Introduction

On the old site, I started a series of tutorials named "PHP Application Design". I planned for at least three parts, and while I did publish the first two and wrote at least half of part three, I found myself out of time to make it to the finish.

Still dedicated to writing reusable and extensible PHP code and spreading the knowledge, this is a restart of the series, renamed as simply "OO PHP". So:

Welcome to part one of the "OO PHP" series.

This article is an overview of what PHP has to offer with respect to OOP, with examples on how to use these concepts.

If you have wanted to grasp OOP, but haven't come around to learning it, this article is for you. If you've read some "Hello World" in OOP tutorial, gotten familiar with OOP syntax but are interested in learning more, this article is for you too.

A couple of things have changed in this tutorial. First off, PHP 4 examples and references have been stripped. PHP 4 is dead, let’s look to the future. In that same frame of mind, this tutorial now has some brief coverage of namespaces and late static binding, introduced in PHP 5.3. Finally, the last chapter from the original has been stripped, or rather moved to a future part of the series.

Index

1. The Very Basics
1.1 A tiny bit of theory
1.2 Hold on, read this disclaimer first
1.3 Absolute basic syntax
2 Defining how objects should behave
2.1 Classes
2.2 Inheritance
2.3 Constructors
2.4 Scope Resolution Operator
2.5 Abstract Classes
3 Beyond the Absolute Basics
3.1 Object Handles
3.2 Interfaces
3.3 Autoload
3.4 Destructors
3.5 Visibility
3.6 Class Constants
3.7 Type Hinting
3.8 Exceptions
3.9 The Final Keyword
3.10 More Magic Methods
3.10.1 Object Overloading
3.10.2 Object cloning
3.10.3 Serialization
3.10.4 Other
3.11 Object Iteration
4. Hot off The Press Features
4.1 Namespaces
4.2 Late static binding
In conclusion

1. The Very Basics

1.1 A tiny bit of theory

So what is an object anyway? In a definition proposed by IBM's Grady Booch in 1991:
An object is an instance of a class with the following essential properties:

- Each object has its own identity.
- Each object can have certain behaviour.
- Each object is in a state.

Let's do a quick elaboration on these properties.
Each object has its own identity.
'Having its own identity' implies that even if two objects are structurally the same (of the same class and in the same state), they are still not identical.
Each object can have behaviour.
This behaviour is used, thus this is also referred to as "offer services". Basically an object has functions and variables local to that object, referred to as methods and properties (more on that in the next section), which define and influence this behaviour.
Each object is in a state.
The before mentioned properties (local variables) define the state an object is in. This state can influence the behaviour (the methods act differently depending on the properties).
In addition, objects often represent some concept, an object in the more literal sense.
I know you all can't stand all this abstract talk and are itching for a code example, so here we go…

1.2 Hold on, read this first

Wait. Before I show you ANY code AT ALL, note that in this tutorial I violate a truckload of 'good practice' heuristics. This is done to be able to explain you the basics. Take note of the fact that this tutorial is intended to show the features available to you in PHP, it does not promote any actual practice. Don't worry, I will definitely get to that later.

1.3 Absolute basic syntax

Instantiating an (creating a new, unique) object:

```
$someVariable = new SomeClassDefiningAnObject;
```

Executing a function in the object (a method):

```
$someVariable->someMethod($someArgumentJustLikeARegularFunction);
```

Assigning the return value from a method to a variable:

```
$returnValue = $someVariable->someMethodThatReturnsSomething();
```

Setting and retrieving the current value of a property (a variable local to the object, not unlike an array index):

```
$someVariable->someProperty = 'SomeValue';
$currentValue = $someVariable->someProperty;
```

Hopefully you now have a basic understanding of what objects are and how to operate on them. Next we're looking at defining the state and behaviour of objects.

2 Defining how objects should behave

2.1 Classes

Classes are the (partial) definition of an object. Also referred to as a “object blueprint”, the class defines how the object will look after first instantiation, and how it will behave in response to operating on it. However it also possible to operate within the scope of a class without an instantiation. You will read more about that in the “Scope Resolution Operator” section.
Consider this extremely simple example of a class:

```
class Dog
{
        public $hungry = 'hell yeah.';

        function eat($food)
        {
                $this->hungry = 'not so much.';
        }
}
```

$this is a reserved variable name, referring to the current instantiation of the object. By instantiating, we create a new Dog. Don’t mind the public keyword for now.

```
$dog = new Dog;
```

The initial state of this dog is that it is pretty hungry.

```
echo $dog->hungry;
```

Echoes:

```
hell yeah.
```

So we feed our newly instantiated dog a treat, to modify its state.

```
$dog->eat('cookie');
```

Yep, doggie isn’t as hungry as before:

```
echo $dog->hungry;
```

Will echo:

```
not so much.
```

Doggie is happy enough, moving on.

