Beyond Hello World: From Scripting to Software

```
<html>
 <head>
  <title>PHP Test</title>
 </head>
 <body>
   <?php echo '<p>Hello World</p>'; ?>
 </body>
</html>
```

Patrick Schwisow

Madison PHP Conference

September 30, 2016

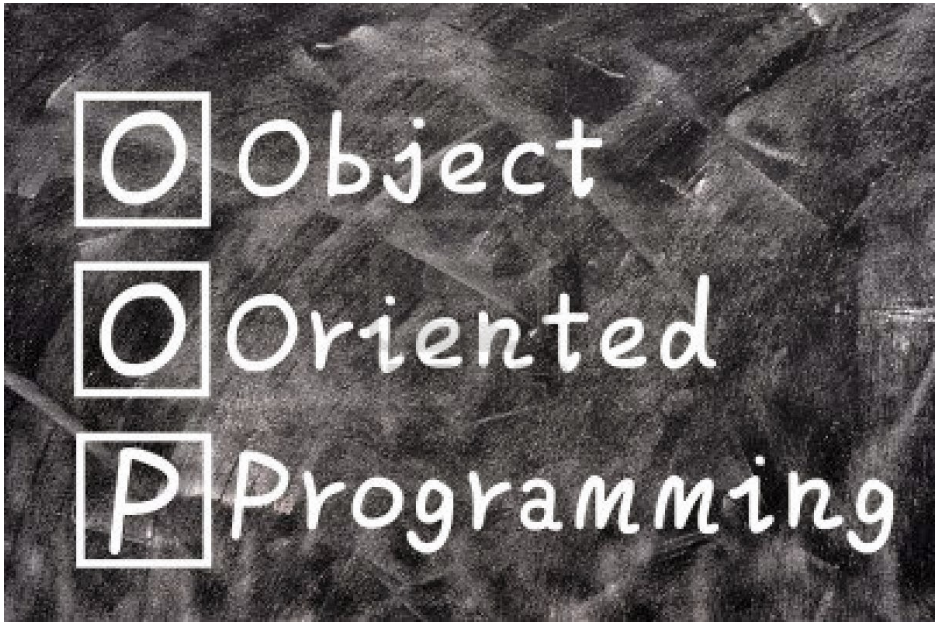
@PSchwisow

Introduction

- What is a script?
  - Small Program
  - Limited Scope
  - Limited Control Flow
- What is software?
  - Application Design / Architecture
  - Separation of Concerns
  - Reusable Components

What Do We Already Know?

- Primitive Data Types
- Arrays
- Variables
- Operators
- Control Structures
- Functions
- How to Run PHP



Section 1

OBJECT-ORIENTED PROGRAMMING

What is an Object?

- **Object** – A collection of properties and methods.
- **Property** – A variable attached to an object.
- **Method** – A function attached to an object.
- **Class** – A blueprint used to create new objects.

Creating Objects

- **Constructor** – A special function that creates a new object.
- Objects are created using the new keyword and a constructor.
- Objects can be “stored” in variables using =.
- The object itself is stored separately, the variable just points to it.
- This pointer to the object is called a **reference**.

Example 1-1: Creating objects

```php
<?php

$object = new stdClass();

$object1 = $object;

$object2 = new stdClass();

var_dump($object);
var_dump($object1);
var_dump($object2);
```

Copying and Deleting Objects

- = does NOT copy objects, it only copies the reference.
- Objects can be copied using `clone`.
- Objects are automatically deleted when all references are gone.
- Object can be deleted by using `unset()` on all references.

Example 1-2: Copying and Deleting Objects

```php
<?php

$object = new stdClass();

$objectclone = clone $object;

var_dump($object);
var_dump($objectclone);

unset($object);
unset($objectclone);
```

Creating Classes

- Classes are created using the `class` keyword.
- Class names follow the same rules as function names.
- Classes should be created before they are used.

Example 1-3: Creating Classes

```php
<?php

class Foobar
{

}

$object = new Foobar();

var_dump($object);
```

Properties and Methods: Visibility

- Every property and method has a visibility.
- Visibility determines where the property/method can be used.
- Visibility can be `public`, `private`, or `protected`.
  - `public` – accessible from anywhere
  - `private` – only accessible from the object's methods
  - `protected` – similar to private, preferred because reasons

Properties

- Defined using a visibility, a name, and optionally a default value.
- If no value is given, the default is `null`.
- Naming rules are the same as variables.
- Accessed using the `->` operator, used just like variables.
- `$` is used for defining, not used for accessing.
- `foreach` can be used to go through an object's properties.

