OOP and PHP 5.3

- **South Florida PHP Users Group**

Adam Culp
http://www.Geekyboy.com

In this presentation we will talk about OOP, and discuss the object model progression in PHP from version 4 through the newest 5.3.

I apologize in advance, but this will not be in great detail though I hope it is enough to help everyone in some way.

Procedural? Really?

- PHP was started as a procedural tool to perform quick tasks. Large adoption brought more users who wanted OOP, so here we are.
  - Everything went fairly sequential
  - Took a fair amount of labor within code to change values afterward
  - Procedural does make sense for simple tasks, such as CRON jobs to perform quick tasks.

```
<?php
function vehicle($wheels, $color, $hp) {

      // build a car using info provided
      $car = array(
                        'wheels' => $wheels,
                        'color' => $color,
                        'hp' => $hp);

      return $car;
}

$car = vehicle(4, 'red', 240);

// now continue with code that uses $car
If ($car['color'] == 'red') {
      echo 'Your car is red';
}

?>
```

Output:
Your car is red

Possible usage:
Here is the car: `<?php print_r($car); ?>`

OOP the basics

- PHP 4 brought the introduction of oop to PHP.
  - Application builds an object
  - Helps with DRY (Don't repeat yourself)
  - Simplifies structure, code still can be sequential but is more 'modular'
  - Helps keep code more readable
  - Object can be manipulated easier prior to use of the object
  - We instantiate the class by creating a "new" object, then manipulate the object by calling the different "methods" within the class.

```
<?php
Interface mobility {
        function setColor($color);
        function setHorsepower($hp);
}

class Vehicle implements mobility {
        public $color;
        public $horsepower;

        function setColor($color) {
                $this->color = $color;
        }

        function setHorsepower($hp) {
                $this->horsepower = $hp;
        }
}

Class Car extends Vehicle {
        public $wheel;

        function addWheel($n) {
                $this->wheel += $n;
        }
}

$myCar = new Car();
$myCar->setColor('red');
$myCar->setHorsepower(250);
$myCar->addWheel(3);
$myCar->addWheel(1);

print_r($myCar);
?>

Output:
Car Object([wheel]=>4[color]=>red[horsepower]=>250)
```

PHP 4 revisit

- PHP 4 had a very basic OOP presence, and looked something like this ->
  - Variables need to be defined as ‘var’.
  - Constructors carried the same name as the Class.
    - Note: the constructor always runs “automagically” when the class is instantiated
  - No visibility
  - No abstraction
  - Very simple and easy to use, but lacked many features of other OOP languages at the time.

```
<?php
class Reference {
	var $reference;

	// acted as constructor
	function Reference() {
		$this->reference = ‘dictionary’;
	}

	function getReference() {
		return $this->reference;
	}
}

$rt = new Reference();
$reference = $rt->getReference;
echo ‘reference:’ . $reference;
?>
```

Output:
reference: dictionary

PHP 5.0 brought changes

- With the rollout of PHP 5.0 there were many changes.
  - Protected data with Visibility
    - Public (default) – accessed and changed globally
    - Protected – access and changed by direct descendants
    - Private – access and changed within class only
  - Type Hinting – notice how we specify that an object of Language is passed to the constructor. This means we must create an object using the Language class first.
  - Variables no longer need the 'var' keyword
  - Constructor now defined using __construct call
  - CONSTANT values may now be assigned per-class, cannot be a variable or property or mathematical operation or function call.

```
<?php
class Reference {
	const DEFAULT_LANG = 'eng';

	private $reference;
	private $lang;

	public function __construct(Language $lang) {
		if($lang ) {
			$this->lang = $lang->esp;
		} else {
			$this->lang = DEFAULT_LANG;
		}
		$this->reference = 'dictionary';
	}

	public function getReference() {
		return $this->reference;
	}

	private function setPrivateReference() {
		$this->reference = 'my_dictionary';
	}
}

class Language {
	public $esp = 'Spanish';
}

$lang = new Language();
$rt = new Reference($lang);
$reference = $rt->getReference;
echo 'reference:' . $reference;
?>
```

