

PHP OOP

An Object Oriented Programming Primer

Eli White

Vice President — One for All Events

@EliW



Hang on to your hat



OOP Review

Just the basics ma'am



What is OOP?

Object Oriented Programming (OOP) is the idea of putting related data & methods that operate on that data, together into constructs called **classes**.

When you create a concrete copy of a class, filled with data, the process is called **instantiation**.

An instantiated class is called an **object**.

Basic Class Structure

Two types of constructs

properties:

The variables that hold the data

methods:

The functions that hold the logic

```
class Animal
{
    // The following are properties:
    public $weight;
    public $legs = 4; // A default value

    // The following is a method:
    public function classify() { /* ... */ }
    public function setFoodType($type) { /* ... */ };
}
```

Instantiation & Access

instantiate
by using the **new**
keyword

access
properties and
methods via **->**

```
class Animal
{
    public $weight;
    public $legs = 4;

    public function classify() { /* ... */ }
    public function setFoodType($type) { /* ... */ };
}

$horse = new Animal();

echo $horse->legs;
$horse->setFoodType("grain");
```

Constructors

A class can have a method called a constructor.

This method, named `__construct`, allows you to pass values when instantiating.

```
class Animal
{
    // The following are properties:
    public $weight;
    public $legs = 4; // A default value

    // The following is a method:
    public function classify() { /* ... */ }
    public function setFoodType($type) { /* ... */ };

    // Constructor:
    public function __construct($weight) {
        $this->weight = $weight;
    }
}

$cat = new Animal(13.4);
```

Privacy & Visibility

Within a class, methods & properties have three levels of privacy

public
modified & accessed by anyone

private
only accessed from within the class itself

protected
only be accessed from within the class, or within a child

```
class Animal
{
    protected $weight; // Accessible by children
    public $legs = 4; // Publicly accessible

    public function __construct($weight) {
        $this->setWeight($weight);
    }

    private function setWeight($weight) {
        $this->weight = $weight;
    }
}

$cat = new Animal(13.4);
echo $cat->weight; // Fatal Error
```


Static & Constants

Constants are immutable properties.

Methods and properties can be **static** making them accessible without instantiation.

Access both via ::

```
class Math
{
    const PI = 3.14159265359; // Constant:

    public static $precision = 2; // Static property:

    // Static method:
    public static function circularArea($radius) {
        $calc = self::PI * pow($radius, 2);
        return round($calc, self::$precision);
    }
}

Math::$precision = 4;
$answer = Math::circularArea(5);
```

Referencing Classes from within

`$this->`

References the object

`self::`

Reference static & const

`<class_name>::`

Same as self (within context)

`static::`

Late Static Binding
(more on this later)

```
class Math
{
    const PI = 3.14159265359; // Constant
    public static $precision = 2; // Static property
    private $last; // Private property

    public function circularArea($radius) {
        $this->last = Math::PI * pow($radius, 2);
        return round($this->last, self::$precision);
    }
}
```

Inheritance

Learning from the past



Inheritance

Through the **extends** keyword, this allows one class to be a copy of another and build upon it. The new class is called the child, and the original the parent.

```
class Person {
    public $first;
    public $last;

    public function __construct($first, $last) {
        $this->first = $first;
        $this->last = $last;
    }

    public function name() {
        return "{$this->first} {$this->last}";
    }
}
```

```
class Employee extends Person {
    public $title;

    public function name() {
        return $this->title;
    }
}

class Intern extends Employee {
    protected $title = 'Intern';
}
```

Accessing Your Parent

You can also
access
properties &
methods in the
parent by using
the keyword
parent::

```
class Person {
    public $name;

    public function __construct($name) {
        $this->name = $name;
    }

    public function announcement() {
        return "{$this->name}";
    }
}

class Employee extends Person {
    public $job;

    public function announcement() {
        return parent::announcement . ", " . $this->job;
    }
}
```

Stopping Extension

Classes can prevent children overriding a method

Uses **final** keyword

Attempting to override a **final** causes a fatal error

```
class A {
    public function __construct() {
        $this->notify();
    }

    final public function notify() {
        echo "A";
    }
}

// Causes a fatal error:
class B extends A {
    public function notify() {
        echo "B";
    }
}
```

Final Classes

Entire classes may also be declared as **final** to prevent extension completely

```
final class A
{
    public function __construct() {
        $this->notify();
    }

    public function notify() {
        echo "A";
    }
}
```