Methods

- Defined just like a function, but with a visibility.
- Naming rules are the same as functions.
- Called using the `->` operator, used just like functions.
- Constructor must be named `__construct`, cannot return anything.

$this

- Methods have access to a special variable: `$this`.
- `$this` is used to access the current object.
- Can be used with `->` to access other methods and properties.

Example 1-4: Properties and Methods

```php
<?php

class Foobar
{
    protected $property = "abcdef";
    public function __construct() {…}
    public function method()
    {
        echo $this->property;
    }
}

// In another file:
$object = new Foobar();

$object->method();
```

Namespaces

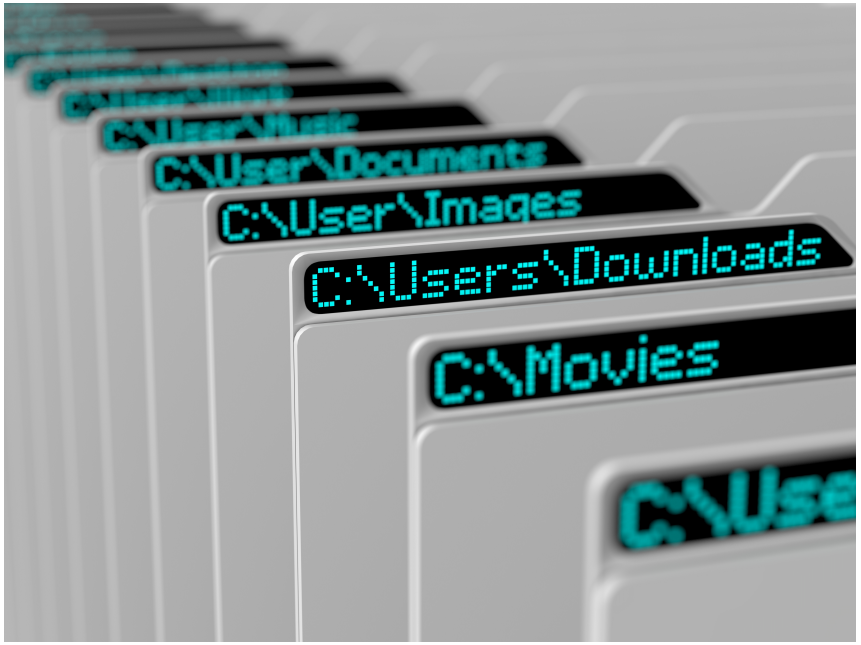
- Avoid name collisions
  - PHP's `DateTime` vs. `MyProject\DateTime`
  - `Zend\View\View` vs. `MyProject\View`
- Alias

```
use Phergie\Irc\Plugin\React\Command
    \CommandEventInterface as EventInterface;
```

- Organize / group related classes together

Example 1-5: Namespaces

```php
<?php
namespace PSchwisow\FooProject;
use PSchwisow\Other\MyClass;
use PSchwisow\Other\Foobar as OtherFoo;

class Foobar {
    public function method() {
        $o1 = new \DateTime(); //global namespace
        $o2 = new Baz(); //current namespace
        $o3 = new MyClass(); //imported namespace
        $o3 = new OtherFoo(); //aliased
    }
}

// In another file (with no namespace set):
$object = new PSchwisow\FooProject\Foobar();
```



Section 2

FILE LAYOUT AND ORGANIZATION

File Layout and Organization

- How do PHP developers tend to lay out their projects in files and directories?
- How do we organize the parts of our applications?
- How do we allow our applications to expand?

Basic Layout

```
/foo-project
/foo-project/data
/foo-project/config
/foo-project/src
/foo-project/public
```

The Document Root

- The document root is a directory (a folder) that is stored on your host's servers and that is designated for holding web pages.
- For security reasons, ONLY actual files for the user should be present – css, js, images, and the PHP files that control logic.
- Other files (configuration, PHP classes) should be parallel to the docroot.

