Exceptions Handling

Introduction

- Exceptions are objects created (instantiated) to describe errors.

  Once an exception is instantiated it is thrown. We can write code to catch and handle it.

- We can handle the exceptions at different points in our script execution.

  We can code separated scripts to provide separated handling for each one of the possible exception types.

Introduction

- When a specific exception is not handled it functions as a fatal error that stops the execution of our PHP script.

  The exceptions thrown during a PHP script execution can change the flow of our code.

The `Exception` Class

- The exceptions are objects instantiated from a class that must extend `Exception` whether directly or indirectly.

```
class Exception
{
    protected $message = ’Unknown Exception’;
    protected $code = 0;
    protected $file;
    ...
}
```

The `Exception` Class

- When instantiating a class that extends `Exception` the interpreter takes care of filling nearing all member attributes of the new instantiated object. All is left is filling in the message ID number and the message textual message.

  We can easily extend the `Exception` class in order to describe specific errors and exceptions related to our application.

Throwing Exceptions

- We can create PHP code that throws exception using the 'throw' construct.

```
...
if($my_exception_condition)
{
    throw new MyException();
}
...
```

Throwing Exceptions

- When exception is thrown it bubbles up till it is either handled by a specific PHP script matching the thrown exception or becomes a fatal error that crashes our application.

The Try & Catch Block

- Exceptions can be caught using the try & catch block.

```
...
try
{
    doSomething();if($my_exception_condition)
    ...
}
catch(Exception $e)
{
    ...
}
...
```

Nesting Try & Catch Blocks

- We can nest different try & catch blocks within each other handling different type of exceptions in a different way.

```
...
try
{
    try
    {
        ...
    }
    catch(OneException $eOne) {... }
}
catch(TwoException $eTwo) {... }
...
```

Nesting Try & Catch Blocks

- We can place separated catch blocks to handle different types of exceptions in a separated different way.

```
...
try
{
    ...
}
catch(TwoException $eTwo)
{... }
catch(OneException $eOne)
{... }
...
```

Nesting Try & Catch Blocks

- Once an exception has been caught the execution will continue directly after the last enclosing catch block.

The 'catch all' Function

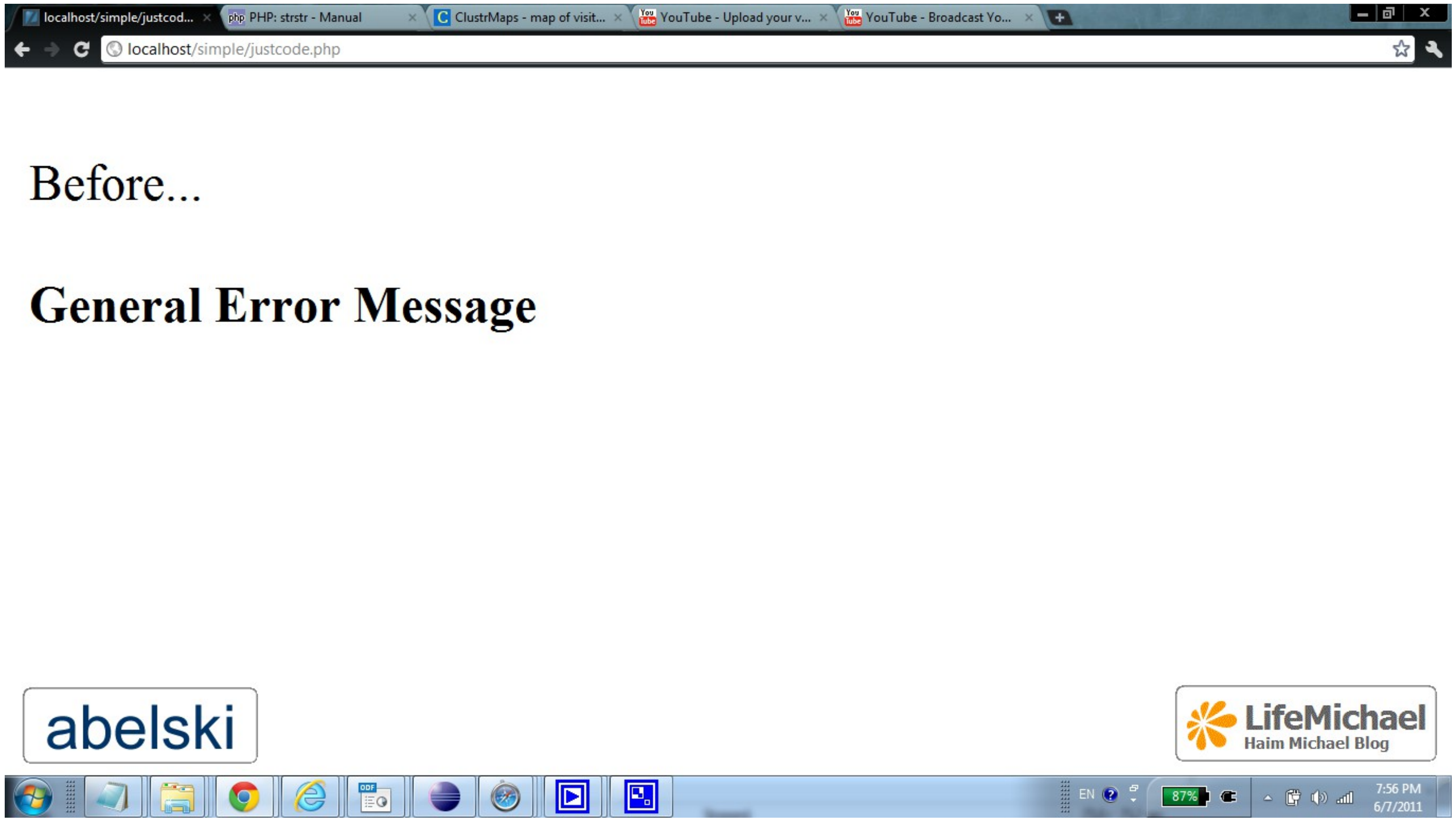
- Calling the `set_exception_handler()` we can set a specific function to be called whenever an exception is thrown and is not handled.

```
...
function myGeneralHandler($e exception)
{
    ...
}
...
set_exception_handler(“myGeneralHandler”)
...
```

