Notes

* Usually optimal to have one connection.
* It is optimal to connect to the database once.
* We could connect and disconnect every time we needed to query the database but that’s inefficient.
* We could have a global variable assigned to MySQLi connection, but that’s fragile at best, as anything could manipulate it.
* It’s better to control access and restrict the number of connections.
* Static properties can be manipulated after being defined, and the value is stored at a class level.

Reasons for using static method

* The reason is that we cannot anticipate what class should be associated with a row.
* Instead, we can add logic to the method to return the correct class for the row and call load statically.

Commenting with DocBlocks

* The DocBlock commenting style is a widely accepted method of documenting classes.

/\*\*

\* This is a very basic DocBlock

\*/

- DocBlock tags allow developers to define authors of a file, the license for a class, the property or method information, and other useful information.

- The most common tags used follow:

* @author: The author of the current element (which might be a class, file, method, or any bit of code) are listed using this tag.
* @copyright: This signifies the copyright year and name of the copyright holder for the current element.
* @license: This links to the license for the current element.
* @var: This holds the type and description of a variable or class property.
* @param: This tag shows the type and description of a function or method parameter.
* @return: The type and description of the return value of a function or method are provided in this tag.
* A sample class commented with DocBlocks might look like this:

<?php

/\*\*

\* A simple class

\*

\* This is the long description for this class,

\* which can span as many lines as needed. It is

\* not required, whereas the short description is

\* necessary.

\*

\* It can also span multiple paragraphs if the

\* description merits that much verbiage.

\*

\* @author Jason Lengstorf <jason.lengstorf@ennuidesign.com>

\* @copyright 2010 Ennui Design

\* @license http://www.php.net/license/3\_01.txt PHP License 3.01

\*/

class SimpleClass

{

/\*\*

\* A public variable

\*

\* @var string stores data for the class

\*/

public $foo;

/\*\*

\* Sets $foo to a new value upon class instantiation

\*

\* @param string $val a value required for the class

\* @return void

\*/

public function \_\_construct($val)

{

$this->foo = $val;

}

/\*\*

\* Multiplies two integers

\*

\* Accepts a pair of integers and returns the

\* product of the two.

\*

\* @param int $bat a number to be multiplied

\* @param int $baz a number to be multiplied

\* @return int the product of the two parameters

\*/

public function bar($bat, $baz)

{

return $bat \* $baz;

}

}

?>

* Once you scan the preceding class, the benefits of DocBlock are apparent: everything is clearly defined so that the next developer can pick up the code and never have to wonder what a snippet of code does or what it should contain.

Autoloading

* Autoloading can be used to include missing class definitions

Extending

* Can only extend one class’s methods/properties at a time
* Cannot specify multiple classes at a time
* Can extend a class that extends another class
* Use ‘extends’ keyword

Abstracting

* Abstract classes cannot be instantiated
* When extending, abstract methods must also be abstract
* If we define a method as abstract, then any class that extends the abstract class containing that method must declare a method with the same name and arguments.
* If a class has an abstract method, then the class must be abstract
* This solves the problem of the generic address class and provides a mechanism to specify the behavior of any child’s classes.
* Provides behavior of child classes

Approaches to initialization

* Abstract \_\_construct – fragile, unwieldy
* Abstract \_\_init – call from class constructor

Abstract scope

* Methods implementing must have same or less-restrictive scope
* Public must stay public
* Private can be made public
* Prevents unexpected behavior

Common Problem in Development

* When working with multiple kinds of objects that are similar, there can be a lot of details to keep track of.
* As a developer, it makes sense to mitigate these types of issues by ensuring that anything that needs to be initialized is handled properly
* Our first reaction may be to make the construction magic method abstract.
* We can do this, but we will be copying and pasting a lot of code and this can lead to a fragile system where refactoring or adding new functionality becomes cumbersome.
* This is not a best practice and not very object-oriented.
* Instead we can create an abstract method to initialize an object and call that from the abstract class constructor.
* Methods that implement an abstract method will also need to have the same scope or something less-restrictive scope
* If we declare an abstract method as public, we won’t be able to make it private later on.
* But we can make an abstract private method public.

Interface

* Sharing interfaces using polymorphism
* Specifies methods that a class must implement
* But doesn’t say how those methods should be implemented
* Is similar to abstract class
* There are similarities to an abstract class, except that an abstract class can have both scope and methods defined.
* Every method is abstract and public
* A class can implement more than one interface at a time, as long as those interfaces don’t have conflicting method names.
* Interfaces can also have constants, which will be available to any classes that implement it.
* Interface constants cannot be overwritten or defined.
* An interface can extend another interface.
* While an interface is not a class, it’s very similar and can be loaded with the autoloader.

Sharing common API

* The practice of sharing a common application programming interface between classes is known as polymorphism.