2.2 Inheritance

Classes can have parent and child classes. When an object is instantiated by name of a child class, PHP looks for parent classes and starts combining the declarations, from the top, to create the initial object. This is called inheritance. It isn’t such a hard concept to grasp: a child inherits anything from its parents and ancestors that isn’t redefined on the way down. If a class lower in the chain has a property or method with the same name as one higher in the chain, the last declaration is used for requests to the object.
To declare one class as a child of another, use the extends keyword.
Like with most popular languages, in PHP one class can only extend a single other class. An infinite number of classes can be derived from (extending) a single class though.
A simple example:

```
class Animal
{
        public $hungry = 'hell yeah.';

        function eat($food)
        {
                $this->hungry = 'not so much.';
        }
}
class Dog extends Animal
{
        function eat($food)
        {
                if($food == 'cookie')
                {
                        $this->hungry = 'not so much.';
                }
                else
                {
                        echo 'barf, I only like cookies!';
                }
        }
}
```

Method eat is overridden, because doggie only likes cookies. But, because all animals are hungry when you don't feed them, the initial state of $hungry need not be defined in Dog. The fictional Bird, Cat and Piggy could all extend Animal.
Class Animal is unaware of the fact that it is being extended; there are no references to Dog whatsoever. Say Animal extended another class called LifeForm, and I instantiated Animal, only methods and properties of Animal and LifeForm would be included in the object.

2.3 Constructors

Constructors are a way to define the default behaviour (set the default state) upon instantiation. You can, as shown in the previous example, define the default state by setting default values for properties. A constructor is nothing more than a method that is executed when instantiating an object. This is referred to as initializing the object.
An object isn't fully initialized, constructed, until the constructor method has completed (if present). A constructor method isn't required, our previous examples worked just fine without them. Another thing that should be noted is that only the constructor of the class used to instantiate is called automatically, any parent constructors need to be called explicitly.
PHP 5 uses the __construct magic method. You can use the name of the class as well, but this is only for backwards compatibility, and will throw an error if you have E_STRICT error reporting enabled (which you should have).

```
class Dog extends Animal
{
        public $breed;

        function __construct($breed)
```

```
        {
                $this->breed = $breed;
        }

        function eat($food)
        {
                if($food == 'cookie')
                {
                        $this->hungry = 'not so much.';
                }
                else
                {
                        echo 'barf, I only like cookies!';
                }
        }
}
```

Instantiating a Dog requires us to specify it's breed. The property breed is assigned a new value by cause of this initialization (previously NULL), and we have a Dog which initial state specifies it's breed:

```
$dog = new Dog('Golden Retriever');
```

We could change its breed afterwards if we'd like:

```
$dog->breed = 'Bloodhound';
```

2.4 Scope Resolution Operator

Officially called Paamayim Nekudotayim (Hebrew for double colon, now you know what the parser is talking about when you get errors), the scope resolution operator (::) allows you to perform static calls to methods and class members.

Static methods can be called without instantiation of an object. This can be useful both inside a class declaration (within an object hierarchy) as outside of it.

After an object is created, some reference to the overwritten methods and properties remains intact. One can access these methods from within the object statically (no new instantiation is required, yet the object from within you are operating is still affected.).

```
class Animal
{
        public $hungry = 'hell yeah.';

        function __construct()
        {
                echo 'I am an animal.';
        }

        function eat($food)
        {
                $this->hungry = 'not so much.';
        }
```

```
}
class Dog extends Animal
{
        public $breed;

        function __construct($breed)
        {
                $this->breed = $breed;

                Animal:: __construct();
        }

        function eat($food)
        {
                if($food == 'cookie')
                {
                        Animal::eat($food);
                }
                else
                {
                        echo 'barf, I only like cookies!';
                }
        }
}
$dog = new Dog('Rotweiler');
$dog->eat('cookie');
echo $dog->hungry;
```

