Essential Drupal 8 Developer Training

Jonathan Daggerhart

Introduction to Object Oriented Programming

Jonathan Daggerhart

Introduction to Object Oriented Programming

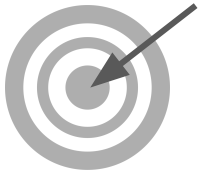
Jonathan Daggerhart

- **Developer at Hook 42**
- **Organizer for Drupal Camp Asheville**

Drupal.org: daggerhart
Twitter: @daggerhart
Blog: https://www.daggerhart.com

Drupal Camp Asheville
Site: https://drupalasheville.com
Twitter: @drupalasheville

What we will cover



1. **Vocabulary** - how to talk about OOP and its concepts.
2. **Concepts**
   a. Classes
   b. Objects
   c. Encapsulation
   d. Inheritance
   e. Interfaces
3. **Syntax** - how to write classes and use objects

Getting to Objects

A story of Encapsulation & Abstraction

Getting to Objects

From Calculation to Calculator

Here we have a calculation that represents a business problem.

We need to calculate a value that is a part of some common operation the business must perform.

Let's convert this simple operation into a generic and reusable class.

```php
<?php

$total = 0;

$total = $total + 8;

$total = $total - 2;

$total = $total / 2;

$total = $total * 6;

print $total;

// 18
```

Getting to Objects

Encapsulation & Abstraction - Vocabulary

Abstraction - Making some section of code more generic and reusable. Generally involves providing parameters of a specific type to an operation.

Encapsulation - Wrapping related functionality and values together within an abstraction.

```php
$total = 1 + 3;
// 4

function add($a, $b) {
  return $a + $b;
}

add(1, 3);
// 4
```

```php
$total = 1 + 3;
$total = $total - 2;
// 2

function calculate($a, $b, $c) {
  $total = $a + $b;
  $total = $total - $c;
  return $total;
}

calculate(1, 3, 2);
// 2
```

Getting to Objects

Encapsulation

First encapsulation, a reusable function that contains our calculation.

Now we can use our calculation function multiple times throughout the code base, but it is still limited to exactly one calculation.

```php
function calculation() {
  $total = 0;

  $total = $total + 8;

  $total = $total - 2;

  $total = $total / 2;

  $total = $total * 6;

  return $total;
}

print calculation(); // 18
```

Getting to Objects

Abstraction

First abstraction, providing the starting total as a parameter to our calculation function.

Now the calculation function has become generic. It can be run on any number.

```php
function calculation($total = 0) {
  $total = $total + 8;

  $total = $total - 2;

  $total = $total / 2;

  $total = $total * 6;

  return $total;
}

print calculation(); // 18
print calculation(4); // 30
```

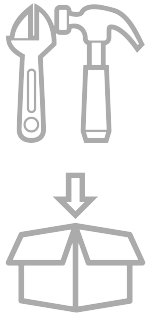
Getting to Objects

More Abstractions

Another level of abstraction. Breaking our code into multiple functions that are each reusable.

```php
function add($a, $b) {
  return $a + $b;
}

function subtract($a, $b) {
  return $a - $b;
}

function divide($a, $b) {
  return $a / $b;
}

function multiply($a, $b) {
  return $a + $b;
}
```

```php
function calculation($total = 0) {
  $total = add($total, 8);

  $total = subtract($total, 2);

  $total = divide($total, 2);

  $total = multiply($total, 6);

  return $total;
}

calculation(); // 18
```



Getting to Objects

Encapsulate our Abstractions

Let's encapsulate these functions into a class and instantiate it.

Related Functions → **Class** **Object**

```php
function add($a, $b) {
  return $a + $b;
}

function subtract($a, $b) {
  return $a - $b;
}

function divide($a, $b) {
  return $a / $b;
}

function multiply($a, $b) {
  return $a + $b;
}
```

```php
class Calculator {

  public $total = 0;

  function add($a) {
    $this->total += $a;
  }

  function subtract($a) {
    $this->total -= $a;
  }

  function divide($a) {
    $this->total = $this->total / $a;
  }

  function multiply($a) {
    $this->total = $this->total * $a;
  }
}
```

```php
$calculator = new Calculator();

$calculator->add(8);

$calculator->subtract(2);

$calculator->divide(2);

$calculator->multiply(6);

print $calculator->total; // 18
```

Getting to Objects

Class Vocabulary & Syntax

Class - Template for an object.

