException extends Exception {}
// function to test a number
function testNumber($num) {
    // float
    // trigger an exception
    if (is_float($num)) {
        throw new FloatException($num);
    }
    // negative
    // trigger an exception
    if ($num < 0) {
        throw new NegativeNumException($num);
    }
    // out of range
    // trigger an exception
    if ($num > 1000 || $num < 100) {
        throw new OutOfRangeException($num);
    }
}
A negative number was provided (-19). Please provide a positive integer between 100 and 1000.