27. PHP Exception Handling

Exceptions are used to change the normal flow of a script if a specified error occurs.

27.1. What is an Exception

With PHP 5 came a new object oriented way of dealing with errors.

Exception handling is used to change the normal flow of the code execution if a specified error (exceptional) condition occurs. This condition is called an exception.

This is what normally happens when an exception is triggered:

- The current code state is saved
- The code execution will switch to a predefined (custom) exception handler function
- Depending on the situation, the handler may then resume the execution from the saved code state, terminate the script execution or continue the script from a different location in the code

We will show different error handling methods:

- Basic use of Exceptions
- Creating a custom exception handler
- Multiple exceptions
- Re-throwing an exception
- Setting a top level exception handler

Note: Exceptions should only be used with error conditions, and should not be used to jump to another place in the code at a specified point.

27.2. Basic Use of Exceptions

When an exception is thrown, the code following it will not be executed, and PHP will try to find the matching "catch" block.

If an exception is not caught, a fatal error will be issued with an "Uncaught Exception" message.

Lets try to throw an exception without catching it:

```
<?php
//create function with an exception
function checkNum($number)
  {
  if($number>1)
    {
    throw new Exception("Value must be 1 or below");
    }
  return true;
  }

//trigger exception
checkNum(2);
?>
```

The code above will get an error like this:

```
Fatal error: Uncaught exception 'Exception'
with message 'Value must be 1 or below' in
C:\webfolder\test.php:6
Stack trace: #0 C:\webfolder\test.php(12):
checkNum(28) #1 {main} thrown in C:\webfolder\test.php on
line 6
```

27.3. Try, throw and catch

To avoid the error from the example above, we need to create the proper code to handle an exception.

Proper exception code should include:

1. 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"
2. Throw - This is how you trigger an exception. Each "throw" must have at least one "catch"
3. Catch - A "catch" block retrieves an exception and creates an object containing the exception information

Lets try to trigger an exception with valid code:

```
<?php
//create function with an exception
function checkNum($number)
  {
  if($number>1)
    {
    throw new Exception("Value must be 1 or below");
    }
  return true;
  }

//trigger exception in a "try" block
try
  {
  checkNum(2);
  //If the exception is thrown, this text will not be
shown
  echo 'If you see this, the number is 1 or below';
  }

//catch exception
catch(Exception $e)
  {
  echo 'Message: ' .$e->getMessage();
  }
?>
```

The code above will get an error like this:

```
Message: Value must be 1 or below
```

Example explained:

The code above throws an exception and catches it:

1. The checkNum() function is created. It checks if a number is greater than 1. If it is, an exception is thrown
2. The checkNum() function is called in a "try" block
3. The exception within the checkNum() function is thrown
4. The "catch" block retrives the exception and creates an object ($e) containing the exception information
5. The error message from the exception is echoed by calling $e->getMessage() from the exception object

However, one way to get around the "every throw must have a catch" rule is to set a top level exception handler to handle errors that slip through.

27.4. Creating a Custom Exception Class

Creating a custom exception handler is quite simple. We simply create a special class with functions that can be called when an exception occurs in PHP. The class must be an extension of the exception class.

The custom exception class inherits the properties from PHP's exception class and you can add custom functions to it.

Lets create an exception class:

```
<?php
class customException extends Exception
  {
  public function errorMessage()
    {
    //error message
    $errorMsg = 'Error on line '.$this->getLine().' in '.
$this->getFile()
    .': <b>'.$this->getMessage().'</b> is not a valid E-
Mail address';
    return $errorMsg;
    }
  }

$email = "someone@example...com";

try
  {
  //check if
  if(filter_var($email, FILTER_VALIDATE_EMAIL) === FALSE)
    {
    //throw exception if email is not valid
    throw new customException($email);
    }
  }

catch (customException $e)
  {
  //display custom message
  echo $e->errorMessage();
  }
?>
```

The new class is a copy of the old exception class with an addition of the errorMessage() function. Since it is a copy of the old class, and it inherits the properties and methods from the old class, we can use the exception class methods like getLine() and getFile() and getMessage().