Abstracts & Interfaces

The contracts that we make with ourselves



Abstract Class

Defines an
'incomplete' class
using **abstract**
keyword on both
class & methods

Children need to
implement all
abstract methods
for the parent

```
abstract class DataStore
{
    // These methods must be defined in the child class
    abstract public function save();
    abstract public function load($id);

    // Common properties
    protected $data = [];

    // Common methods
    public function setValue($name, $value) {
        $this->data[$name] = $value;
    }
    public function getValue($name) {
        return $this->data[$name];
    }
}
```

Abstract Contract

All abstract methods
must be implemented
or this class must be
abstract as well

Method signatures
must match **exactly**

```
class FileStore extends DataStore {
    private $file;

    public function load($id) {
        $this->file = "/Users/eli/{$id}.json";
        $input = file_get_contents($this->file);
        $this->data = (array)json_decode($input);
    }

    public function save() {
        $output = json_encode($this->data);
        file_put_contents($this->file, $output);
    }
}

$storage = new FileStore();
$storage->load('Ramsey White');
$storage->setValue('middleName', 'Elliott');
$storage->save();
```

Interfaces

An **interface** defines method signatures that an implementing class must provide

Similar to abstract methods, but **sharable** between classes

No code, No properties just an interoperability framework

```
interface Rateable
{
    public function rate(int $stars, $user);
    public function getRating();
}

interface Searchable
{
    public function find($query);
}
```

Interface Implementation

Uses the
implements
keyword

One class may
implement multiple
interfaces

```
class StatusUpdate implements Rateable, Searchable {
    protected $ratings = [];

    public function rate(int $stars, $user) {
        $this->ratings[$user] = $stars;
    }

    public function getRating() {
        $total = array_sum($this->ratings);
        return $total/count($this->ratings);
    }

    public function find($query) {
        /* ... database query or something ... */
    }
}
```

Interface Extension

It is possible for an **interface** to extend another **interface**

Interfaces can provide **constants**

```
interface Thumbs extends Rateable {
    const THUMBUP = 5;
    public function thumbsUp();
    public function thumbsDown();
}

class Chat extends StatusUpdate implements Thumbs {
    public function rate(int $stars, $user) {
        $this->ratings[] = $stars;
    }
    public function thumbsUp() {
        $this->rate(self::THUMBUP, null);
    }
    public function thumbsDown() {
        $this->rate(0, null);
    }
}
```

Traits

When you want more than a template



Traits

Enable horizontal
code reuse

Create code and
inject it into
different classes

Contains actual
implementation

```
// Define a simple, albeit silly, trait.
trait Counter
{
    protected $_counter;

    public function increment() {
        ++$this->_counter;
    }

    public function decrement() {
        --$this->_counter;
    }

    public function getCount() {
        return $this->_counter;
    }
}
```

Using Traits

Inject into a class
with the **use**
keyword

A class can include
multiple **traits**

```
class MyCounter
{
    use Counter;

    /* ... */
}

$counter = new MyCounter();
$counter->increment();
echo $counter->getCount(); // 1
```


Namespaces

Who are you again?



Defining Namespaces

Namespaces let you have multiple libraries with identically named classes & functions.

```
<?php
namespace WorkLibrary;

class Database {
    public static connect() { /* ... */ }
}
```

Define with the **namespace** keyword to include all code that follows.

```
<?php
namespace MyLibrary;

class Database {
    public static connect() { /* ... */ }
}
```

Sub-namespaces

Use a backslash '\'
to create these

```
<?php
namespace MyLibrary\Model;

class Comments {
    public __construct() { /* ... */ }
}
```

```
<?php
namespace Treb\Framework\Utility;

class Cache {
    public __construct() { /* ... */ }
}
```

Using Namespaces

Use the fully qualified name

```
$db = MyProject\Database::connect();  
$model = new MyProject\Model\Comments();
```

Import via the **use** keyword

```
use MyProject\Database;  
$db = Database::connect();
```

Alias when importing via **as** keyword

```
use MyProject\Model\Comments as MyCo;  
$model = new MyCo();
```

Reference builtin classes with top level ****

```
$images = new \DateTime();
```

Late Static Binding

It's fashionable to show up late



You mentioned this before

Traditionally when you use **self::** in a parent class you always get the value of the parent.

```
class Color {
    public static $r = 0;
    public static $g = 0;
    public static $b = 0;

    public static function hex() {
        printf("#%02x%02x%02x\n",
            self::$r, self::$g, self::$b);
    }
}
```

```
class Purple extends Color{
    public static $r = 78;
    public static $g = 0;
    public static $b = 142;
}
```

```
Color::hex(); // Outputs: #000000
Purple::hex(); // Outputs: #000000 - Wait what?
```

