PHP Error and Exception Handling
Lecture-4

By: Ashiqullah Alizai
Herat University
Computer Science Faculty
0795642400
Alizai.csf@hotmail.com

PHP Errors

- **ERRORS ARE A FACT OF LIFE. Mr. Murphy has an entire collection of laws detailing the** prevalence and inescapability of errors.
- In programming, errors come in two basic flavors
  - **External errors—These are errors in which the code takes an unanticipated path** due to a part of the program not acting as anticipated.
    - For example, a database connection failing to be established when the code requires it to be established successfully is an external error.
  - **Code logic errors—These errors, commonly referred to as *bugs, are errors in*** which the code design is fundamentally flawed due to either faulty logic ("it just doesn't work that way") or something as simple as a typo.

External vrs logical errors

- These two categories of errors differ significantly in several ways:
- External errors will always occur, regardless of how "bug free" code is.
- They are not bugs in, and of themselves because they are external to the program.
- External errors that aren't accounted for in the code logic can be bugs.
  - For example, blindly assuming that a database connection will always succeed is a bug because the application will almost certainly not respond correctly in that case.
- Code logic errors are much more difficult to track down than external errors because by definition their location is not known. You can implement data consistency checks to expose them, however.

Using PHP die() Function

- While writing your PHP program you should check all possible error condition before going ahead and take appropriate action when required.
- The die() function is commonly used to Stop execution of the program
- It takes one optional parameter –
  - string : If status is string, it is printed before exiting the user defined message
  - integer status
  - Exiting with status 0 means program finished successfully
  - Any other status means error

PHP die Function

```
<?php
     if(!file_exists("/tmp/test.txt"))
    {       die("File not found");
    }
else{       $file=fopen("/tmp/test.txt","r");
      print "Opend file sucessfully";
    }
 ?>
```

Using PHP exit() Function

- Similar to php die function

```
//Output a message and terminate the current script

Context:

void exit ([ string $status ] )

void exit ( int $status )

<?php

    if(!file_exists("/tmp/test.txt"))

   {       exit("File not found");

   }

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

        print "Opend file sucessfully";

   } ?>
```

PHP built-in support for error handling

- PHP has built-in support for error handling, as well as a built-in severity system that allows you to see only errors that are serious enough to concern you.
- PHP has three severity levels of errors:
  1. E_NOTICE
  2. E_WARNING
  3. E_ERROR

Informational Errors(E_Notice)

- Harmless problem, and can be avoided through use of explicit programming.
- E_NOTICE errors are minor, nonfatal errors designed to help you identify possible bugs in your code.
- In general, an E_NOTICE error is something that works but may not do what you intended.
- An example might be using a variable in a non-assignment expression before it has been assigned to, as in this case:

```
<?php
   $variable++;
?>
```

This example will increment $variable to 1 (because variables are instantiated as 0/false/empty string), but it will generate an E_NOTICE error.
e.g. use of an undefined variable, defining a string without quotes, etc.

Informational Errors(E_Notice)

- This check is designed to prevent errors due to typos in variable names.
- For example, this code block will work fine:

```
<?
    $variable = 0;
    $variabel++;
?>
```

However, $variable will not be incremented, and $variabel will be. E_NOTICE warnings help catch this sort of error;

- In PHP, E_NOTICE errors are turned off by default because they can produce rather large and repetitive logs. In my applications,
- It is preferred to turn on E_NOTICE warnings in development to assist in code cleanup and then disable them on production machines.

E_WARNING

- E_WARNING errors are nonfatal runtime errors.
- They do not halt or change the control flow of the script, but they indicate that something bad happened.
- Many external errors generate E_WARNING errors.
  - An example is getting an error on a call to fopen(),mysql_connect().

Fatal Errors E_ERROR

- Something so terrible has happened during execution of your script that further processing simply cannot continue.
  - e.g. parsing error, calling an undefined function, etc.
- E_ERROR errors are unrecoverable errors that halt the execution of the running script.
  - Examples include attempting to instantiate a non-existent class and failing a type.