Dog is using its parent's method eat to refer to the declaration of eat that was overwritten by its own declaration of eat. You need not specify the specific class name like in the above example to refer to the last declaration of a method (or property), the parent keyword will point to that reference:

```
parent::eat($food);
```

Should you have multiple levels of inheritance, and want to address a declaration other than the one last defined, you'll have to specify the class like in the above example class.
The constructor of parent classes is called upon instantiation, unless it is overridden. We override __construct(), so if we desire to run a parent constructor, we have to call it explicitly:

```
Animal::__construct();
```

Trying to call a method statically which has references to object specific variables (makes use of $this) from outside of the object, will throw an error:

```
Dog::eat('cookie');
```

Result:
`Fatal error: Using $this when not in object context`

PHP 5 features static class members. It requires you to define what methods and properties can be used statically. Static properties work the same as regular static variables, if you are unfamiliar with static variables, have a thorough read on Variable Scope in the manual.

```
class Dog
{
        static $hungry = 'hell yeah.';

        static function eat($food)
        {
                if($food == 'cookie')
                {
                        self::$hungry = 'not so much.';
                }
                else
                {
                        echo 'barf, I only like cookies!';
                }
        }
}
Dog::eat('cookie');
echo Dog::$hungry;
```

Note the self keyword. It is simply referring to this class declaration. Like with the parent keyword, alternatively one could use the actual name of the class (Dog).

2.5 Abstract Classes

Abstract classes are, well, abstract. An abstract class isn't intended to be instantiated, but to serve as a parent to other classes, partly dictating how they should behave. Abstract classes can have abstract methods, these are required in the child classes.
Abstract classes can be used to enforce a certain interface.
An example:

```
abstract class Animal
{
        public $hungry = 'hell yeah.';

        abstract public function eat($food);
}
class Dog extends Animal
{
        function eat($food)
        {
                if($food == 'cookie')
                {
                        $this->hungry = 'not so much.';
                }
                else
                {
```

```
                        echo 'barf, I only like cookies!';
                }
        }
}
$dog = new Dog();
echo $dog->hungry; //echoes "hell yeah."
$dog->eat('peanut'); //echoes "barf, I only like cookies!"
```

3 Beyond the Absolute Basics

3.1 Object Handles

In PHP5, objects are defined by handles, not unlike resource type variables.

Passing an object to a function doesn't make a copy of it. Have a read on References Explained if you're still a little confused by this.

3.2 Interfaces

PHP5 features interfaces. Not to be confused with interfaces in the more general sense, the interface keyword creates an entity that can be used to enforce a common interface upon classes without having to extend them like with abstract classes. Instead an interface is implemented.

Interfaces are different from abstract classes. For one, they're not actually classes. They don't define properties, and they don't define any behaviour. The methods declared in an interface must be declared in classes that implement it.

Because an interface in the more general sense is a definition of how an object interacts with other code, all methods must be declared public (see section on visibility in this chapter). Using abstract classes, an abstract method can have any visibility, but the extending classes must have their implementations use the same (or weaker) visibility. Implementing an interface adds the methods as abstract methods to the subject class, failure to implement it will result in an error like the following:

```
Fatal error: Class SomeConcreteClass contains n abstract method(s) and
must therefore be declared abstract or implement the remaining
methodsYes, abstract classes can implement interfaces.
```

Interfaces can be looked upon as a contract, a certificate of compliance if you will. Other code is guaranteed that a class implementing it will use certain methods to interact.

Enough babbling, let's see a code example:

```
interface Animal
{
        public function eat($food);
}
interface Mammal
{
        public function giveBirth();
}
class Dog implements Animal, Mammal
{
        public $gender = 'male';

        function eat($food)
        {
                if($food == 'cookie')
```

```
                {
                        $this->hungry = 'not so much.';
                }
                else
                {
                        echo 'barf, I only like cookies!';
                }
        }
        function giveBirth()
        {
                if($this->gender == 'male')
                {
                        echo 'I can\'t, I am a boy :P';
                }
                else
                {
                        echo 'I\'m not even pregnant yet.';
                }
        }
}
```

Doggie implements 2 interfaces, both Animal and Mammal. You can implement as many interfaces as you like.

