**OOP:**

**Using Existing Classes**

**Catching expressions**

PHP has the *try* and *catch* statements for handling exceptions. The premise is that you *try* to do certain things in your PHP code, specifically the kind of things that might fail (like connecting to a database or including a file). If an error occurs, an exception is *thrown*. Your code will then *catch* the exception and respond accordingly.

Try{

//do something

}catch(exception){

//do whatever else

}

//this is a more sophisticated version of: if(doSomething){}else{//do whatever else}

One benefit that exception has over the conditional is that it further separates the functionality and logic from the error handling. Furthermore, multiple errors can be handled without having to use lots of nested conditionals.

Exceptions are thrown using the syntax: throw new Exception(‘error message’); This code throws an object of type *Exception,* a class defined in PHP. To catch this exception, you would have: catch(exception $e) – where $e is an object of the *Exception* type.

The exception class contains the following methods:

|  |  |
| --- | --- |
| getCode() | The numeric exception code received, if any |
| getMessage() | The string exception message received, if any |
| getFile() | The name of the file where the exception occurred |
| getLine() | The line number from which the exception was thrown |
| getTrace() | An array of information, like the file name, line number and so on |
| getTraceAsString() | The same information as getTrace() but as a string |
| \_\_toString() | All of the preceding information as a string |

Any code within a *try* block, after an exception is thrown will never run. Also, if no exception ever occurs, the code in the *catch* block will never be executed.

Example:

Class WriteToFile {

//for storing the file pointer

Private $\_fp = NULL;

//constructor opens the file for writing:

Function \_\_construct($file) {

//check that the file exists and is a file:

If(!file\_exists($file) || !is\_file($file)) {

Throw new Exception(‘The file does not exist’);

}

//open the file

If(!$this->\_fp = @fopen($file, ‘w’)) {

Throw new Exception(‘Could not open the file’);

}

}//end of constructor

//this method writes data to the file:

Function write($data) {

//confirm the write:

If(@!fwrite($this->\_fp, $data . “\n”)) {

Throw new Exception(‘Could not write to the file’);

}

}//end of write method

//this method closes the file:

Function close() {

//make sure its open:

If($this->\_fp) {

Fclose($this->\_fp);

$this->\_fp = NULL;

}

}//end of close method

//the destructor calls close(), just in case:

Function \_\_destruct() {

$this->close();

}//end of destructor

}//end of WriteToFile class

//in html file

//load the class definition

Require(‘WriteToFile.php’);

//start the try…catch block

Try{

//create the object

$fp = new WriteToFile(‘data.txt’);

//write the data

$fp->write(‘This is a line of data’);

//close the file

$fp->close();

//delete the object

Unset($fp);

//if we got this far, everything worked!

Echo “<p>The data has been written</p>”;

}catch (Exception $e) {

Echo ‘<p>The process could not be completed because the script: ‘ . $e->getMessage() . ‘</p>’;

}

Echo “<p>This is the end of the script</p>”;

**Using PDO**

PDO stands for PHP Data Objects – it provides an alternative way to interact with a database. It also provides a consistent way to execute queries, regardless of the database application in use.

Connecting to a database

$pdo = new PDO(‘dsn’,’username’,’password’);

The DSN, short for Data Source Name, is the most important part. The DSN is a string that indicates several things:

* The database driver to use
* The database name
* Optionally, the host name
* Optionally, the port

Driver:name1=value1; name2=value2

E.g. $pdo = new PDO(‘mysql:dbname=test; host=localhost’, ‘username’, ‘password’);

//to close the database connection, simply put: unset($pdo); *or* $pdo = NULL;

Catching exceptions

Being a class, PDO does not generate errors but rather throws exceptions. Specifically, PDO throws exceptions of type PDOException.

With this in mind, to interact with the database you’ll want to use a *try…catch* block like:

Try {

$pdo = new PDO(‘mysql:dbname=test; host=localhost’, ’username’, ‘password’);

//execute queries

//unset the object

Unset($pdo);

}catch(PDOException $e) { //report error

Echo ‘<p>An error occurred: ‘ . $e->getMessage() . ‘</p>’;

}

Executing simple queries

Once you’ve established a database connection, and a PDO object, you can run queries on a database.

For simple queries that do not return results (INSERT, UPDATE, DELETE ), use the exec() method:

$q = “DELETE FROM tablename”;

$pdo->exec($q);

For these simple queries, exec() will return the number of rows affected by the query:

$num = $pdo->exec($q);

//use $num

If you just ran an INSERT query and you need to know the dynamically generated primary key value, call the lastInsertId() method:

$id = $pdo->lastInsertId();

To prevent SQL injection attacks, use the quote() method, alternative to mysqli\_real\_escape\_string():

$data = $pdo->quote($unsafe\_data);

Executing SELECT queries

Queries that return results, i.e. SELECT, should be run using the query() method, assigning the results to a new variable:

$results = $pdo->query($q);

To see how many records were returned by the query:

$results->rowCount();

//equivalent to mysqli\_num\_rows()

The $results variable will be an object of type PDOStatement, which is one of the three most important PDO classes, along with PDO and PDOException.

