TASK

PHP - Error Handling

Visit our website

Introduction

HANDLING PROBLEMS BEFORE THEY OCCUR

Certain methods or operations in programming may result in an error under specific circumstances (like attempting to access the 5th element of an array that has 3 elements). But some of these kinds of operations have unpredictable situations in which they could fail — e.g. the network going down during a GET request, or the hard drive failing during a file read. While you may think some errors aren't worth making provisions for (because of their supposed rarity), it is always good programming practice to cover all possible edge cases rather than leaving potential for your program crashing. In this task, we'll cover best practices for taking care of potential errors in your PHP code.

THE GLORIOUS EXCEPTION

In many programming languages, including Java and PHP, an exception is a circumstance whereunder a certain method or operation fails. Examples include non-existent array index accesses, file permission errors, network transfer failures, and database query syntax errors, to name a few. Luckily, PHP provides native support for handling these errors gracefully and allowing the programmer to specify what should happen in these cases, as opposed to having the program crash.

PHP built-in functions are designed in most part not to raise any exceptions, but just to fail silently. For example, when failing to read a file, the `fread()` function returns false instead of throwing an exception. This means it is mostly up to the developer to foresee exceptions and handle them appropriately. Let's look at how to raise an exception explicitly:

```php
function get_index($arr, $i){
   if ($i >= 0 and $i < sizeof($arr))
       return $arr[$i];
   else
       throw new Exception('Index out of bounds!');
}
```

This function will throw an exception when the index is out of bounds for the array's size. The Exception's default constructor takes a message which should describe the reason for failure.

HANDLING EXCEPTIONS

To handle an expression you must use a try-catch block, like so:

```
try {
   echo get_index(['a'. 'b', 'c'], 5);
}catch (Exception $e){
   echo 'Caught error: ' . $e->getMessage();
}
```

The code inside the try block will be executed, and if any of the statements produce an exception, the code in the catch block will immediately execute with the relevant exception's details. In this case, **`get_index`** will fail (index 5 with array size 3), and the catch-block will execute before anything in the try-block can echo.

The exception object has many useful methods that allow access to information about the exception, such as which script and what function caused it, but for our purposes, just the message suffices.

As we discussed in an earlier task, when PHP encounters an unhandled exception, it immediately stops executing the script. This could result in a blank response or, if some output was already generated, the result is a half-rendered webpage. The catch block allows us to tell the user something went wrong, to try again, or to organise an alternative solution.

Exceptions are named as such because they really should only occur in exceptional cases. In the example above it occurs all the time, so it is inappropriate to use an exception there. What should instead be done is a check to see if the index is within the array's allowed indexes before calling the **`get_index`** method.

CUSTOM EXCEPTIONS

When throwing an exception, you may wish to add more information to it than just a plaintext description. For instance, suppose you're creating a set of functions that is responsible for loading settings from a file. If there is a problem reading the file you may want to store the problematic filename. You can create a custom exception by extending the Exception class.

```
class IOException extends Exception {
    private $filename;
```

```php
    public function __construct($message, $filename) {
        parent::__construct($message);
        $this->filename = $filename;
    }

    public function getFilename() {
        return $this->filename;
    }
}
```

To use the exception, create it like any other object and throw it like an Exception.

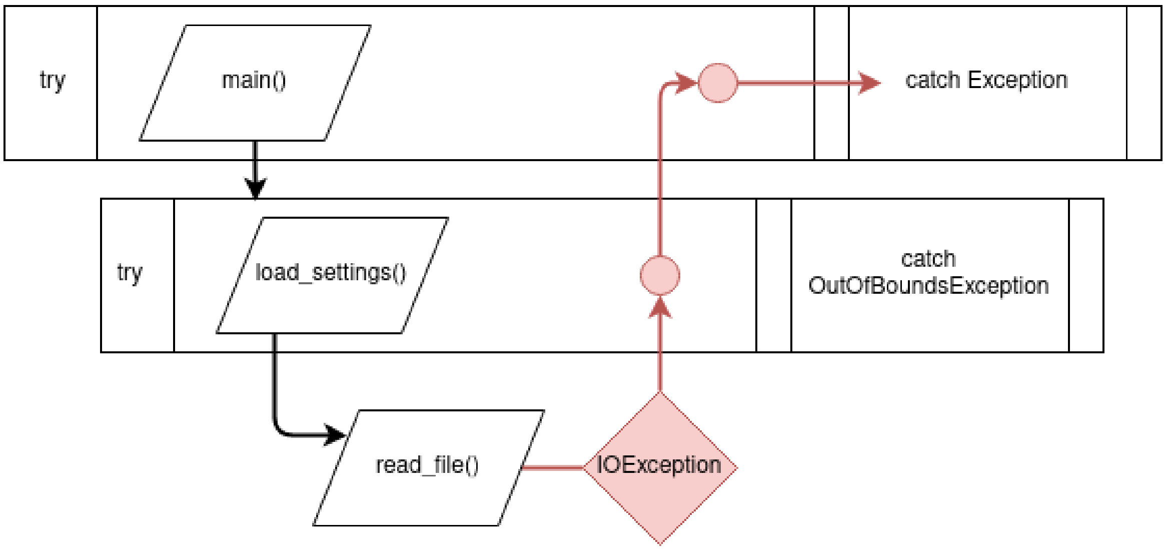
```php
function read_file($the_file) {
   if (file_exists($the_file)) {
       // read it
   } else {
       throw new IOException('Unable to find file.', $the_file);
   }
}

try{
   read_file('some_file.txt');
}catch (IOException $e){
   echo $e->getMessage() . $e->getFilename();
}
```