3.3 Autoload

A very convenient feature, __autoload allows you to get rid of all those annoying includes that need to be managed. This magic function will execute whenever a class or interface is referenced that hasn't been defined. That gives you the opportunity to include it.

Here's a simple example of how that might look using the standard PEAR naming scheme (the segments between underscores become directory names, bar the last, which becomes the filename):

```
function __autoload($className)
{
        $file = str_replace('_', DIRECTORY_SEPARATOR, $className) .
'.php';

        if(!file_exists($file))
        {
                return false;
        }
        else
        {
                require_once $file;
        }
}

new Foo_Bar(); //Maps to 'Foo/Bar.php'
```

We don't really need require_once, because once the file is included, __autoload will not trigger on that class or interface reference again.

3.4 Destructors

Destructors are another type of magic methods. Indicated by __destruct, these methods are called when all references to their objects are removed. This includes explicit un-setting and script shutdown. A quick example of this:

```
class Example
{
        private $_name;

        public function __construct($name)
        {
                $this->_name = $name;
        }
        function __destruct()
        {
                echo "Destructing object '$this->_name'." . PHP_EOL;
        }
}

$objectOne = new Example('Object one');
$objectTwo = new Example('Object two');
unset($objectOne);
echo 'Script still running.' . PHP_EOL;
```

This echoes:

```
Destructing object 'Object one'.
Script still running.
Destructing object 'Object two'.
```

Object one is destructed when we explicitly unset it, object two just before the script execution is completed. An object's destructor is always executed (or rather 'should be': don't rely on this mechanism too much). Note that if we had created a second object from within class Example, it's destructor would also have been executed, as it would have removed all references to the embedded object.

3.5 Visibility

PHP 5 allows you to declare the visibility of methods and properties. There are three types of visibility: public, protected and private.

Public

Public methods and properties are visible (accessible) to any code that queries them. No accessibility restrictions are applied. In PHP 5, methods without visibility declaration are assumed public, the visibility declaration is required, excluding static class members (if you don't include it, 'public' is assumed). Class constants are always globally available.

Protected

Requests are only allowed from within the objects blueprint (that includes parent and child classes). Meaning the following would fail:

```
class Teeth
{
        protected $_colour = 'white';
```

```
        public function stain()
        {
                $this->_colour = 'yellow';
        }
}
class Dog
{
        public $teeth;

        public function __construct()
        {
                $this->teeth = new Teeth();
        }
        public function eat($food)
        {
                if($food == 'cookie')
                {
                        $this->hungry = 'not so much.';
                        //Attempt to turn teeth green:
                        $this->teeth->_colour = 'green';

                }
                else
                {
                        echo 'barf, I only like cookies!';
                }
        }
}
$dog = new Dog();
```

Produces:

```
Fatal error: Cannot access protected property Teeth::$_colour
```

Should Teeth have been a parent class of Dog instead of a member object (class Dog extends Teeth - don't do that), $colour would have been accessible though $this, but also statically by Teeth::$_colour (or parent::$_colour).

Private

Access is limited to the declaring class (the class the property is declared in). No external access whatsoever is allowed.

One thing that should be noted when using protected or private properties, is that if you attempt to assign a value to a property that isn't visible to the class you are doing it in, you will be creating a new property instead of resetting the original. Keep that in mind when you get unexpected values: check the property's visibility.

3.6 Class Constants

We already covered using static class members, there is another type of class member, which we haven't covered: class constants.

I am assuming you are already familiar with regular constants, and are aware of how they differ from variables.

Class constants are just regular constants, declared in a class. It's reference is obtained through the class scope. Because constants are unchangeable, they are independent of any state the object could be in. Therefore they can only be called statically.

```
class Dog
{
        const NUMBER_OF_LEGS = '4';

        public function __construct()
        {
                echo 'I have '.self::NUMBER_OF_LEGS.' legs,
                and you can\'t take that away from me!';
        }
}
$dog = new Dog();
```

Both $dog->NUMBER_OF_LEGS and $this->NUMBER_OF_LEGS would have PHP looking for a non-existent object property: $NUMBER_OF_LEGS.
Class constants are always publicly accessible. Any code can call Dog::NUMBER_OF_LEGS.

