Object Oriented PHP



# Lesson 1: Classes and objects

##### New keywords

* class
* new
* ->
* $this

# Introduction

Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Bundling code into individual software objects provides a number of benefits, including: - Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system. - Information-hiding (encapsulation): By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world. - Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code. - Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement.

## Classses and Objects

A class is a source code blueprint for objects. Other analogies are classes provide a factory to create objects with, or are an 'object template'.

Objects are instantiated out of classes by using the new keyword. Another way to think about it is that we instantiate (create instances from) a class in order to assign an instance of the class to a variable.

## The Class Keyword

The class keyword takes a code block that represents an object factory. Inside the code block, properties and methods are created. Objects are instantiated out of classes by using the new keyword. Here is a completely bare Person class:

<?php
class Person {
}
$person = new Person();

The “class” blueprint can specify: - Name: identifies the class. - Properties (variables): contain the attributes of the class. - Methods (functions): contain the behaviors of the class.

We might have a Person class that represents a person. The person might have properties such as a name, a height, a weight, a date of birth, etc. A person might have some ability to walk, which we could represent as a method. We can access properties or methods of the object using the object operator, ->

class Person {
public $name;
function sayHi() {
echo ‘Hi!’;
}
}
$person = new Person();
echo $person->name; // Hi!
echo $person->sayHi(); // Hi!

## $this

Within an object method, $this is a reference to the current object. Example:

class Person {
public $name;
function setName($name) {
$this->name = $name;
}

##### Notes:

Objects are always passed by reference, meaning that a function that accepts an object as a parameter modifies the original object passed to it, rather than making a copy of it.

## Coding Standards:

* Class name: uppercase camels, eg: MyClass
* Class properties/methods: lowercase camels, eg: myProperty
* One class per file.
* No closing ?> tag.

## Exercises

1. Create a “Person” class and instantiate an object of it.
2. Add properties for name, age.
3. Add methods getName(), getAge().
4. Invoke var\_dump() on the person object to inspect it.

# Lesson 2: Visibility

##### New keywords

* public
* private
* protected

## Visibility

One interesting thing to note is that each method and property can have an associated level of visibility, which enables information-hiding. There are three types visibility a property can have are public, private, and protected. Public properties are ones that directly accessible by the instantiated object. Private properties cannot be accessed anywhere except from within the class. This allows one to create a public interface, where some critical internal state can be protected. We will discuss the protected visibility later as it's dependent upon understanding inheritance. Notes: Any methods created without a specific declared visibility default to public. Keep in mind that all non-method properties must be declared public, private, or protected.

class Person {
private $name;
public function setName($name) {
$this->name = $name;
}
public getName() {
return $this->name;
}
}
$person = new Person();
$person->name = 'error'; // This will error out because it's a private property
$person->setName('Alice');
echo $person->getName(); // Alice

## Exercises

1. Expand the previous “Person” class, add setName(), getName() methods.
2. Add a height property, and a gettter and setter.
3. Declare the correct “visibility” for each property and method.
4. Invoke var\_dump() on the person object to inspect it.

# Lesson 3: Constructors and Type Introspection

## New Keywords:

* \_\_construct
* is\_a()

## Constructors

In the previous exercise, we created a class and instantiated an object based on the class. We then manually set the values of properties of the object.

It is often helpful to be able to be able to declare an instance and assign values at the same time.

Fortunately, we can do this with a special *constructor* method. In PHP the constructor method is named \_\_construct. Any method you see in object oriented PHP with two leading underscores is a *special* method provided by the language.

Consider the following:

<?php
class Person {
private $name;
function \_\_construct($name) {
$this->name = $name;
}
function getName() {
return $this->name;
}
}

In the above example the $name value is requried each time we instantiate an object. The following example will will return an error.

<?php
$myperson = new Person()

If you would like to allow an object to be instantiated with default values you can allow for this using a default constructor as represented in the following example.