Example explained:

The code above throws an exception and catches it with a custom exception class:

1. The customException() class is created as an extension of the old exception class. This way it inherits all methods and properties from the old exception class
2. The errorMessage() function is created. This function returns an error message if an e-mail address is invalid
3. The $email variable is set to a string that is not a valid e-mail address
4. The "try" block is executed and an exception is thrown since the e-mail address is invalid

5. The "catch" block catches the exception and displays the error message

27.5. Multiple Exceptions

It is possible for a script to use multiple exceptions to check for multiple conditions.

It is possible to use several if..else blocks, a switch, or nest multiple exceptions. These exceptions can use different exception classes and return different error messages:

```
<?php
class customException extends Exception
{
public function errorMessage()
{
//error message
$errorMsg = 'Error on line '.$this->getLine().' in '.
$this->getFile()
.': <b>'.$this->getMessage().'</b> is not a valid E-Mail
address';
return $errorMsg;
}
}

$email = "someone@example.com";

try
  {
  //check if
  if(filter_var($email, FILTER_VALIDATE_EMAIL) === FALSE)
    {
    //throw exception if email is not valid
    throw new customException($email);
    }
  //check for "example" in mail address
  if(strpos($email, "example") !== FALSE)
    {
    throw new Exception("$email is an example e-mail");
    }
  }

catch (customException $e)
  {
  echo $e->errorMessage();
  }
```

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

Example explained:

The code above tests two conditions and throws an exception if any of the conditions are not met:

1. The customException() class is created as an extension of the old exception class. This way it inherits all methods and properties from the old exception class
2. The errorMessage() function is created. This function returns an error message if an e-mail address is invalid
3. The $email variable is set to a string that is a valid e-mail address, but contains the string "example"
4. The "try" block is executed and an exception is not thrown on the first condition
5. The second condition triggers an exception since the e-mail contains the string "example"
6. The "catch" block catches the exception and displays the correct error message

If the exception thrown were of the class customException and there were no customException catch, only the base exception catch, the exception would be handled there.

27.8. Re-throwing Exceptions

Sometimes, when an exception is thrown, you may wish to handle it differently than the standard way. It is possible to throw an exception a second time within a "catch" block.

A script should hide system errors from users. System errors may be important for the coder, but is of no interest to the user. To make things easier for the user you can re-throw the exception with a user friendly message:

```
     <?php
     class customException extends Exception
       {
       public function errorMessage()
         {
         //error message
         $errorMsg = $this->getMessage().' is not a valid E-
     Mail address.';
         return $errorMsg;
         }
       }
```

```
$email = "someone@example.com";

try
  {
  try
    {
    //check for "example" in mail address
    if(strpos($email, "example") !== FALSE)
      {
      //throw exception if email is not valid
      throw new Exception($email);
      }
    }
  catch(Exception $e)
    {
    //re-throw exception
    throw new customException($email);
    }
  }

catch (customException $e)
  {
  //display custom message
  echo $e->errorMessage();
  }
?>
```

Example explained:

The code above tests if the email-address contains the string "example" in it, if it does, the exception is re-thrown:

1. The customException() class is created as an extension of the old exception class. This way it inherits all methods and properties from the old exception class
2. The errorMessage() function is created. This function returns an error message if an e-mail address is invalid
3. The $email variable is set to a string that is a valid e-mail address, but contains the string "example"
4. The "try" block contains another "try" block to make it possible to re-throw the exception
5. The exception is triggered since the e-mail contains the string "example"
6. The "catch" block catches the exception and re-throws a "customException"
7. The "customException" is caught and displays an error message

If the exception is not caught in its current "try" block, it will search for a catch block on "higher levels".

27.9. Set a Top Level Exception Handler

The set_exception_handler() function sets a user-defined function to handle all uncaught exceptions.

```
<?php
function myException($exception)
{
echo "<b>Exception:</b> " , $exception->getMessage();
}

set_exception_handler('myException');

throw new Exception('Uncaught Exception occurred');
?>
```

The output of the code above should be something like this:

```
Exception: Uncaught Exception occurred
```

In the code above there was no "catch" block. Instead, the top level exception handler triggered. This function should be used to catch uncaught exceptions.

27.10. Rules for exceptions

- Code may be surrounded in a try block, to help catch potential exceptions
- Each try block or "throw" 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) in a catch block within a try block

A simple rule: If you throw something, you have to catch it.