OOP with PHP

- Kadhem Soltani
- Soltani.kadhem@novavision-it.com
- http://kadhem-soltani.com/blog/

NOVAVISION-IT

http://novavision-it.com/

Whats in store

- OOP Introduction
- Classes and Objects
- Encapsulation
- Polymorphism
- Inheritance

OOP introduction

Introduction to OOP

- Concepts are represented as objects
- Objects have **properties** which contain information about the object
- Functions associated with objects are called **methods**

Classes and objects

Classes and Objects

- A class is a definition of an object. It is a file containing:
  - The namespace of the class
  - The name of the class
  - A list of the properties of the class
  - Implementations of any methods
- A class becomes an object when it is instantiated with the **new** keyword

Defining a class

```
<?php
class MyClass {

        // implementation of the class goes in here

}
```

Visibility

- **Public** refers to methods and properties which can be accessed or called from both within the object itself and outside of the object (e.g. other files and objects)
- **Private** refers to methods and properties which can only be accessed or called from within the object itself
- **Protected** refers to methods and properties which can be accessed or called from within the object itself **and** objects which extend from the object (child objects)

Methods and Properties

```php
<?php
class MyClass {
        private $someProperty = 'a default value';
        private $anotherPropert;

        public function myMethod()
        {
                echo "This is my method being run";
        }

}
```

Magic Methods

These are methods which are called automatically when certain actions are undertaken, though they can be manually called too.

Typically prefixed with two underscores

Magic methods

- __**construct** the constructor method; called on an object as soon as it is instantiated. It can accept parameters.
- __**destruct** the destructor method; called when an object stops being used/referenced or during shutdown sequence
- __**toString** is called when you try to use an object as a string (e.g. echo $my_object) and returns the string representation of the object
- __**clone** is called when you try and clone an object (e.g. $new_object = clone $old_object) useful for dereferencing any unique IDs, etc.
- Full list: http://php.net/manual/en/language.oop5.magic.php

With a constructor

```php
<?php
class MyClass {

	private $someProperty = 'a default value';
	private $anotherPropert;

	public function __construct()
	{
		echo "this is called when you create the object";
	}

	public function myMethod()
	{
		echo "This is my method being run";
	}

}
```

Working with objects

- **Creating**

  $my_object = new SomeObject();

- **Accessing properties**

  echo $my_object->someProperty; // must be public

- **Calling methods**

  $my_object->someMethods(); // must be public

$this

When working within an object, you access properties and methods within the object using the $this keyword.

For example, consider a method which returns the total cost for an order. This needs to add the cost and the delivery cost. Both of these costs are calculated using other methods

```
return $this->calculateCost() + $this->calculateDeliveryCost();
```

Static

Properties can be accessed and methods can be called from an uninstantiated class (i.e. not an object) if they are prefixed with the static keyword. They are called and accessed with the Scope Resolution Operator (::)

```
class MyClass {
        public static function printHello()
        {
        print “hello”;
        }
}

MyClass::printHello();
```

Class constants

- Classes can also contain **constants**; these are similar to properties except they cannot be changed (unless you edit the php file of course!)
- const myConstant = “some value”;

Interfaces and Implements

- An **interface** defines non-private methods (as well as the number of parameters they accept) that a class which **implements** the interface must have
- If a class specifically **implements** an **interface** it *must* implement the methods defined, if it does not PHP will raise errors.

Defining an interface

```
<?php
interface MyInterface {

        public function mustImplementThis();

        public function mustImplementThisAsWell($with, $some, $parameters=null);


}
```

Creating a class which implements an interface

```php
<?php
class MyClass implements MyInterface {

	public function mustImplementThis()
	{
		// put some code here
	}

	public function mustImplementThisAsWell($with, $some, $parameters=null)
	{
		// put some code here
	}

}
```

Extends

- One class can **extend** another. When it does this the class inherits properties, methods and constants from the **parent** class (the one it **extends**) - this is where visibility settings are essential.
- Public and protected properties and methods can be overridden in the child class; provided the method names and number of parameters match.

Extends in action

```php
<?php
class MyChildClass extends MyParentClass {
        // now we have a class which has the same properties
        // and methods as the MyParentClass

        // we can add new ones here and override the parent ones
        // if we want to

}
```

Parent keyword

- If you have a method in a **child** class from which you want to access properties or methods in the **parent** class, you use the **parent** keyword with the scope resolution operator.

```php
<?php
class Someclass extends Parentclass {

	public function test()
	{
		// this will call the someMethod method
		// in the Parentclass class
		echo parent::someMethod();
	}

}
```

Abstract class

- An **abstract** class gives us the best of both worlds; we can define methods which need to be **implemented and** we can create methods and properties which can be **extended** by a **child** class
- An abstract class is defined using the abstract keyword
- However...an abstract class cannot be instantiated directly. Only a class which extends it can be instantiated.

Defining an abstract class

```php
<?php
abstract class MyAbstractClass {

	protected $someProperty;

	public function implementMe();
	public function implementMeToo();

	protected function someMethod()
	{
		echo 'a';
	}

	public function __toString()
	{
		return $this->someProperty;
	}

}
```

Using an abstract class

```
<?php
class MyClass extends AbstractClass {

        protected $someProperty;

        public function implementMe()
        {
                // implementation
        }

        public function implementMeToo()
        {
                // implementation
        }

        // we dont need to implement someMethod() or __toString
        // as the abstract class implements them
        // we can override them if we want to

}
```

Encapsulation

Encapsulation

With encapsulation the **internal representation** of an object is hidden from view outside of the class. Often only the object itself is permitted to directly access and modify its properties.

This approach means we, the programmer, can have greater control over how these properties are modified.

Imagine setting an email address property when a user fills out a contact form. If the property is public, then the property could be any value. If we hide the property and use a setter method, we can put some business logic between the user input and the property such as some validation.

Polymorphism

Polymorphism

Polymorphism allows an object, variable or function have more than one form.

Why polymorphism

- You have a class (Email) which represents an email
- You have a number of classes which can take an email object and send it using a specific technology (e.g. SMTP, sendmail, postmarkapp) lets use SmtpTransport as an example
- When you need to send an email you have some code (emailer code) which takes the email, takes the SmtpTransport and links the two together
- If you want to use your other class, PostmarkappTransport how does “email code” know how to work with PostmarkappTransport...these are both different classes

Why polymorphism...

- Answer: Interfaces and polymorphism
- We have a TransportInterface which defines the public methods we will use to send the email
- The *Transport classes all implement this and use their implementations of the public methods to send the email differently
- Our email code doesn’t care about if its given a SmtpTransport or PostmarkappTransport it only cares if they implement the TransportInterface
- Through polymorphism our Transport classes are both instances of:
  - *Transport
  - TransportInterface

Type hinting to require an interface

```
function myEmailCode(TransportInterface $transport, $email)
{
        $transport->setEmail($email);
        if( ! $transport->send() ) {
                echo $transport->getErrors();
        }
}
```