<?php
class Person {
private $name;
function \_\_construct($name = 'null') {
$this->name = $name;
}
}

## Type Introspection

PHP comes with some functions to test the type of a given variable, and they all start with is\_. For example, to test that a variable is an integer, one can invoke is\_int() and it will return true or false.

How can we test that an object is a certain 'type' of object? PHP provides an is\_a function that takes as its first parameter the object in question, and as a second parameter a string of the class you wish to test against.

### Exercise:

1. Open up Exercise 1, and test that the 'person' returned from our basic function indeed returns an instance of 'stdClass'.
2. Using the class from Exercise 2, create a constructor function that takes a $name parameter and assigns that value to the name property.

# Lesson 4: Abstraction and Inheritance

## New Keywords:

* abstract
* extends
* parent

"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries relative to the perspective of the viewer."

## Abstract Classes

*An abstract class cannot be instantiated, but rather, child classes can be extended from the abstract class.*

Abstract classes are typically used as a means to organize a project. You can't create an object from an abstract class. Instead, a child class extends an abstract class, and then an object can be instantiated from the child class.

The child class *must* implement all of the abstract methods listed in the *parent class*. PHP doesn't use abstract properties, only methods, so you can signify an abstract property, but PHP doesn't force you to use it in the implementing classes.

One declares an class as abstract by using the abstract keyword on both the class, and any abstract methods must also use the abstract keyword.

Private methods cannot be declared as abstract, only public or protected methods.

### Example:

<?php
abstract class APerson {
abstract public function getName();
abstract public function setName($name);
public function sayHi() {
return "Hi.\n";
}
}

Abstract classes can have have concrete methods defined. This allows for classes that only partially need to be implemented by a derived class. In the example above, the sayHi concrete method will be availble to classes that extend the APerson class

If you fail to implement all of the abstract methods outlined in the abstract class in a child concrete class, PHP will throw an error.

Remember that abstract methods are only declarations, and that concrete methods must contain a function body whereas abstract methods must not.

PHP will throw an error if:

* a method in the abstract class isn't implemented in a derived class.
* an implemented method doesn't abide by its signature (parameters)
* the method is of a different visibility than specified

We can extend an abstract into a concrete class by using the extends keyword:

<?php
class CPerson extends APerson {
public function getName() {
}
public function setName($name) {
}
// Other functions not declared in the abstract class.
public function otherFunc() {
}
}

### Parent

You may find yourself writing code that refers to variables and functions in base classes. This is particularly true if your derived class is a refinement or specialisation of code in your base class.

class DPerson implements CPerson {
function sayHi() {
$name = $this->getName();
return parent::sayHi() . "My name is {$name}.\n";
}

## Examples in Drupal:

In includes/database.inc, Drupal defines a abstract DatabaseConnection class that extends the PDO class that comes with the PHP PDO extension for it's own purposes. This abstract class contains many concrete methods as well as abstract methods.

Different supported databases each extend this DatabaseConnection class and then have access to the augmented PDO class, with inherited methods available. These supported databases then implement the abstract methods declared in the DatabaseConneciton class as they are unique to a given database.

For instance, within the DatabaseConnection class an abstract queryRange function is declared. The implementations of this abstract method differ from MySQL to PostgreSQL as the syntax for range queries differs between the two databases.

## Exercise:

* Create an abstract class out of the Person class from lesson 2.
* Create a concrete class that extends the abstract class.
* Instantiate an object from the concrete class.
* Set the name of the person object, and then echo it back out.

# Lesson 5: Interfaces

## New Keywords:

* interface
* implements
* scope resolution operator ::

Interfaces closely resemble abstract classes, but cannot contain concrete methods or properties. In order to create an interface, use the interface keyword.

<?php
interface IPerson {
function getName();
function setName($name);
}

You can then implement an interface with another class by using the implements keyword. Just as concrete classes can contain additional properties and methods to augment an abstract class, so to can they augment interfaces.