Web Information

```
/public
/public/css
/public/fonts
/public/images
/public/index.php
```

Data Information

```
/data
/data/cache
/data/tmp
```

Configuration

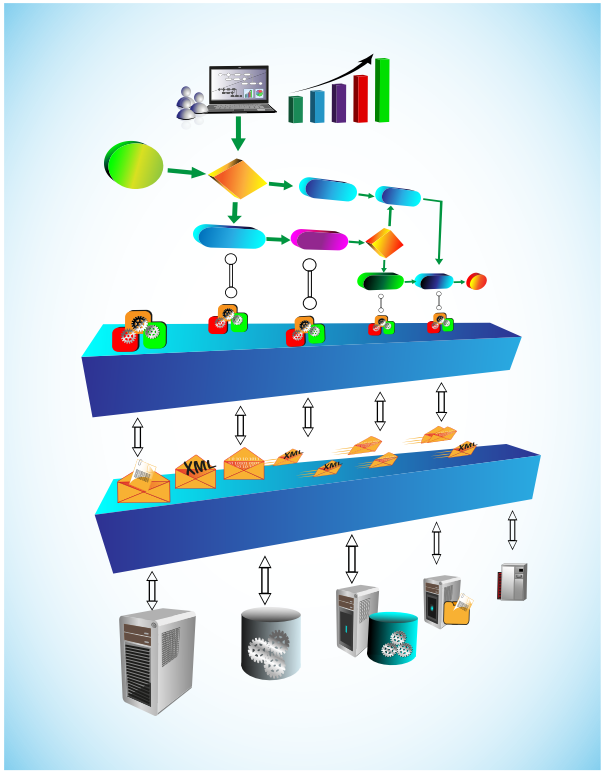
- Information that might change
- Information that might need to expand
- Specific settings for parts of your code

Example 2-1: config/config.php

```php
<?php

return [
    'name' => 'value'
];
```

Example 2-2: public/index.php

```php
<?php

$config = include '../config/config.php';

var_dump($config);
```



Section 3

APPLICATION ARCHITECTURE

Separating Application Code

- Actual HTML output – our View
- Logic to act on – our Controller
- Data rules and reading/writing – our Model

App Layout

```
/src/Foo/Model
/src/Foo/View
/src/Foo/Controller
```

Template File

- Mainly HTML
- Shortcut echo tags
- Very little logic
  - simple conditionals
  - simple loops
- May or may not include other templates

Template Class

- Holds the data to display.
- Generates HTML to return to web browser.
- Has “helpers” for common functionality.

Example 3-1: src/View/template.html

```
1  <!doctype html>
2  <html>
3      <head>
4          <title><?= $this->title ?></title>
5      </head>
6      <body>
7          <div>
8              <?= $this->message ?>
9          </div>
10     </body>
11 </html>
12
13
14
15
16
```

Example 3-2: src/View/Template.php

```php
<?php

namespace PSchwisow\FooProject\View;

class Template
{
    public function render($tpl, $data = [])
    {
        foreach ($data as $key => $value) {
            $this->{$key} = $value;
        }
        require __DIR__ . '/' . $tpl;
    }
}
```

Example 3-3: Using Templates

```php
<?php

use PSchwisow\FooProject\View\Template;

$vars = [
    'title' => 'Hello and Welcome',
    'message' => 'This is my templated app',
];
$template = new Template();
$template->render('template.html', $vars);
```

Considerations for Templates

- Escaping Output
- “Base” templates
- Set our template file location

Section 4

DEPENDENCIES AND COMPOSER

Dependencies

- **Library** – a collection of code, data, configuration, documentation, etc. that can be reused in multiple projects.
- **Dependency** – something that is required for your project to work correctly.
  - Libraries
  - PHP versions
  - PHP extensions

Composer

- You declare your project's dependencies.
- Composer determines what versions to install.
  - Reads `composer.json` to find direct dependencies.
  - Reads composer.json of libraries to find ***THEIR*** dependencies.
  - Repeat until complete (or failure).
- Installs into your project's `vendor` directory.

Example 4-1: composer.json

```
{
    "require": {
        "php": ">=5.6.0, <7.0",
        "ext-xdebug": "*",
        "pschwisow/other": "1.0.*"
    }
}
```

Project Layout with Composer

`/foo-project` (contains composer.json)

`/foo-project/data`

`/foo-project/config`

`/foo-project/src`

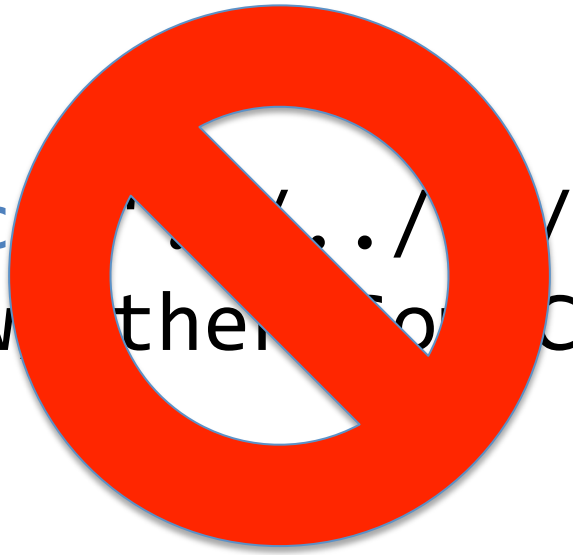
`/foo-project/public`

`/foo-project/vendor`

`/foo-project/vendor/pschwisow/other`

How do we actually use these classes?

```
require_onc      ../    /vendor/
  pschwisow    ther    Class.php';
```



MAGIC!!!