Property - A variable within the top level scope of a class.

Method - A function within the top level scope of a class.

Visibility - Ability to access properties and methods from other parts of the system.

Syntax & Keywords

- `class`
- `public, private, protected`
- `$this`

```php
class Calculator {

  public $total = 0;

  function add($a) {
    $this->total += $a;
  }

  function subtract($a) {
    $this->total -= $a;
  }

  function divide($a) {
    $this->total = $this->total / $a;
  }

  function multiply($a) {
    $this->total = $this->total * $a;
  }
}
```

Getting to Objects

Object Vocabulary & Syntax

Object - Instance of a class.

Instance - *noun* - Concrete occurrence of a class. Synonymous with “object”.

Instantiate - *verb* - The act of creating an “instance”.

Syntax & Keywords

- new
- ->

```php
$calculator = new Calculator();
$calculator->add(8);
$calculator->subtract(2);
$calculator->divide(2);
$calculator->multiply(6);
print $calculator->total; // 18


$calculator2 = new Calculator();
$calculator2->add(10);
$calculator2->divide(2);
$calculator2->multiply(3);
print $calculator2->total; // 15
```

Getting to Objects

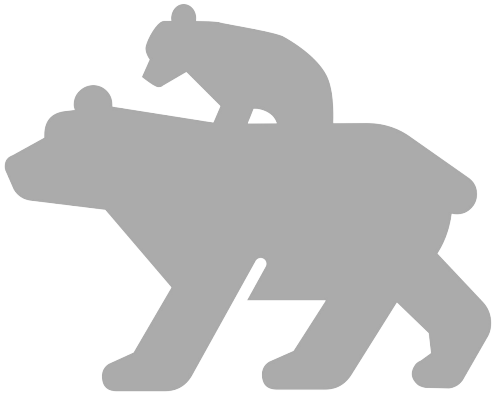
Class & Object Review

Class

```php
class Calculator {

  public $total = 0;

  function add($a) {
    $this->total += $a;
  }

  function subtract($a) {
    $this->total -= $a;
  }

  function divide($a) {
    $this->total = $this->total / $a;
  }

  function multiply($a) {
    $this->total = $this->total * $a;
  }
}
```

Object

```php
$calculator = new Calculator();

$calculator->add(8);

$calculator->subtract(2);

$calculator->divide(2);

$calculator->multiply(6);

print $calculator->total; // 18
```



Inheritance

Class Subtyping

Inheritance

Extending a Class with a new Class

Classes can inherit the properties and methods of another class by use of the `extends` keyword. This class has access to everything the `Calculator` class defines.

A class can only extend one other class. Child classes can override the methods and properties of the parent.

Syntax & Keywords

- `extends`
- `parent::`

```php
class WholeNumberCalculator extends Calculator {

  /**
   * Children classes can override the methods
   * of its parent.
   */
  public function divide($a) {
    parent::divide($a);

    round($this->total);
  }

  /**
   * Children can add new properties and
   * methods.
   */
  public function isOdd() {
    return ($this->total % 2) === 1;
  }
}
```

Inheritance

Child Class Example

```php
class WholeNumberCalculator extends Calculator {

  /**
   * Children classes can override the methods
   * of its parent.
   */
  public function divide($a) {
    parent::divide($a);

    round($this->total);
  }

  /**
   * Children can add new properties and
   * methods.
   */
  public function isOdd() {
    return ($this->total % 2) === 1;
  }
}
```

```php
$object = new WholeNumberCalculator();

$object->add(10);

$object->divide(3);

print $object->total;
// 3
```

Inheritance

Another Example - Animals

```php
class Animal {
  public $legs = 0;
  public $noise = '';

  public function vocalize($words = '') {
    print $this->noise . "!";
  }
}

class Labrador extends Animal {
  public $legs = 4;
  public $noise = 'bark';
}

class Parrot extends Animal {
  public $legs = 8;
  public $noise = 'chirp';

  public function vocalize($words = '') {
    print "{$words}. {$this->noise}";
  }
}
```

```php
$Bandit = new Labrador();
$Bandit->vocalize();
// bark!

$Polly = new Parrot();
$Polly->vocalize("Polly want a cracker");
// Polly want a cracker. chirp
```

Interfaces

A Contract for Class Methods

Interfaces

Vocabulary & Syntax of Interfaces

An interface provides strict method signature specifications for a class that implements it. A common analogy is that of a “contract” that the class accepts when implementing.