Enter Late Static Binding

By using the **static::** keyword it will call the child's copy.

```
class Color {
    public static $r = 0;
    public static $g = 0;
    public static $b = 0;

    public static function hex() {
        printf("#%02x%02x%02x\n",
            static::$r, static::$g, static::$b);
    }
}

class Purple extends Color{
    public static $r = 78;
    public static $g = 0;
    public static $b = 142;
}

Color::hex(); // Outputs: #000000
Purple::hex(); // Outputs: #4e008e - Right!
```

Affects Methods Too

Allows a parent to
rely on a child's
implementation of a
static method.

const work as well

```
class Color {
    public $hue = [0,0,0];
    public function __construct(Array $values) {
        $this->hue = $values;
    }
    public function css() {
        echo static::format($this->hue), "\n";
    }
    public static function format(Array $values) {
        return vsprintf("#%02x%02x%02x", $values);
    }
}

class ColorAlpha extends Color{
    public static function format(Array $values) {
        return vsprintf("rgba(%d,%d,%d,%0.2f)", $values);
    }
}

$purple = new Color([78,0,142]);
$purple->css(); // Outputs: #4e008e
$purple50 = new ColorAlpha([78,0,142,0.5]);
$purple50->css(); // Outputs: rgba(78,0,142,0.50)
```


Magic Methods

Part of the fairy dust that makes PHP sparkle



Why do we need magic?

All magic methods start with `__`

Allow for code that is not directly called to run

Often have counterparts, so just as `__construct()` we have `__destruct()` which runs on object cleanup

```
class UserORM {
    public $user;
    private $db;

    function __construct($id, PDO $db) {
        $this->db = $db;
        $sql = 'SELECT * FROM user WHERE id = ?';
        $stmt = $db->prepare($sql)->execute([$id]);
        $this->user = $stmt->fetchObject();
    }

    function __destruct() {
        $sql = 'UPDATE user
              SET name = ?, email = ? WHERE id = ?';
        $stmt = $db->prepare();
        $stmt->execute($this->user->email,
                     $this->user->name, $this->user->id);
    }
}

$eliw = new User(37);
$eliw->user->email = 'eli@eliw.com';
```

__get and __set

__get

Called when an unknown property is read

__set

Called when an unknown property is written to

Often used for storage classes & overloading

```
class Data
{
    protected $_data = [];

    public function __set($name, $value) {
        $this->_data[$name] = $value;
    }

    public function __get($name) {
        return isset($this->_data[$name])
            ? $this->_data[$name] : NULL;
    }
}

$o = new Data();
$o->neat = 'Something';
echo $o->neat;
```

__isset and __unset

__isset

Called when **isset()** is requested on an unknown property

__unset

Called when **unset()** is requested on an unknown property

```
class DataAll extends Data
{
    public function __isset($name) {
        return isset($this->_data[$name]);
    }

    public function __unset($name) {
        unset($this->_data[$name]);
    }
}

$o = new DataAll();
$o->phone = 'iPhone';
$o->desktop = true;
if (isset($o->desktop)) {
    unset($o->phone);
}
```

__call and __callStatic

These two methods are called when you attempt to access an undeclared method

```
class Methods
{
    public function __call($name, $args) {
        $pretty = implode($args, ",");
        echo "Called: {$name} with ({$pretty})\n";
    }

    public static function __callStatic($name, $args) {
        $count = count($args);
        echo "Static call: {$name} with {$count} args\n";
    }
}

// Output - Static call: sing with 2 args
Methods::sing('Barbara', 'Ann');

// Output - // Called: tea with (Earl Gray,Hot)
$m = new Methods();
$m->tea('Earl Gray', 'Hot');
```

__invoke & __toString

__invoke
allows your object
to be called as a
function.

```
class TwoPower {  
    public function __invoke($number) {  
        return $number * $number;  
    }  
}  
  
$o = new TwoPower();  
$o(4); // Returns: 16
```

__toString
determines what
happens when you
echo your class.

```
class Doubler {  
    private $n;  
    public function __construct($number) {  
        $this->n = $number;  
    }  
    public function __toString() {  
        return "{$this->n} * 2 = " . $this->n * 2;  
    }  
}  
  
$four = new Doubler(4);  
echo $four; // Output: 8
```

And more...