Sample

```
<?php

function generalExceptionHandler($e)
{
    echo "<BR><B>General Error Message</B><BR>";
}

set_exception_handler("generalExceptionHandler");

echo "<BR>Before...<BR>";

if(true) throw new Exception("MokoBoko Exception La La La");

echo "<BR>After...</BR>";

?>
```

Sample


localhost/simple/justcod...
PHP: strstr - Manual
ClustrMaps - map of visit...
YouTube - Upload your v...
YouTube - Broadcast Yo...
localhost/simple/justcode.php
abelski
LifeMichael
Haim Michael Blog
EN
87%
7:56 PM
6/7/2011

Before...

General Error Message

The Finally Block

- As of PHP 5.5 we can add a finally block right after the last catch. Whether an exception was thrown or not and whether the exception was handheld or not, the finally block always executes.

The Finally Block

```
<?php
function divide($a,$b)
{
    if ($b===0)
    {
        throw new Exception('cannot divide by zero!');
    }
    return $a/$b;
}
```

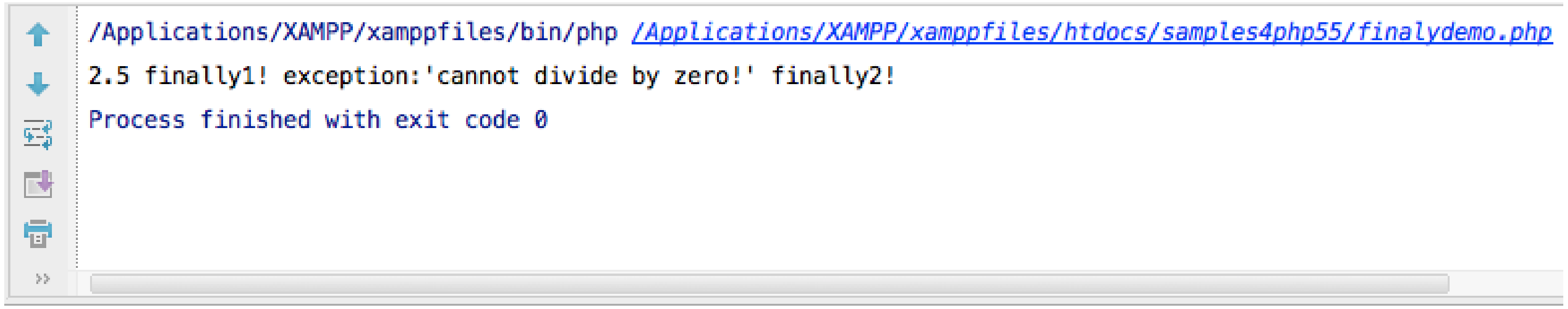
The Finally Block

```
try
{
    echo divide(5,2)." ";
}
catch (Exception $e)
{
    echo 'exception:\'',  $e->getMessage(), "' ";
}
finally
{
    echo "finally1! ";
}
```

The Finally Block

```
try
{
    echo divide(4,0)." ";
}
catch (Exception $e) {
    echo 'exception:\'',  $e->getMessage(), "' ";
}
finally
{
    echo "finally2! ";
}
?>
```

The Finally Block



```
/Applications/XAMPP/xamppfiles/bin/php /Applications/XAMPP/xamppfiles/htdocs/samples4php55/finalydemo.php
2.5 finally1! exception:'cannot divide by zero!' finally2!
Process finished with exit code 0
```

The Output

The `assert` Function

- The `assert` function allows us to specify a boolean expression that describes our expectation. We will usually use it in order to specify a precondition for each and every function.

The `assert` Function

- This function isn't new. Its prototype allows us to pass over two arguments. The second argument is optional. The first argument is the boolean expression we want to check. The second argument is either a string message we want the AssertionError object to include or a reference for a customized exception object we want to be thrown when the condition isn't met.

```
void assert (mixed $expression [, mixed $message]);
```

The `assert` Function

- As of PHP 7 the `php.ini` file includes two additional configurations. The `zend.assertions` and the `assert.exception` settings.
- The `zend.assertions` can be assigned with three possible values.
- The `assert.exception` can be assigned with two possible values.

The `assert` Function

- Assigning `zend.assertions` with `'1'` fits the development phase. Additional code will be generated and if the assert condition is false an exception will be thrown.
- Assigning `zend.assertions` with `'-1'` fits the production phase. There won't be any additional code generated and there won't be any exception thrown if the condition false. There won't be any price in performance.

The `assert` Function

- Assigning `zend.assertions` with `'0'` means that additional code for throwing the AssertionError will be generated, but it won't be executed.

The `assert` Function

- Assigning `assert.exception` with the value '`1`' means that when the assertion condition fails then an exception will be thrown.
- Assigning `assert.exception` with the value '`0`' means that when the assertion condition fails nothing will happen.

The `assert` Function

```
<?php
ini_set('assert.exception', 1);

function calculateMagicalNumber($number) {
    assert(false,"number cannot be 13");
    //...
    return 7;
}

$temp = calculateMagicalNumber(13);
echo $temp;
?>
```