Interfaces can define methods but not properties.

Syntax & Keywords

- `interface`
- Methods do not have code.

```php
/**
 * Interfaces are like templates for classes.
 * Another common analogy is that they are a
 * "contract" that a class must adhere to.
 *
 * Define a new interface withe the "interface"
 * keyword.
 */
interface Dog {

  /**
   * Interfaces can define methods, but not
   * properties.
   *
   * Methods defined on an interface do not
   * contain code, only the signature of the
   * method that the class must define itself.
   */
  public function bark();
}
```

Interfaces

Implementing an Interface

A class uses an interface with the "`implements`" keyword. When implemented, the class **must** contain the methods defined by all interfaces it implements.

Syntax & Keywords

- `implements`
- A `class` will "`implements`" an interface.

```php
interface Dog {
  public function bark();
}

/**
 * A class uses an interface
 * with the "implements" keyword.
 */
class Labrador implements Dog {

  public function bark() {
    print "bark! bark! bark!";
  }
}
```

Interfaces

Mixing Inheritance w/ Interfaces

Classes can both extend another class and implement interfaces.

```php
class Animal {
  public $legs = 0;
  public $noise = '';

  public function vocalize() {
    print $this->noise . "!";
  }
}

interface Dog {
  public function bark();
}
```

```php
class Labrador extends Animal implements Dog {
  public $legs = 4;
  public $noise = "bark";

  public function vocalize() {
    $this->bark();
  }

  public function bark() {
    for( $i = 0; $i < 3; $i++) {
      print $this->noise . " ";
    }
  }
}

$Bandit = new Labrador();
$Bandit->vocalize();
// bark bark bark !
```

Interfaces

Another Example - Refactoring Animals

```php
class Animal {
  public $legs = 0;
  public $noise = '';

  public function vocalize($words = '') {
    print $this->noise . "!";
  }
}

class Labrador extends Animal {
  public $legs = 4;
  public $noise = 'bark';
}

class Parrot extends Animal {
  public $legs = 8;
  public $noise = 'chirp';

  public function vocalize($words = '') {
    print "{$words}. {$this->noise}";
  }
}
```

```php
interface Animal {

  public function getLegs();

  public function getNoise();
}

interface Dog {
  public function bark();
}

class Labrador implements Animal, Dog {

  public function getLegs() { return 4; }

  public function getNoise() { return 'bark'; }

  public function bark() {
    for( $i = 0; $i < 3; $i++) {
      print $this->getNoise() . "!";
    }
  }
}
```

Object Oriented Takeaways

Concepts to Remember

Class - A template for a “thing” that is modeled by the system.

Inheritance - Subtyping a class. Extending a class with a new class.

Interfaces - Defining method signatures that a class must implement.

Object - An instance of a class.

```php
interface StrictContract {
  public function aMethod();
}

interface AnotherContract {
  public function anotherMethod();
}

class SomeThing {
  public $property;
  public function method() {}
}

class Thing
  extends SomeThing
  implements StrictContract, AnotherContract
{

  public function aMethod() {}
  public function anotherMethod() {}
}

$object = new Thing();
```

Questions?

Concepts, Vocabulary, or Syntax?

Class - A template for a “thing” that is modeled by the system.

Inheritance - Subtyping a class. Extending a class with another class.

Interfaces - Defining method signatures that a class must implement.

Object - An instance of a class.

```php
interface StrictContract {
  public function aMethod();
}

interface AnotherContract {
  public function anotherMethod();
}

class SomeThing {
  public $property;
  public function method() {}
}

class Thing
  extends SomeThing
  implements StrictContract, AnotherContract
{

  public function aMethod() {}
  public function anotherMethod() {}
}

$object = new Thing();
```

Thanks!

Jonathan Daggerhart

Bonus: Your Goal

Programming to Interfaces

Write code that expects interfaces instead of classes.

Type Hinting - providing the type of data a parameter expects.

```php
/**
 * Interface describes strict method specifications.
 */
interface DoesStuff {
  public function doStuff();
}

/**
 * "Something" must implement doStuff();
 */
class Something implements DoesStuff {
  public function doStuff() {
    print "This is awesome. I'm doing stuff.";
  }
}

/**
 * Somewhere else in the system...
 * The point is that it expects the interface
 * as a parameter.
 *
 * @param \DoesStuff $specialThing
 */
function UseAnObject(DoesStuff $specialThing) {
  $specialThing->doStuff();
}
```