Error Levels

- In addition to these errors, there are five other categories that are encountered some what less frequently:
- E_PARSE—The script has a syntactic error and could not be parsed .This is a fatal error.
- E_COMPILE_ERROR—A fatal error occurred in the engine while compiling the script.
- E_COMPILE_WARNING—A nonfatal error occurred in the engine while parsing the script.
- E_CORE_ERROR—A fatal runtime error occurred in the engine.
- E_CORE_WARNING—A nonfatal runtime error occurred in the engine.
- E_ALL error category for all error reporting levels.
- You can control the level of errors in php.ini error_reporting = E_ALL to some thing else

Error Reporting

- error_reporting() function: This function controls which levels of error will be displayed, and can be set from none to all.
  - int error_reporting(int [level]);
- error_reporting(0);: Turn off the error reporting
- To find the value, we add together the values for the error levels we wish to display.
- error_reporting() with the setting of 7 (1+2+4, or fatal errors, warnings and parse errors).
- To set the error reporting for our code to the maximum, we must add all error level.
- It is advisable to set error reporting to the maximum value of 15 while writing and debugging PHP

Possible error level

Value	Constant	Description
1	E_ERROR	Fatal run-time errors. 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 that occur during PHP's initial startup.
32	E_CORE_WARNING	Non-fatal run-time errors. This occurs during PHP's initial startup.
256	E_USER_ERROR	Fatal user-generated error. This is like an E_ERROR set by the programmer using the PHP function trigger_error()
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()
1024	E_USER_NOTICE	User-generated notice. This is like an E_NOTICE set by the programmer using the PHP function trigger_error()
2048	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.
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())
8191	E_ALL	All errors and warnings, except level E_STRICT (E_STRICT will be part of E_ALL as of PHP 6.0)

Set error reporting settings

```
<?php
// Turn off all error reporting
error_reporting(0);

// Report simple running errors
error_reporting(E_ERROR | E_WARNING | E_PARSE);


// Reporting E_NOTICE can be good too (to report uninitialized
// variables or catch variable name misspellings ...)
error_reporting(E_ERROR | E_WARNING | E_PARSE | E_NOTICE);


// Report all errors except E_NOTICE
error_reporting(E_ALL ^ E_NOTICE);

// Report ALL PHP errors
error_reporting(E_ALL);
?>
```

Handling Errors

- Now that you've seen what sort of errors PHP will generate, you need to develop a plan for dealing with them when they happen.
- PHP provides four choices for handling errors that fall within the error_reporting threshold:
  1. Display them.
  2. Log them.
  3. Ignore them.
  4. Act on them.

Displaying Errors

- When you opt to display errors, an error is sent to the standard output stream
- In the case of a Web page means that it is sent to the browser.
- You toggle this setting on and off via the php.ini setting:
  - display_errors = On

Display errors

- Display errors is very helpful for development because it enables you to get instant feedback on what went wrong with a script without having to tail a logfile or do anything but simply visit the Web page you are building.
- Displaying PHP errors to an end user is usually undesirable for three reasons:
  - It looks ugly.
  - It conveys a sense that the site is buggy.
  - It can disclose details of the script internals that a user might be able to use for nefarious purposes.

Logging Errors

- PHP internally supports both logging to a file and logging via syslog via two settings in the php.ini file.
- This setting sets errors to be logged:
  - log_errors = On
- And these two settings set logging to go to a file or to syslog, respectively:
  - error_log = /path/to/filename
  - error_log = syslog
- Logging provides an auditable trace of any errors that transpire on your site.

Logging Errors

- In addition to the errors logged from system errors or via trigger_error(), you can manually generate an error log message with this:
  - error_log("This is a user defined error");
- Alternatively, you can send an email message or manually specify the file.
  - See the PHP manual for details.
- error_log logs the passed message, regardless of the error_reporting level that is set;
- error_log and error_reporting are two completely different entries to the error logging facilities.

Ignoring Errors

- PHP allows you to selectively suppress error reporting when you think it might occur
- The special @ operator can be used to suppress function errors.
- Any error produced by the function is suppressed and not displayed by PHP regardless of the error reporting setting.
- For example, if you want to open a file that may not exist and suppress any errors that arise, you can use this:

```
$fp = @fopen($file, $mode);
$db = @mysql_connect($h,$u,$p);
if (!$db) {
trigger_error('blah',E_USER_ERROR);
}
```

Acting on Erorrs

- Generally, how PHP handles errors is defined by various constants in the installation (php.ini).
- You can write your own function to handle PHP errors in any way you want.
- You simply need to write a function with appropriate inputs, then register it in your script as the error handler.
- The handler function should be able to receive 4 arguments, and return true to indicate it has handled the error.
- The 4 arguments are:
  - **error_code**
  - error_message,
  - error_file,
  - error_line

Error structucre

- error_message: Specifies the error message for the user-defined error
- error_file: Specifies the filename in which the error occurred
- error_line: Specifies the line number in which the error occurred
- error_context: Specifies an array that points to the active symbol table at the point the error occurred.
  - In other words, error_context will contain an array of every variable that existed in the scope the error was triggered