3.7 Type Hinting

PHP 5 features type hinting as a means to limit the types of variables that a method will accept as an argument. Let's kick off with a simple example:

```
class Rabbit
{}

class Feeder
{
        public function feedRabbit(Rabbit $rabbit, $food)
        {
                $rabbit->eat($food);
        }
}
$dog = new Dog();
$feeder = new Feeder();
$feeder->feedRabbit($dog, 'broccoli');
```

Attempting to use an instance of Dog with the feedRabbit method results in the following error:

```
Fatal error: Argument 1 passed to Feeder::feedRabbit() must be an
instance of Rabbit
```

However, type hinting allows a more generic use. Consider the following example:

```
class Animal
{}
class Feeder
{
        public function feedAnimal(Animal $animal, $food)
        {
                $animal->eat($food);
```

```
        }
}
class Dog extends Animal
{
        public function eat($food)
        {
                if($food == 'cookie')
                {
                        $this->hungry = 'not so much.';

                }
                        else
                {
                        echo 'barf, I only like cookies!';
                }
        }
}
$dog = new Dog();
$feeder = new Feeder();
$feeder->feedAnimal($dog, 'broccoli');
```

Because $dog is not only a Dog, but also an Animal, the requirement is met. Doggie doesn't like broccoli (who does?), but that is besides the point. Currently PHP only supports type hinting with objects and arrays.
Note: since PHP 5.2, failing type hinting requirements result in a E_RECOVERABLE_ERROR type error, not an immediate fatal error.

3.8 Exceptions

PHP 5 introduces the concept of exceptions to PHP. An exception is not an error, an uncaught exception is (a fatal error).
Using exceptions you will need the following keywords: try, throw and catch. PHP 5 has it's build-in Exception class, which you can extend.
You can have multiple catch blocks following a try block. PHP will execute the first catch block that matches the type of the exception thrown. If no exception is thrown or none of the catch declarations match the thrown exception type, no catch blocks are executed.

```
class LiarException extends Exception
{}

try {
        if($doggy->talk() == 'Doggie likes broccoli.')
        {
                throw new LiarException(
                        'Doggie is a big fat liar. He only likes cookies.'
                );
        }
        else
        {
                throw new Exception('Just because we can.');
        }
```

```
        echo 'An exception was thrown, so this will never print...';
}
catch(LiarException $e)
{
        echo "Somebody lied about something: {$e->getMessage()}'";
}
catch(Exception $e)
{
        echo "Somebody threw an exception: {$e->getMessage()}'";
}
```

In the example above, the code in the try block is executed until an exception is thrown. If an exception is thrown, the code in the corresponding catch block is executed. As noted, PHP executes the FIRST matching catch block. So if we switched the catch blocks, the first block would be executed, even though the second is a ‘closer match’.
If there isn’t a corresponding catch block, the exception is ‘falls through’. Assuming the call to the code is not itself contained in a try block, the exception is uncaught, resulting in a fatal error.

3.9 The Final Keyword

The final keyword indicates that the declaration following it is final, no child class will be able to redefine it. One can declare both a single method or a whole class as final. If the whole class is declared final, it becomes un-extendable: no child classes are allowed.
There really isn’t much more to it, apply it only if you are absolutely sure you don’t want that specific class or method extended or re-declared respectively.

3.10 More Magic Methods

Aside from __construct, __destruct, there are quite a few more magic methods:
Overloading

- __call
- __get
- __set
- __isset
- __unset

Object cloning

- __clone

Serialization

- __sleep
- __wakeup

Other

- __toString
- __setState

3.10.1 Object Overloading

If you're coming from a different OO language, the term 'overloading' likely has a very different meaning to you: defining different method with the same name having different signatures. This has nothing to do with that.

Object overloading in PHP refers to the mechanism where a call to a method or property will 'overload' the call to a different property or method. The call ultimately made can depend on the type of the arguments or the context.

These magic methods allow you catch calls to methods and properties that haven't been defined, because you didn't know (or didn't want to specify) the exact name.

The magic methods are executed only if the object doesn't have the method or property declared.