<?php
class Person implements IPerson {
private $name;
function getName() {
return $this->name
}
function setName($name) {
$this->name = $name;
}
function \_\_construct($name) {
$this->setName($name);
}
function sayHi() {
echo "Hi.\n";
}
}

### Constants

You can't use variables inside of an interface, but you can use constants.

<?php
interface IDB {
const HOST='example.com';
const USER='root';
const PASSWORD='root';
const DB='allthedata';
function connString();
}

Those constants can be accessed via the scope resolution operator :: inside of an implementing class. For example:

class SqlDb implements IDB {
private $host = IDB::HOST;
function connString() {
echo $this->host;
}
}

### Typing Data

When a client class declares a method that uses an object, we can use type hinting to ensure that that the method is passed the correct object.

Given our previously defined IPerson interface, and Person concrete class, we can define a Client class that does something with a Person object, for example we can print their name.

It would be nice if we could ensure that the method in the client class that prints the name is passed a Person object. However, we want to type hint to say that we can use any object that impelemnts the IPerson interface rather than limit it to a specific implemenation of the interface.

When we type hint, we can specify an interface, and a instance of an implementing class will still pass the type hint. Consider the following:

<?php
class Client {
function \_\_construct() {
$person = new Person();
$this->printName($person);
}
function printName(IPerson $person) {
echo $person->getName();
}
}

In this example, even though we've passed a Person object to printName the and we've specificed that printName should be passed an IPerson, the type hint passes because the Person object is from a class that implements the IPerson interface. In the case that we created another class, lets just say WeirdPerson, we could still pass an instance of that object to printName as well since so long as the WeirdPerson class implements the IPerson interface.

### Drupal Examples:

Drupal's entities are a great place to look at interfaces. Inside of includes/entity.inc a DrupalEntityControllerInterface is defined.

This interface allows for a developer wishing to create a custom entity controller to do so simply by adhering to the interface.

Drupal's own default controller DrupalDefaultEntityController implements this interface and defines a bunch of concrete methods as well. For most developers, simply extending Drupal's DrupalDefaultEntityController is more than enough for creating custom entities, but for those using some sort of noSQL database, creating a custom controller is available, it just needs to implement the DrupalEntityControllerInterface interface.

## Exercise:

* Create an interface to give a Person object from lesson 3 an email address and website.
* Modify the concrete class from lesson 3 to implement your new interface.
* Instantiate an object from the concrete class.
* Set the email and website of the object, and then echo it back out.

# Lesson 6: Static Properties

## New Keywords:

* static

## Not Static Variables

You may have seen the use of static inside non-OO functions. Static variables keep their values across multiple function calls.

function test()
{
static $a = 0;
echo $a . "\n";
$a++;
}
test();
test();

## Static Keyword in Classes

Declaring class properties or methods as static makes them accessible without needing an instantiation of the class.

An attribute declared as static **cannot** be accessed with an instantiated class object.

A method declared as static **can** be accessed with an instantiated class object.

class Dog {
static $numLegs = 4;
static function hasClaws() {
return TRUE;
}
function bark() {
echo "bow wow\n";
}
}
class MyDog {
static $numLegs = 3;
function bark() {
echo "bow ow\n";
}
}
echo "Most dogs have " . Dog::$numLegs . " legs.\n";
echo "Can dogs scratch? " . Dog::hasClaws() . "\n";
$dog = new MyDog();
echo "My dog has " . MyDog::$numLegs . " legs.\n";
// Not
// echo "My dog has " . $dog->numLegs . " legs.\n";

# Lesson 7: Design Patterns

## Program to an interface, not an implementation.

We learned an *interface* is a set of methods that an object responds to, and an *implentation* is the code and logic for the object. Generally you want to write code you want to reference interfaces instead of implementations.

This decouples design from the specific implementation of concrete classes, and allows you to swap out one implementation for another easily, as well as create new implementations.

