PHP Exception Handling

How to handle and create user-defined exceptions

Mario Peshev

Technical Trainer

http://peshev.net/

Software University

http://softuni.bg

```
try {
        // some code that may fail
} catch (Exception $e) {
        echo 'This code failed';
}
```

Table of Contents

Introduction to Exceptions

- Exceptions are used to change the normal flow of a script if a specified error occurs.
- Exceptions are error messages that:
  - Allow part of application to notify the rest that there is a problem.
  - Are used when the error does not allow the application or part of it to continue execution.

PHP Exceptions

- PHP doesn't throw any exceptions by default
  - Example: division by zero produces only warning
  - The program continues execution and presents incorrect result
- However PHP provides inbuilt engine for handling errors and warnings and turning them in exceptions
- PHP incorporates the **`Exception`** class to handle exceptions

Error Reporting Level

- PHP has many levels of errors and warnings
  - You can configure which are printed to the browser or the log files
  - The error reporting level is integer representing bit fields
    - Can be defined as Boolean operations between constants

Level Examples

- **E_ERROR** (level 1) – Fatal run-time errors. Errors that can not be recovered from. Execution of the script is halted;
- **E_WARNING** (level 2) – Non-fatal run-time errors. Execution of the script is not halted;
- **E_NOTICE** (level 8) – Run-time notices. The script found something that might be an error or might not;

Level Examples

- **E_STRICT** (level 2048) – Run-time notices. PHP suggest changes to your code to help interoperability and compatibility of the code;
- **E_RECOVERABLE_ERROR** (level 4096) – Catchable fatal error. This is like an E_ERROR but can be caught by a user defined handle;
- **E_ALL** (level 8192) – All errors and warnings, except level E_STRICT (E_STRICT will be part of E_ALL as of PHP 6.0);

Error Reporting Level

- Error reporting level can be set in the **php.ini** or at runtime with the **error_reporting** function
  - Takes one parameter – the desired error level

```
error_reporting(E_ALL & ~E_NOTICE);
```

  - **E_ALL & ~E_NOTICE** means all errors except those of level "**E_NOTICE**"
    - This is the default level

Catching an Exception

- Exceptions can be caught and some code executed

```
try {
        // some code that may fail
} catch (Exception $e) {
        echo 'This code failed';
}
```

- Exception, raised in the **`try`** block is caught (catched)
- Each **`try`** block must have at least one **`catch`** block
- Different **`catch`** blocks may correspond to different classes of exceptions
  - In this example the catch block will intercept all types of exceptions

Catching an Exception - Example

```
$a = 5;
$b = 0;
try {
        if ($b == 0) {
                throw new Exception("You cannot divide by zero");
        } else {
                $i = $a/$b;
                echo $i;
        }
} catch (Exception $e) {
        echo $e->getMessage();
}
```

Catching Exceptions

- The exceptions, matched by a `catch` block, are stopped
  - The rest of the application continues working
  - Exceptions can be re-thrown
  - If exception is not caught, a PHP Fatal Error is issued and execution stops
  - When exception is raised in a `try` block, the rest of the block is not executed
  - `Try-catch` blocks can be nested

The Exception class

- The Exception class has several useful methods
- `getMessage()` – returns user friendly message, explaining the error
- `getCode()` – returns integer code, usually specifying the error and is useful to distinguish exceptions
- `getFile()`, `getLine()`, `getCode()` – return where the exception occurred
- `getTrace()`, `getTraceAsString()` – return the trace data as array or string and is useful to log the data about the exceptions

The Exception class - Example

- The methods provided by the Exception class can be used to notify the user.

```
function checkNum($number) {
        if($number>1) {
            throw new Exception("Value must be 1 or below", 999);
        }
        return true;
}
try {
        checkNum(2);
} catch (Exception $e) {
        echo $e->getMessage()."<br/>"." code: ".$e->getCode();
}
```

Creating Custom Exceptions

- Creating custom exception is as simple as creating object of class **`Exception`**

```
throw new Exception("Value must be 1 or below", 999);
```

  - Creating object of class exception does not mean it is thrown
  - Constructor has two parameters – message and optional error code

- Exceptions are thrown with the **`throw`** operator

```
throw new Exception("You cannot divide by zero");
```