Autoloading: Not Really Magic

- When the PHP interpreter reaches a classname it does not know, the autoloader is triggered to attempt to load it.
- Autoloading Strategies:
  - PSR-4 (recommended)
  - PSR-0
  - Classmap
  - File
- Composer automatically generates an autoloader based on your `composer.json` and those in your dependencies.

Example 4-2: composer.json with autoloader

```
{
    "require": {
        "pschwisow/other": "1.0.*"
    },
    "autoload": {
        "psr-4": {
            "PSchwisow\\FooProject\\": "src/"
        }
    }
}
```

Example 4-3: public/index.php

```php
<?php

require __DIR__ . '../vendor/autoload.php';

$config = include 'config.php';

$app = new PSchwisow\FooProject\App($config);
$app->run();
```

Composer Commands

- `php composer.phar install`
  - Installs dependencies (from `composer.lock`).
  - Creates `composer.lock` file.
  - Creates autoloader files.
- `php composer.phar update`
  - Installs latest versions of all dependencies.
  - Updates `composer.lock` file.
  - Creates autoloader files.
- See https://getcomposer.org/ for more details.

Section 5

LEVERAGING OPEN SOURCE

Packagist

- Packagist (https://packagist.org/) is the main source of public Composer packages.
- Composer can use other sources with proper configuration.
- Don’t reinvent the wheel! Use what’s available, tested, and supported.

Monolog: Application Logging

- **Channel** – The name of a logger instance.
- **Handlers** – Records pass through handlers in a stack. They may be handled (output to log) and / or passed on to the next handler on the stack.
- **Log Levels** – Severity assigned to a record: DEBUG, INFO, NOTICE, WARNING, ERROR, CRITICAL, ALERT, EMERGENCY

Example 5-1: Monolog Example

```php
<?php

use Monolog\Logger;
use Monolog\Handler\SlackHandler;
use Monolog\Handler\StreamHandler;

$logger = new Logger('my_logger');

$logger->pushHandler(new StreamHandler(
    __DIR__.'/my_app.log', Logger::DEBUG));
$logger->pushHandler(new SlackHandler(…,
    Logger::CRITICAL));

$logger->debug('Log only in my_app.log.');
$logger->critical('Log both places.');
$logger->warning('Where will this log?');
```

PHPUnit: Testing Framework

- Automated tests ensure correct behavior, reduce bugs, and prevent regression.
- PHPUnit provides a structure to build and run tests.
- Testing is a huge topic. Great starting place: https://grumpy-learning.com/

Example 5-2: composer.json with more stuff

```
{
    "require": {
        "pschwisow/other": "1.0.*",
        "monolog/monolog": "1.*"
    },
    "require-dev": {
        "phpunit/phpunit": "5.5.*"
    },
    "autoload": {
        "psr-4": {
            "PSchwisow\\FooProject\\": "src/"
        }
    }
}
```

Example 5-3: Running PHPUnit

```
$ ./vendor/bin/phpunit --bootstrap
    vendor/autoload.php tests/*

PHPUnit 5.5.0 by Sebastian Bergmann and
contributors.

....................

Time: 121 ms, Memory: 4.50Mb

OK (20 tests, 39 assertions)
```

Other Popular Libraries

- phpDocumentor – Generate documentation from PHP source code / comments.
- Guzzle – HTTP client.
- Twig – Template engine.
- Symfony Console – Helps create command-line interfaces.
- Doctrine – Object-relational mapper (ORM), for abstracting database operations.

Frameworks

- Controls application flow.
- Provides structure.
- Addresses many common needs for you.
- Popular PHP Frameworks:
  - Laravel
  - Symfony
  - Zend Framework
  - Slim
  - Silex

Who Am I?
Feedback / Contact / Slides

- Software Engineer at Shutterstock
- Zend Certified Engineer – PHP 5, Zend Framework
- Founder / Organizer of Lake / Kenosha PHP
- Email: patrick.schwisow@gmail.com
- Twitter: @Pschwisow
- Slides: github.com/PSchwisow/Miscellaneous/
- Joind.in: https://joind.in/e/madisonphp2016
- Portions of this presentation adapted from https://github.com/PHPEmbark/curriculum