Model

* A model is responsible for managing data, storing and retrieving entities and containing business logic.
* This aptly describes what the address is doing what future classes may be doing at a high level.

Why interface

* When developing an application, we are typically working with many different kinds of data in different structures, each with different behavior and needs.
* However, if we were to use different naming conventions for every single structure, your application would descend into chaos.
* The pejorative term spaghetti code is an app description, meaning the source code is impossibly complex and tangled like a pile of pasta.
* To avoid this mess, we can define an object interface.

Spaghetti code is a pejorative phrase for source code that has a complex and tangled control structure, especially one using many GOTO statements, exceptions, threads, or other "unstructured" branching constructs. It is named such because program flow is conceptually like a bowl of spaghetti, i.e. twisted and tangled. Spaghetti code can be caused by several factors, such as continuous modifications by several people over a long-life cycle. Structured programming greatly decreases the incidence of spaghetti code.

Interface

* Interfaces in PHP allow you to define a common structure for your classes.
* Interface cannot be instantiated on its own.
* One of the goals of OOP is re-use of code.
* Interfaces make this a much easier process.
* The interface methods have no internal logic, they are simply a "mapping" or constraint of what the class, or classes, should implement.

Final

* Prevents child classes from overriding a method.
* We can also declare a class as final. If we do, then it cannot be extended.

Overriding

* Is re-declaring method, property or constant
* Usually done in order to define a new or altered behavior.
* Cannot specify stricter
* Overriding a constant just re-declare
* We can’t declare a constant that was declared in an interface.

Overriding a Method

* The new method must have the same number of arguments as the overridden method
* Exception: constructors where we can re-declare more or less arguments.
* Methods also have the ability to call the parent class access methods and properties

Cloning and Comparing Objects

* There are many reasons to do this, including making a backup before performing an operation or as a shortcut for defining a similar object.
* To copy of an object, use the clone keyword.
* The comparison operator, double equals, only checks to see if properties are the same, while the identity operator, tripe equals, will ensure that in addition to the properties, both objects are instances of the same class.

References

* Alias: multiple variables write to the same value
* Objects are different, in that object variables do not contain the actual object itself, only the object’s identifier.

Standard Class

* A standard class is the result of typecasting a value of any type of an object.
* Typecast to an object.
* Have no methods
* Does have values
* If we typecast an object to an object, it will not be modified.
* Object->Object 🡪 no change
* Array -> object
* Keys -> properties
* If we typecast an array to an object, the result will be an object with properties named for the keys of the array, with the values corresponding to the array values.
* Values are array values

Using Standard Classes

* Standard class objects are particularly used in a number of ways:

1. Returning a defined data structure with fixed branches without nesting
2. Strict data structures

* Sometimes is used as shorthand to speed code development as it takes less characters to access a property than it us to specify a key

1. Shorthand

* $array[‘key’] = ‘value’;
* $array->key = ‘value’;

1. Finally, we can use a standard object when interacting with a method that is looking for an object with particular properties, but isn’t actually checking for the class.

Retrieving Objects from the database

1. Standard objects are also created in another context: using a fetch object method in MySQLi
2. The object returned from the database is a standard object.
3. MySQLi can also populate objects if you pass the object name as a parameter, but as there is potentially data from multiple classes in this one table, we will not able to use that.

Error handling with exceptions

1. Exception handling refers to the methodology that responds to the abnormal or exceptional situations that often will change the normal flow of program execution
2. In PHP exceptions are thrown then caught
3. To catch an exception, we surround our code in a try block.
4. To catch: use try
5. If an exception is thrown and not caught, a PHP fatal error will occur.
6. Exceptions are a much more forgiving and logical way of handling an anomalous circumstance.
7. Rather than just halting execution or displaying error messages the program is given a chance to recover
8. To throw: throw new Exception(‘parameter’);
9. Exceptions are thrown, which we’ve done and then caught, which hasn’t been done yet.
10. The program attempts to do something or try and catches the exception
11. Therefore add a try block to the demo before the attempt to load and display the address from the database.
12. The catch block must follow the try block
13. To catch an exception, use the catch keyword, followed by parentheses containing the class or the exception we wish to catch, the same what was thrown, and a variable to put the thrown exception in, if any. Catch(Exception $e)
14. Often exception message is intended for debugging, so consider what is actually being displayed to the user before just displaying an exception message.
15. The exception Message is displayed, which is much more graceful than a fatal exit.
16. To further customize the exception, extend the exception class with us own custom exception subclass.

Design Patterns

* Reusable solutions to common problems within a given context, which are referred to as design patterns.

Singleton Pattern

* A singleton pattern restricts the instantiation of a class to one object in order to coordinate actions throughout the system.
* Example: within the demo database class uses the singleton pattern