__sleep() and **__wakeup()**
control how your object is
serialized & unserialized.

__set_state() lets you
control exported properties
when `var_export` is

__debugInfo() lets you
change what a `var_dump` of
your object shows.

__clone() lets you modify
your object when it
becomes cloned.

Pardon a brief commercial interruption



PHP[TEK] 2017

The Premier PHP Conference
12th Annual Edition

MAY 24-26 — ATLANTA

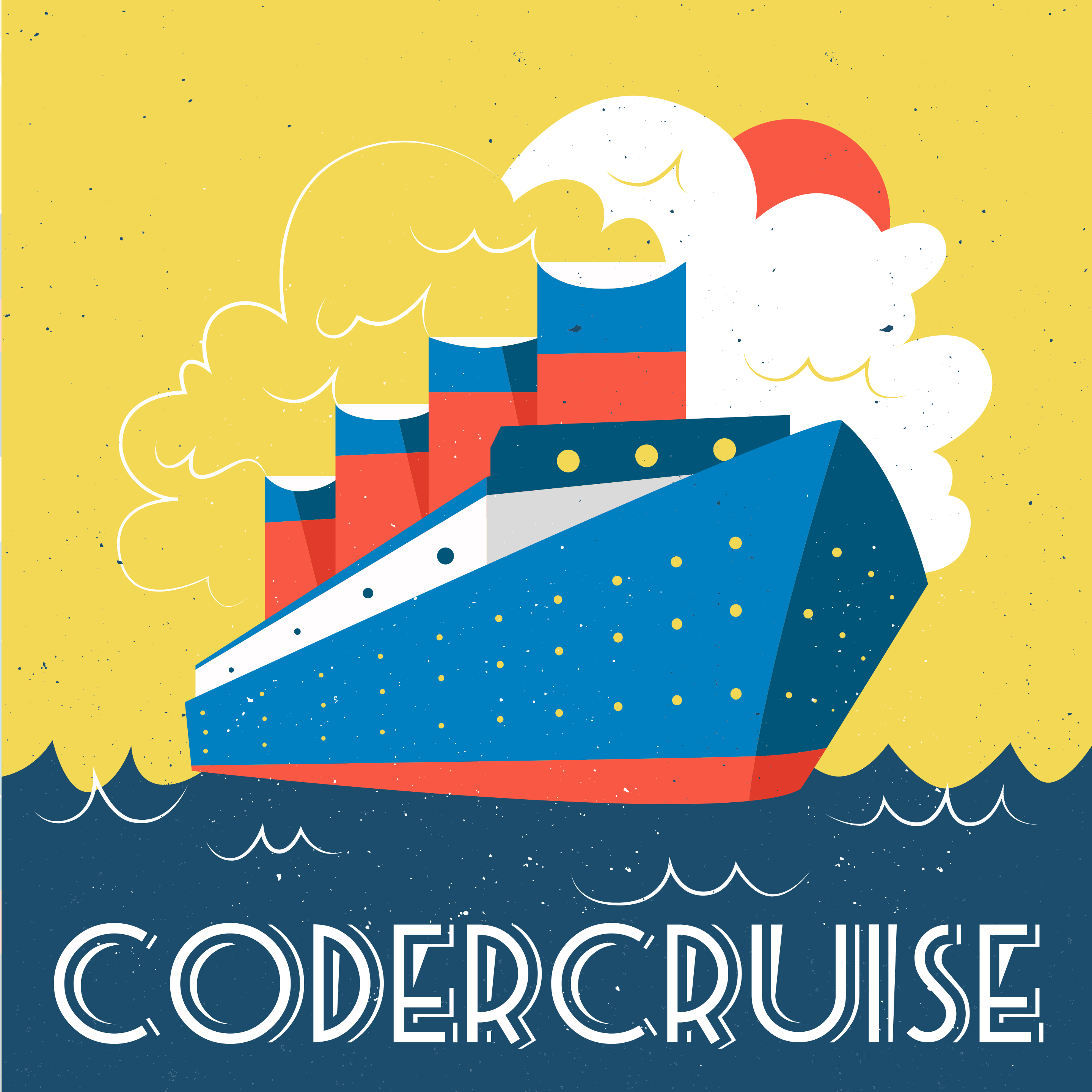
tek.phparch.com

CODERCruise.COM

JULY 16-23, 2017

PORT OF NEW ORLEANS

***Montego Bay
Grand Cayman
Cozumel***



CODERCruise

For this presentation & more:
eliw.com

Twitter: @EliW

One for All Events:
www.oneforall.events



*One for All
Events*