Output:
Reference: Spanish

PHP 5.0 Abstraction

- Abstraction
  - Abstraction, if a class contains any abstract methods the class must also be abstract.
  - Abstracted methods must be defined by the child class.
  - Visibility in the method of the child class must be the same, or less, restricted.
  - “final” keyword prevents child classes from overriding a method
  - “clone” creates a copy of an object rather than continuing to use the same object.

```
<?php
abstract class Reference {
        public $reference;

        abstract public function read();

        public function __construct() {
                $this->reference = ‘dictionary’;
        }

        final public function getReference() {
                return $this->reference;
        }

        protected function setProtectedReference($myReference) {
                $this->reference = $myReference;
        }
}

class Letter extends Reference {
       public function read($personal) {
                $myDictionary = $personal;

                parent::setProtectedReference($myDictionary);

                return $this->reference;
       }
}

$rt = new Letter();
$reference = $rt->read(‘my english dictionary’);
$rt2 = clone $rt;

echo ‘reference:’ . $reference;
?>
```

Output:
reference: my english dictionary

PHP 5.0 Interfaces

- Interface
  - Interface, specifies which methods a class must implement.
  - All methods in interface must be public.
  - Multiple interfaces can be implemented by using comma separation
  - Interface may contain a CONSTANT, but may not be overridden by implementing class

```
<?php
interface rTemplate
{
        public function getReference();
        public function setProtectedReference();
}

class Reference implements rTemplate {
          public $reference;

          public function __construct() {
                  $this->reference = 'dictionary';
          }

          public function getReference() {
                  return $this->reference;
          }

          protected function setProtectedReference($myReference) {
                  $this->reference = $myReference;
          }
}

$rt = new Letter();
$reference = $rt->getReference();

echo 'reference:' . $reference;
?>
```

Output:
reference: dictionary

PHP 5.0 Exceptions

- Additional features with PHP 5.
  - Exceptions – throwing an exception to gracefully error out, while continuing to execute the rest of the script even after the error.
    - Notice how we still get 'Hello World' even after an exception caused by 'Division by zero'.
  - Exceptions can then be caught for logging, etc.

```
<?php
function inverse($x) {
        if (!$x) {
                throw new Exception('Division by zero.');
        } else {
                return 1/$x;
        }
}

try {
        echo inverse(5) . "\n";
        echo inverse(0) . "\n"; // trigger exception
} catch (Exception $e) {
        echo 'Caught exception: ', $e->getMessage(), "\n";
}

// continue execution
echo 'Hello World';
?>
```

Output:
0.2
Caught exception: Division by zero.
Hello World

Finally, PHP 5.3 features

- Additional features with PHP 5.3
  - Namespaces
  - Late Static Bindings
  - Jump labels (goto)
  - Closures
  - __callStatic() and __invoke()
  - Class Constants
  - Nowdoc syntax supported
  - Use Heredocs to initialize static variables and class properties
  - Heredocs with double quotes
  - Ternary operator shortcut
  - HTTP status 200 to 399 = success
  - Dynamic access to static methods
  - Exception nesting
  - mail() logging of sent email

Namespaces

- Namespaces
  - Help create a new layer of code encapsulation.
    - Keep properties from colliding between areas of your code
    - Only classes, interfaces, functions and constants are affected
  - Anything that does not have a namespace is considered in the Global namespace (namespace = “”)
  - Namespace must be declared first (except ‘declare’ statement)
  - Can define multiple namespaces in the same file.

```
<?php
declare (encoding='UTF-8');

namespace automobile;

class Automobile {
	function setType($type) {
		echo __NAMESPACE__ . "\n";
	}
}

namespace automobile\car;

class Car {
	function toyota() {
		echo "test drive\n";
	}
}

$car = new Car
$car->toyota();

// OR you can use the namespace

$auto = new \automobile\car\Car;
$auto->toyota();

?>

Output:
test drive
test drive
```

Namespaces – cont.

- Namespaces – cont.
  - You can define that something be used in the “Global” namespace by enclosing a non-labeled namespace in {} brackets. (Note: if you have multiple namespaces in the same file they must all use this notation.)
  - Use namespaces from within other namespaces, along with aliasing

```
<?php
namespace automobile;

class Automobile {
        function setType($type) {
          echo __NAMESPACE__ . “\n”;
        }
}

namespace automobile\car;

use automobile as auto;

class Car {
        function toyota() {
          echo “test drive\n”;
        }
}

namespace {
        //global code, for this to work the examples above would also
        need to use bracketed syntax
}

$automobile = new auto\Automobile;
$automobile->setType(‘none’);
?>

Output:
automobile
```

Late Static Bindings

- Late Static Binding
  - Stores the class name of the last "non-forwarded call".

```
<?php
class Automobile {
        private function type() {
          echo "Success!\n";
        }

        public function test() {
          $this->type();
          static::type();
        }
}

class Car extends Automobile {
        // empty
}

Class Truck extends Automobile {
        private function type() {
          // empty
        }
}

$car = new Car;
$car->test();
$truck = new Truck;
$truck->test(); //fails because there is no test() in Truck
?>

Output:
Success!
Success!
Success!
Fatal error: Call to private method Truck::type() from context
        'Automobile' in {file} on line n
```