You can fetch results using the fetch() method, first though, you should tell PHP how you want to fetch the records, using setFetchMode(), with a constant:

* PDO::FETCH\_ASSOC
* PDO::FETCH\_NUM
* PDO::FETCH\_OBJ
* PDO::FETCH\_CLASS

Example:

$results = $pdo->query(‘SELECT id, username FROM users’);

$results->setFetchMode(PDO::FETCH\_NUM);

While($row = $results->fetch()) {

//use $row[0] for id

//use $row[1] for userame

}

Prepared statements

Prepared statements are a different way of running queries.

With prepared statements, the query is sent as one step and the specific data is sent separately, as opposed to all being sent as one step. The end result can be much better performance and easier security management: because the data is sent separately from the query, it does not need to be protected against SQL injection attacks.

$stmt = $pdo->prepare(‘SELECT \* FROM users WHERE email=:email AND pass=SHA1(:pass)’);

$stmt->execute(array(‘:email’=>’me@example.com’, ‘:pass’=>’mypass’));

Example:

$q = ‘INSERT INTO tasks (parent\_id, task) VALUES (:parent\_id, :task)’;

$stmt = $pdo->prepare($q);

//confirm results

If($stmt->execute(array(‘:parent\_id’ => $parent\_id, ‘:task’ => $\_POST[‘task’’])) ) {

Echo “<p>The task has been added</p>”;

}else{

Echo “<p>The task could not be added!</p>”;

}

**Using the Standard PHP Library (SPL)**

SPL is a collection of commonly needed tools that can easily be used in your own programs.

File handling

The SPL has defined a few classes for working with files and directories. First, there’s the SplFileInfo class, which creates an object from a file reference:

$file = new SplFileInfo(‘filename.ext’);

Once you have created that object, you can invoke various methods to get information about the file:

* getBasename()
* getExtension()
* getMTime()
* getPathName()
* getSize()
* getType()
* isDir()
* isFile()
* isWritable()

Example:

$file = new SplFileInfo(‘test.php’);

Echo “<p>Extension: {$file->getExtension()}</p>”;

If you also want file manipulation functionality, such as the ability to write or read from a file, you can use SplFileObject. This class inherits from SplFileInfo. Hence, you have all of the previous methods, plus some new ones like fgets() and fwrite(), which correspond to the procedural equivalents.

$file = new SplFileObject(‘somefile.txt’,’r’); //where ‘r’ = mode

//now you can read the files data:

While(!$file->eof()) {

Echo $file->fgets();

}

E.g.

Try{

$fp = new SplFileObject(‘data.txt’,’w’);

//write the data

$fp->fwrite(‘This is a line of data’);

//delete object

Unset($fp);

Echo “<p>The data has been written</p>”;

}catch(Exception $e) {

Echo $e->getMessage();

}

Iterators

Iterator is a design pattern that makes it possible to access the components of any kind of complex data structure using a loop.

There are more than a dozen types of iterators defined in the SPL, including: ArrayIterator, RecursiveArrayIterator, LimitIterator, DirectoryIterator.

E.g. to loop through a directory of files:

$dir = new DirectoryIterator(‘.’);

Foreach($dir as $item){

//use $item

}

*See more examples on p 274-277*

Autoloading classes

With OOP, a logical way to modularise the files in an application is to place each class definition in its own file.

This means requiring them individually when needed which can take up time and space. Thankfully, there is a work around:

1. define a function that will know how to include class files for you:

function class\_loader($class) {

require(‘classes/’ .$class . ’.php’);

}

1. Have PHP call the above function for you when a class definition is required:

Spl\_autoload\_register(‘class\_loader’);

1. For each new object type created in the following code, the function above will be invoked:

$obj = new Class();

$me = new Human();

$r = new Rectangle();

**Test**

Catching exceptions

1. How do you handle exceptions in PHP?
2. How do you *throw* an exception?
3. Give an example of catching expressions

PDO

1. What does PDO stand for, and what does it do?
2. Connect to a database using PDO, catch any exceptions
3. How do you execute simple queries using PDO? Give example
4. How do you prevent SQL injection attacks using PDO queries?
5. How do you execute SELECT queries? Give example
6. What is meant by prepared statements and how does it differ against other database interactions?
7. How does prepared statements protect against SQL injection attacks?
8. Give an example of using prepared statements

SPL

1. What is the SPL?
2. What does the SplFileInfo class do? Give example
3. How can you manipulate file functionality using SplFileObject? Give example
4. How can you autoload classes to avoid including all modules individually?