The following example class uses __set to check if an embedded object does have the property requested, before creating a new property for the parent object.

```
class DomXml
{
        private $_domDoc;

        public function __construct()
        {
                $this->_domDoc = new DOMDocument();
        }
        private function __set($name, $value)
        {
                if(property_exists($this->_domDoc, $name))
                {
                        $this->_domDoc->$name = $value;
                }
                else
                {
                        $this->$name = $value;
                }
        }
}
```

An example __call use:

```
private function __call($name, $params)
{
        if(method_exists($this->_doc, $name))
        {
                return call_user_func_array(
                        aray($this->_doc, $name), $params
                );
        }
        else
        {
                throw new DomXmlException(
                        "Call to undeclared method '$name'"
```

```
                );
        }
}
```

Methods __get, __isset and __unset have similar uses.

3.10.2 Object cloning

Because in PHP 5 objects use handles, if you would want to create a copy of an object you have to explicitly indicate so with the clone keyword.

```
$obj = new Example();
$objectCopy = clone $obj;
```

Above gives you a new object that is an exact copy of $obj in the state it was when copied.
The __clone method is triggered by this clone keyword. This allows you, for example, to ensure embedded objects are also cloned (otherwise they would this use the same object handle for the embedded objects).
The following example illustrates this:

```
class Teeth
{
        public $colour = 'white';
}
class Dog
{
        public $teeth;

        public function __construct()
        {
                $this->teeth = new Teeth();
        }
}
$Lassie = new Dog();
$Snoopy = clone $Lassie;
$Snoopy->teeth->colour = 'green';
echo $Snoopy->teeth->colour . PHP_EOL;
echo $Lassie->teeth->colour . PHP_EOL;
```

That will echo:

```
green
green
```

Changing the property colour to green on $Snoopy's teeth, also changes $Lassie's teeth to green, because they share the same teeth. Imagine that, two dogs attached at the mouth…
To give Lassie's clone Snoopy its own set of teeth, we can use the __clone method:

```
class Dog
{
        public $teeth;

        public function __construct()
        {
```

```
                $this->teeth = new Teeth();
        }

        public function __clone()
        {
                $this->teeth = clone $this->teeth;
        }
}
```

Now it will output:

```
green
white
```

Voila, a cloned Dog now has its own teeth.

3.10.3 Serialization

Resource handles don't serialize. If one of your properties has a resource handle as its value, for example a file handle or database connection, You can use the __wakeup magic to re-establish these handles. This method will be called when you call unserialize($serializedObject). The __sleep method is invoked when you serialize an object, and is expected to return an array of property names to be included in the serialization. This allows you to exclude certain properties.

3.10.4 Other

Other methods that don't fit in the previous categories.

__toString: This magic method can be used to define how an object will present itself if the whole object is treated as a string.

Example:

```
class Dog
{
        function __toString()
        {
                return 'I am Dog.';
        }
}
$dog = new Dog();
echo $dog;
```

Result:

```
I am Dog.
```

Note that older versions of PHP 5 didn't support calling __toString() when embedding or concatenating in a string. Currently, this will work as well:

```
$dog = new Dog();
echo "Doggie says: $dog";
```

__set_state: This is a static method that can be used to reinitialize an object after being "flattened" by var_export.

3.11 Object Iteration

PHP allows objects to be iterated, all properties visible to the scope the iteration is requested from are used. Example of this with a foreach loop:

```php
class ExampleParent
{
        protected $propertyOne = 'value1';
        public $propertyTwo = 'value2';
}
class Example extends ExampleParent
{

        private $propertyThree = 'value3';

        public function __construct()
        {
                echo "Internal iteration of Example:" . PHP_EOL;

                foreach($this as $property => $value)
                {
                        echo "Property '$property' => '$value' " .
PHP_EOL;
                }
                echo PHP_EOL;
        }
}
$example = new Example();

echo "External iteration of Example:" . PHP_EOL;

foreach($example as $property => $value)
{
        echo "Property '$property' => '$value' " . PHP_EOL;
}
echo PHP_EOL;

$exampleParent = new ExampleParent();

echo "External iteration of ExampleParent:" . PHP_EOL;

foreach($exampleParent as $property => $value)
{
        echo "Property '$property' => '$value' " . PHP_EOL;
}
```

Output:

```
Internal iteration of Example:
Property 'propertyThree' => 'value3'
Property 'propertyOne' => 'value1'
Property 'propertyTwo' => 'value2'
```

```
External iteration of Example:
Property 'propertyTwo' => 'value2'
External iteration of ExampleParent:
Property 'propertyTwo' => 'value2'
```

You'll notice that the external iteration doesn't list the private and protected properties.