Jump Labels (goto)

- Jump Labels (goto)
  - Used to jump to another section in the program.
  - Target is specified by a label followed by a colon.
  - Target must be within the same file and context.
    - Cannot jump out of a function or method, and cannot jump into one.
  - Cannot jump into any sort of loop or switch structure, but may jump out.

```
<?php
// some code here

goto a;
echo 'Foo';

// more code here

a:

echo 'Bar';
?>
```

Output:
Bar

Closures (Anonymous functions)

- Closures (Anonymous functions)
  - Allows the creation of functions which have no specific name.
  - Most useful as the value of callback parameters.
  - Can be used as the value of a variable.
  - Can inherit variables from the parent scope. (Not the same as using global variables)

```
<?php
class Cart {
      protected $products = array();
      const PRICE_SHIRT = 20.00;
      const PRICE_SCARF = 4.00;

      public function order() {
        $this->products['shirt'] = 2;
        $this->products['scarf'] = 3;
      }

      public function getTotal($tax) {
        $total = 0.00;

        $callback = function ($quantity, $product) use ($tax, &$total)
      {
                $pricePerItem =
                constant(__CLASS__ . "::PRICE_" .
      strtoupper($product));

                $total += ($pricePerItem * $quantity) * ($tax +1.0);
        };
        array_walk($this->products, $callback);

        return round($total, 2);
      }
}
?>
```

Output:
55.64

Nested Exceptions

- Nested Exceptions
  - Now your criteria within a “try” can also have another “try” nested within, thus causing two levels of failure to caught for further logging.

```
<?php
class MyException extends Exception {}

class Test {
    public function testing() {
     try {
            try {
                throw new MyException(‘foo!’);
            } catch (MyException $e) {
                throw $e;
            }
     } catch (Exception $e) {
            var_dump($e->getMessage());
     }
    }
}

$foo = new Test;
$foo->testing();
?>
```

Output:
String(4) “foo!”

New magic methods

- __callStatic()
  - Used during overloading. Gives warning to enforce public visibility and non-static declaration.
    - Triggered when invoking inaccessible methods in a static context.
- __invoke()
  - Called when a script tries to call an object as a function

No code to go with this… yet. Experiment on your own.

Nowdoc

- Nowdoc
  - nowdoc is similar to heredoc, but no parsing is done inside a nowdoc.
    - Ideal for code snippets or large blocks of text without the need for escaping.
    - Best used with static content.
  - Uses the same <<< sequence, but the following identifier is encosed in single quotes.

```
<?php
echo <<<'EOT'
My name is "$name". I am printing some
    $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should not print a capital 'A': /x41
EOT;
?>
```

Output:

```
My name is "$name". I am printing some
    $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should not print a capital 'A': /x41
```

Heredoc changes

- Heredoc
  - heredoc can now initialize static variables and class properties/constants.
  - Can now be declared (optional) using double quotes, complementing the nowdoc syntax which uses single quotes.

```
<?php
// Static variables
function foo() {
    static $bar = <<<“LABEL”
Nothing in here...
LABEL;
}

// Class properties/constants
Class foo {
    const BAR = <<<FOOBAR
Constant example
FOOBAR;

    public $baz = <<<FOOBAR
Property example
FOOBAR
}



?>
```

Constants addition

- Constants addition
  - A constant may now be declared outside a class using the ‘const’ keyword instead of ‘declare’.

```
<?php
const TEST = ‘bar’;

Function foo() {
    echo ‘foo’;
}

foo();

echo TEST;



?>
```

Ternary added shortcut

- Constants added shortcut
  - Can now use ternary for simpler returns of evaluation.
  - Instead of defining the ‘middle’ part of the operation we simply get a ‘1’ if the first expression is true. Otherwise we receive what is in the third part, as we used to.

```
<?php

$test = true;

// old way
$todo = ($test ? ‘Go’ : ‘Stop’);

echo $todo;



// added shortcut
// if $test = true then we get a true flag, otherwise we get
      the second expression as a result
$tada = ($test ?: ‘whatever’);

echo $tada;



?>
```

Output (true):
Go1

Output (false):
Stopwhatever

Questions and Resources

- **South Florida PHP Users Group**

Adam Culp http://www.Geekyboy.com

Email: adam@geekyboy.com

Resources:

http://php.net/manual/en/migration53.new-features.php