Note that the catch-block specifically asks for IOException, not Exception. This will catch IOExceptions (and any derivatives), but not any other kind of exceptions. If you want to catch any exception, you specify the Exception class. You can also have multiple catch blocks, handling different Exceptions differently.

```php
try{
   read_file('some_file.txt');
}catch (IOException $e){
   echo $e->getMessage() . $e->getFilename();
}catch (DivisionByZeroError $e){
   echo 'Read up on the laws of mathematics';
}catch (ParseError $e){
   echo 'Could not understand the file contents';
}catch (Exception $e){
   echo 'Unknown error';
}
```

The terms “throw” and “catch” are very appropriate because exceptions really do not travel through your program the way normal variables do. Instead of going line-by-line, exceptions get thrown “up” to the function caller and keep getting thrown up until there is an appropriate catch block ready to deal with the specific exception. If there is none, the program crashes.



Here, IOException is caught by the `catch Exception` block, because IOException is a derivative of Exception.

LOGICALLY HANDLING EXCEPTIONS

Once you catch an exception, how you deal with it is up to you. Sometimes the error is of a nature your program cannot recover from, in which case your best bet is to echo a descriptive message to the user.

Sometimes, however, the error could be caused by your specific approach to a problem, and thus trying another approach may solve the problem without needing to halt the program. For instance, suppose you’re trying to read a file. You first try to interpret it in ASCII, and if that doesn’t work, try UTF-8.

```
foreach(['ASCII', 'UTF-8'] as $enc){
   try{
       read_file('some_file.txt', $enc);
       // this break statement will only fire if
       // the read_file worked.
       break;
   }catch (EncodingException $e){
       // try next encoding
       continue;
   }catch (Exception $e){
       echo 'Non-encoding error occurred.';
       break;
   }
}
```

Here we use a for-each loop to try all the encodings. If the read was successful, or if a non-encoding exception occurs, the loop stops. If this were your implementation you would need to devise some way to distinguish between the two cases — perhaps by creating a boolean before the loop and setting it appropriately inside.

If an encoding exception occurs (which is a synthetic exception we create and throw inside the `read_file` function) then the loop simply continues to the next encoding.

This same logic could be written by using a try-catch block in another try-catch block. I.e. if trying X fails, try Y in X's catch block. This is bad for code readability and often causes duplicate code (e.g. handling the non-encoding exception the same way in both cases). We recommend using loops or functions where it is expected that very similar logic will run.

The final handling logic we'll look at is simply trying again. The best example of where this may be appropriate is with network requests. Suppose we have a method that makes a request to an external server, and if it fails, it throws a NetworkException.

```
$success = false;
for($i = 0; $i < 3; $i++){
    try{
        external_request();
        $success = true;
        break;
    }catch (NetworkException $e){
        //try again
        continue;
    }catch (Exception $e){
        echo 'Unexpected exception';
        break;
    }
}
```

This loop's behaviour is very similar to the previous loop's, but it does nothing different each iteration — it simply tries the same thing thrice. Here we've set a boolean called **`$success`** that's set to true only if the external request succeeded. After the loop, you can check its value to see whether the networking worked or not.

HIDDEN EXCEPTIONS

As mentioned earlier, most of PHP's built-in functions fail silently — that is, they return special values upon failure instead of throwing an exception. Here are some examples from functions you've seen before.

```
$success = setcookie('name', 'val', time()+120, '/');
if ($success === false)
   echo 'failure setting cookie';

$success = session_start();
if ($success === false)
   echo 'failure starting session';

$index = strpos('haystack', 'x');
if ($index === false)
   echo 'Could not find "x" in "haystack"';
```

Whenever you're working with a new built-in function, it's a good idea to check the **PHP docs** to see how it fails.

Take note:

Keeping exceptions exceptional
PHP does support throwing and catching exceptions, but this is really only to better support OOP. The reason most PHP built-ins don't throw them is because PHP was originally purely procedural.

Because of this, there are many diagnostic functions that help the developer prevent failure cases before they occur. For instance, not only can you use **`fread()`** to read a file, PHP also provides **`file_exists()`**, **`filesize()`**, and **`fileperms()`** functions, all to help the developer check for potential errors before reading the file.

Where possible, we recommend doing as many preliminary checks as possible before doing a risky operation. This allows you better control over errors and shows your ability to predict and resolve problems before they occur. Sometimes using Exceptions is indeed the most elegant solution to error handling, and in these cases, feel free to do so.

Compulsory Task 1

- Create a file called **home.html**, in which you have an html form that takes a number. Create a handling script called **maths.php**. Create the following functions in it:
  - `is_positive()` - checks that a number is positive.
  - `is_divisible()` - checks that a number is divisible by five.
  - `enough_digits()` - checks that the number has at least 4 digits.
- Each of these functions should not return anything but should throw a custom exception each if their respective condition isn't met.
- Have one try block and a connected catch block for each of the potential exceptions. Handle each exception by echoing the respective error data in formatted HTML.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.