Throwing Multiple Exceptions - Example


```
try {
  if (!$_POST['name'])
    throw new Exception('No name supplied', 1001);
  if (!$_POST['email'])
    throw new Exception ('No email supplied', 1002);
  if (!mysql_query("insert into sometable values
('".$_POST['name']."', '".$_POST['email']."'"))
    throw new Exception ('Unable to save!', 1003);
} catch (Exception $e) {
  $code = $e->getCode();
  if ($code > 1000 && $code < 1003)
    echo "Please fill in all the data";
  elseif ($code == 1004)
    echo "Database error or unescaped symbols!";
  else {
    throw $e; // re-throw the exception!
  }
}
```

Extending the `Exception` class

- Extending the **`Exception`** class is highly recommended
  - Allows usage of multiple catch blocks for different classes of exceptions, instead of distinguishing them by their code
  - Each exception class can have predefined error and code
    - No need to set when throwing, constructor may be without parameters
    - Methods of the class may contain more functionality

Exception Extending – Examples

- Using extensions of the **Exception** class is no different than simply extending any other class

```
class EMyException extends Exception {
    public function __construct() {
        parent::__construct('Ooops!', 101);
    }
}

try {
    …
    Throw new EMyException();
} catch (EMyException $e) {
    echo "My exception was raised!"."<br/>";
    echo $e->getMessage()." / code: ".$e->getCode();
}
```

Exception Extending – Examples

- Example with multiple catch blocks

```
try {
        $a = 5;
        $b = 2;
        $i = $a/$b;
        throw new EMyException();
} catch (EMyException $e) {
        echo 'My exception was raised';
} catch (Exception $e) {
        echo 'You cannot divide by zero';
}
```

- Much better than having single **catch** with complex code to handle different types of exceptions

Global Exception Handlers

- `set_exception_handler($callback)` – sets the function, specified by `$callback` as exception handler
  - All exceptions, not stopped by `try…catch` constructs are sent to this function

```
function ex_handler ($exception) {
  echo "Uncaught: ".$exception->getMessage();
}
set_exception_handler('ex_handler');
throw new Exception ('boom');
echo 'this line is not executed';
```

Global Warnings Handler

- Warnings are different from exceptions
  - They are recoverable – engine can continue execution
  - Different function for settings global handler
- `set_error_handler` – similar to `set_exception_handler`
  - The callback function gets other parameters
  - Can be used to convert warnings into exceptions
  - Optional second parameter defines the level of errors caught

Impossible to Catch Errors

- There are errors that cannot be caught with `Exception` class, `set_exception_handler` or `set_error_handler`;
  - Parse errors
  - Fatal errors
  - Core errors
- Example: calling a function that does not exist:

```
$a = 2;
$b = 3;
multiply_two(a,b);
```

`die` Function

- The `die` function is commonly used alias of `exit` function
  - Stops execution of the program
  - Takes one optional parameter – string or integer status
  - If status is string, it is printed before exiting
  - Exiting with status 0 means program finished successfully
    - Any other status means error

die Function – Example

- die is commonly used this way:

```
mysql_connect(…) or die ('Unable to connect DB server');
```

  - If mysql_connect fails, it returns false
    - PHP continues with execution of the next statement to evaluate the logical expression
  - If mysql_connect succeeds, PHP does not execute the other statement
    - The logical result will be true anyway
  - Same can be done with if
    - The "or" approach is inherited by Perl and many developers prefer it

The @ operator

- Expressions in PHP can be prefixed with @
  - Error control operator
  - If error occurs during operation execution, it is ignored
  - Can be used with function calls, variables, include calls, constants, etc.
  - Cannot be used with function and class definitions, conditional statements, etc.

```
@mysql_connect(…);
$res = @file ('no_such_file') or die ('…');
$value = @$my_array['no_such_key'];
$value = @2 / 0;
```