In PHP we can accomplish this by specifying interface data types in type hinting as shown in the previous lesson. It is important to note that by 'interface data types', we mean interfaces *or* abstract classes.

Also, as previously noted, we can instantiate an object of a given concrete implementation, and so long as it implements an interface or extends an abstract class, those parent classes will be used to identify the object curing type hinting.

## Favor object composition over class inheritance.

If at all possible, try to create client classes that are composed of various objects.

Consider the following classes:

1. Inheritance

<?php
class Pet {
function speak() {
echo 'speaking';
}
}
class Dog inherits Pet {
function bite() {
echo '\/\/\/';
}
}
class Client {
function \_\_construct() {
$dog = new Dog();
}
}

This contrived example shows a client class that uses an object.

## Strategy

Encapsulates specific families of algorithms allowing the client class responsible for instantiating a particular algorithm to have no knowledge of the actual implementation.

interface OutputInterface
{
public function load();
}
class SerializedArrayOutput implements OutputInterface
{
public function load()
{
return serialize($arrayOfData);
}
}
class JsonStringOutput implements OutputInterface
{
public function load()
{
return json\_encode($arrayOfData);
}
}
class ArrayOutput implements OutputInterface
{
public function load()
{
return $arrayOfData;
}
}

## Dependency Injection

In Drupal 8 *dependency injection* is the preferred method for accessing and using services. Services are reusable functionality (e.g. database connection, string translation, cacheing, authentication) that are made pluggable and replaceable via dependency injection.

An injection is the passing of a dependency (a service) to a dependent object (a client). The service is made part of the client's state. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

Some of the design patterns that preceded DI:

### Singleton

A singleton is a class that is intended to be instantiated once and reused in multiple scopes in an application.

class Singleton
{
/\*\*
\* Returns the \*Singleton\* instance of this class.
\*
\* @staticvar Singleton $instance The \*Singleton\* instances of this class.
\*
\* @return Singleton The \*Singleton\* instance.
\*/
public static function getInstance()
{
static $instance = null;
if (null === $instance) {
$instance = new static();
}
return $instance;
}
/\*\*
\* Protected constructor to prevent creating a new instance of the
\* \*Singleton\* via the `new` operator from outside of this class.
\*/
protected function \_\_construct()
{
}
/\*\*
\* Private clone method to prevent cloning of the instance of the
\* \*Singleton\* instance.
\*
\* @return void
\*/
private function \_\_clone()
{
}
/\*\*
\* Private unserialize method to prevent unserializing of the \*Singleton\*
\* instance.
\*
\* @return void
\*/
private function \_\_wakeup()
{
}
}
class SingletonChild extends Singleton
{
}
$obj = Singleton::getInstance();
var\_dump($obj === Singleton::getInstance()); // bool(true)
$anotherObj = SingletonChild::getInstance();
var\_dump($anotherObj === Singleton::getInstance()); // bool(false)
var\_dump($anotherObj === SingletonChild::getInstance()); // bool(true)

### Factory

A factory class simply creates the object you want to use. The purpose of the factory patterns is to separate the use of a certain component, from the choice of implementation + instance management of that component.

class Automobile
{
private $vehicleMake;
private $vehicleModel;
public function \_\_construct($make, $model)
{
$this->vehicleMake = $make;
$this->vehicleModel = $model;
}
public function getMakeAndModel()
{
return $this->vehicleMake . ' ' . $this->vehicleModel;
}
}
class AutomobileFactory
{
public static function create($make, $model)
{
return new Automobile($make, $model);
}
}
// have the factory create the Automobile object
$veyron = AutomobileFactory::create('Bugatti', 'Veyron');
print\_r($veyron->getMakeAndModel()); // outputs "Bugatti Veyron"

## References

* Design Patterns - PHP: The Right Way - http://www.phptherightway.com/pages/Design-Patterns.html
* Dependency injection - Wikipedia, the free encyclopedia - http://en.wikipedia.org/wiki/Dependency\_injection