4. Hot off The Press Features

4.1 Namespaces

PHP 5.3 and up offers namespaces which allows you to give your classes logical names without resorting to really long winded names like the ones caused by the PEAR naming standard.

Simply put, the PEAR naming standard defines pseudo namespaces, which map to a location on the file system (see section Autoload). A name like the following:

Application_Input_Validate_PhoneNumber_Us

Would map to Application/Input/Validate/PhoneNumber/Us.php

Sometimes these names get way bigger than this. Here's a different example, from Zend Framework:

Zend_Controller_Action_Helper_AutoComplete_Abstract

It can get even bigger.

PHP 5.3+ namespaces solves this by creating aliases.

```php
namespace Application::Input::Validate::PhoneNumber

class Us {

}
```

Now, I could call this:

```php
$validator = new Application::Input::Validate::PhoneNumber::Us($arg);
```

But that can hardly be called an improvement, can it?

Instead, knowing that I am going to use a phone number validator, I can create a namespace alias:

```php
use Application::Input::Validate::PhoneNumber as PhoneValidators

$validator = new PhoneValidators::Us($arg);
```

Better, no? It saves me having to refer to the fully quantified class name, and I can select the right validator class in my client code without littering it with repeated lengthy namespace selections.

I can also create an alias for a class name. For example:

```php
use Framework::Controller::Response::Http HttpResponse

$response = new HttpResponse();
```

The fully quantified name of the class includes the namespace. Hence, calling var_dump on the above $response object would output something like the following:

```
object(Framework::Controller::Response::Http)#1 (0) {
}
```

This allows us to mend our __autoload function to load classes in a namespace:

```php
function __autoload($className)
{
        $file = str_replace('::', DIRECTORY_SEPARATOR, $className) .
'.php';

        if(!file_exists($file))
        {
                return false;
        }
        else
        {
                require_once $file;
        }
}
```

4.2 Late static binding

In PHP 5.2, you can already do this:

```php
abstract class ParentClass
{
        public static function foo()
        {
                echo 'blah';
        }
}
class ImplementationClass extends ParentClass
{}

ImplementationClass::foo();
```

Static methods are inherited. But without late static binding, ParentClass will not be able to invoke any static methods of ImplementationClass, like in this non-static example:

```php
abstract class ParentClass
{
        public function foo()
        {
                $this->meh();
        }
}
class ImplementationClass extends ParentClass
{
        public function meh()
        {
                echo 'blah';
        }
}
```

```
$impl = new ImplementationClass();
$impl->foo();
```

If we wanted meh to be static, it would be tempting to try this:

```
abstract class ParentClass
{
        public function foo()
        {
                self::meh();
        }
}
class ImplementationClass extends ParentClass
{
        public static function meh()
        {
                echo 'blah';
        }
}
ImplementationClass::foo();
```

But self refers to the current class scope, ParentClass in this case, so we produce this error:
`Fatal error: Call to undefined method ParentClass::meh()`
In PHP5.3+, self still points to the current class reference. To make the above scenario possible, a new use is given to the static keyword:

```
abstract class ParentClass
{
        public static function foo()
        {
                static::meh();
        }
}
class ImplementationClass extends ParentClass
{
        public static function meh()
        {
                echo 'blah';
        }
}
ImplementationClass::foo();
```

Above will echo ‘blah’ on PHP5.3+.
But, all is not as simple as it seems. The static keyword doesn’t take inheritance into account, like $this does. Instead it tries to resolve the correct call.
The following code will fail:

```
abstract class ParentClass
{
        public static function delegate()
```

```
        {
                static::meh();
        }
}
class ImplementationClass extends ParentClass
{
        public static function meh()
        {
                echo 'blah';
        }
        public static function foo()
        {
                parent::delegate();
        }
}
ImplementationClass::foo();
```

Because 'parent' resolves to ParentClass, 'static' in delegate() resolves to ParentClass as well. But ParentClass doesn't have a method called 'meh'; the dreaded fatal error we got before is inescapable. You'll get similar results trying it with multiple levels of inheritance, a fully resolved method call (ParentClass::delegate()), or with static properties instead of methods.
In summary: if you want to use this new feature, be very aware of its limitations.

5 In conclusion

If you have any questions regarding this tutorial, please use the forums. If you want to comment you can do so below.
Thanks for reading, see you on the forums?