Custom Error Handler Function

```
function err_handler(
   $errcode,$errmsg,$file,$lineno) {

   echo 'An error has occurred!<br />';
  echo "file: $file<br />";
  echo "line: $lineno<br />";
  echo "Problem: $errmsg";
   return true;
}
```

Custom Error Handler Registration

- The function then needs to be registered as your custom error handler:
  - `set_error_handler('err_handler');`
- You can 'mask' the custom error handler so it only receives certain types of error. e.g. to register a custom handler just for user triggered errors:

  -
    ```
    set_error_handler('err_handler',
    E_USER_NOTICE | E_USER_WARNING | E_USER_ERROR);
    ```

Restriction to Custom Error Handler

- A custom error handler is never passed **E_PARSE**, **E_CORE_ERROR** or **E_COMPILE_ERROR** errors as these are considered too dangerous.
- Often used in conjunction with a 'debug' flag for neat combination of debug and production code display.

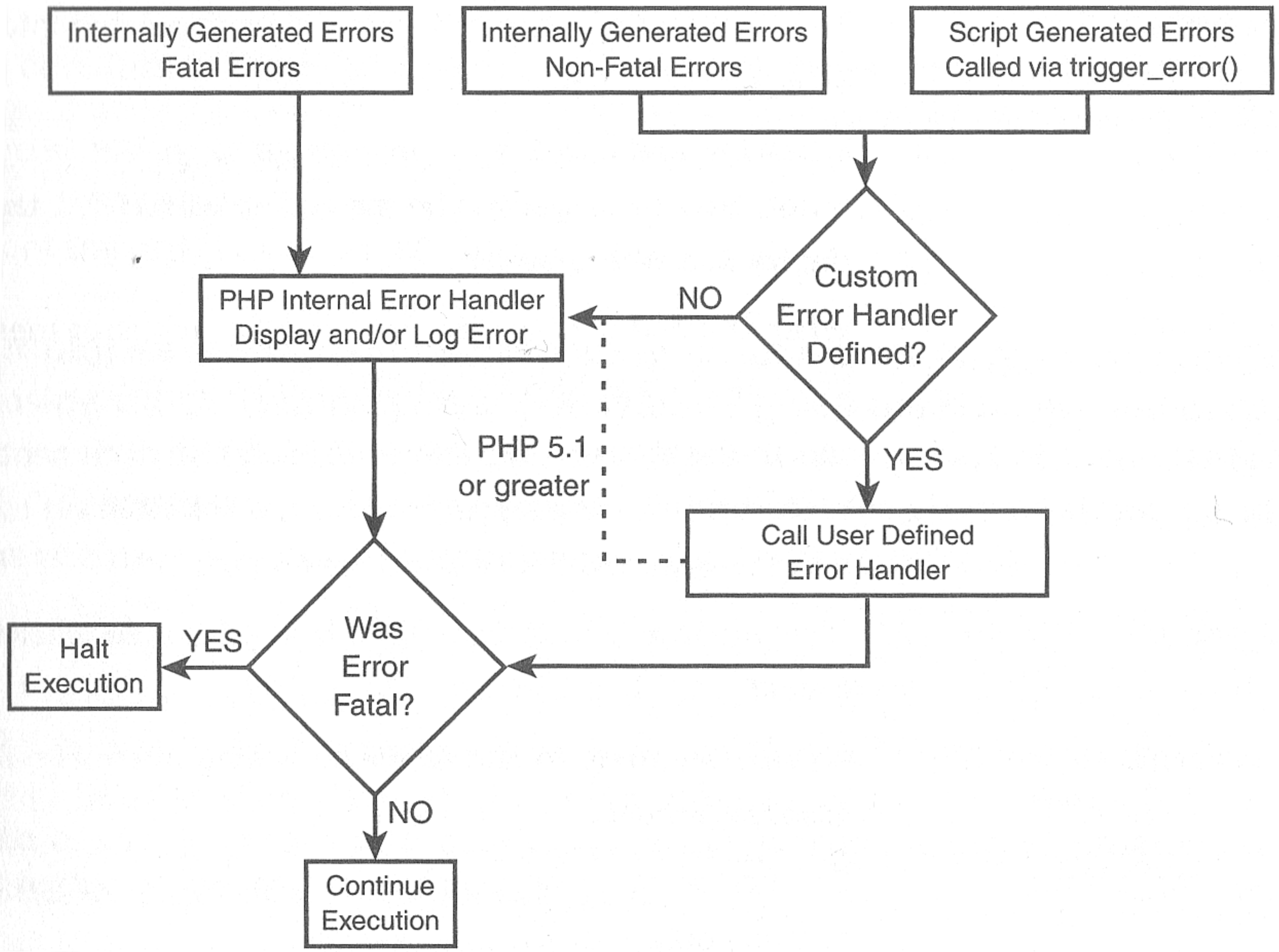
Triggering the error

- PHP supplies the trigger_error() function, which allows a user to generate his or her own errors inside a script.
- There are three types of errors that can be triggered by the user, and they have identical semantics to the errors just discussed:
  - E_USER_NOTICE
  - E_USER_WARNING
  - E_USER_ERROR

You can trigger these errors as follows:

```
while(!feof($fp)) {
$line = fgets($fp);
if(!parse_line($line)) {
trigger_error("Incomprehensible data encountered", E_USER_NOTICE);
}
}
```

PHP ErrorHandling



Exception Handling

Introduction to Exceptions

- Remember from previous section the warning that provided by php allow the application to continue working
  - For example: if a Program required a file that store the password and when if the file not exist what will happen?.
  - the program will throw only a warning and than will continue to execution of other parts.
- PHP doesn't like to throw exceptions
  - Example: division by zero produces only warning
  - The program continues execution, even the result may be incorrect
  - Warnings can be converted in Exceptions
  - PHP 5 has an exception model similar to that of other programming languages.

Introduction

- Exceptions are error messages that halt the program
  - PHP provides builtin engine for handling errors that allows part of application to notify the rest that there is a problem
  - Usually used when the error does not allow the application or part of it to continue execution
- Exceptions are important and provides a better control over error handling.
- **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.

Catching an Exception

- Example of exception is division by zero
- Can be caught and some code executed
- 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

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

Catching an Exception

- 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
- Useful to distinguish exceptions
- getFile(), getLine(), getCode() – return where the exception occurred
- getTrace(), getTraceAsString() – return the trace data as array or string
- Useful to manually log the data about the exception

Example of Using the Exception Class

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

```
<?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';
?>
```

Second Example loging exception

- The methods provided by the Exception class can be used to notify the user or log the error

```
try {
    // some code that may fail
} catch (Exception $e) {
    $error_text = 'An error has occurred: '.
        $e->getMessage()." (code: ".
        $e->getCode().")";
    echo $error_text;
    system ("echo ".$error_text.
        " >> /var/log/myerrors.log");
}
```

Creating Custom Exceptions

- Creating custom exception is as simple as creating object of class Exception
  - *new Exception ('Something went wrong', 7);*
- 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('Oops!');*

Throwing Exceptions

```
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 – Example

- 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 {
   …
} catch (EMyException $e) {
   echo "My exception was raised!";
}
```

Multiple Catch Blocks

- Example with multiple catch blocks

```
try {
   $a = 5;
   $b = 0;
   $i = $a/$b;
   throw new EMyException();
} catch (EMyException $e) {
   echo 'My exception was raised';
} catch (Exception $e) {
   echo 'You cannot divide by zero';
}
```

Cascading Exceptions

- Sometimes you might want to handle an error but still pass it along to further error handlers.
- You can do this by throwing a new exception in the catch block:

```
<?php
    try {
        throw new Exception;
    }
    catch (Exception $e) {
        print "Exception caught, and rethrown\n";
        throw new Exception;
    }
?>
```

The catch block catches the exception, prints its message, and then throws a new exception. In the preceding example, there is no catch block to handle this new exception, so it goes uncaught.

Installing a Top-Level Exception Handler

- An interesting feature in PHP is the ability to install a default exception handler that will be called if an exception reaches the top scope and still has not been caught.
- This handler is different from a normal catch block in that it is a single function that will handle *any uncaught exception, regardless of type (including exceptions that do not inherit* from Exception).

When to Use Exceptions

- There are a number of views regarding when and how exceptions should be used.
- Some programmers feel that exceptions should represent fatal or should-be-potentially-fatal errors only.
- Other programmers use exceptions as basic components of logical flow control.
- The Python programming language is a good representative of this latter style: In Python exceptions are commonly used for basic flow control.

When to Use Exceptions

- This is largely a matter of style, In deciding where and when to use exceptions in your own code, you might reflect on this list of caveats:
- Exceptions are a flow-control syntax, just like if{}, else{}, while{}, andforeach{}.
- Using exceptions for nonlocal flow control (for example, effectively long-jumping out of a block of code into another scope) results in non-intuitive code.
- Exceptions are bit slower than traditional flow-control syntaxes.
- Exceptions expose the possibility of leaking memory.

This is the end for this lecture