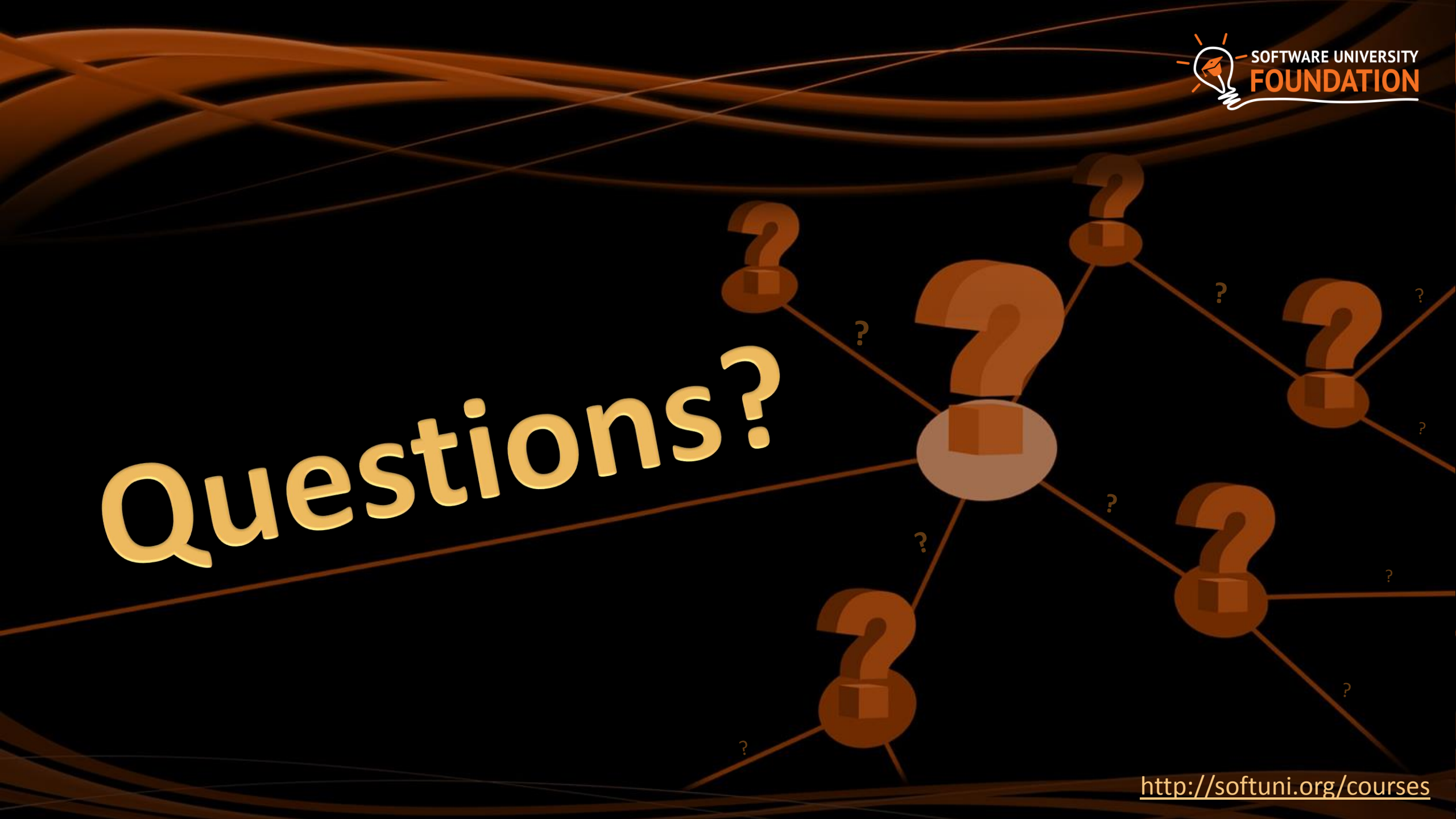
The @ Operator

Summary

- Try-catch Construct
- The Exception Class
- Throwing Exceptions
- Creating Custom Exceptions
- Global Exception Handlers
- Die Function
- Setting the Level of Output
- The @ Operator

SOFTWARE UNIVERSITY FOUNDATION

Questions?



http://softuni.org/courses

License

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

- Attribution: this work may contain portions from
  - "Fundamentals of Computer Programming with C#" book by Svetlin Nakov & Co. under CC-BY-SA license
  - "C# Part I" course by Telerik Academy under CC-BY-NC-SA license

Free Trainings @ Software University

- Software University Foundation – softuni.org
- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University @ YouTube
  - youtube.com/SoftwareUniversity
- Software University Forums – forum.softuni.bg