INFO I-CE9224: Introduction to PHP Programming

Session 10
August 22, 2012

Resources

http://davehauenstein.com/nyu/INFO1-CE9224-2012-Summer

Username: nyuscps
Password: $nyuscps$

Class 10 Agenda

- Intro to OOP (Object Oriented Programming)
- Classes and Objects
  - Visibility (public, protected, private)
  - Properties
  - Methods
- Intro to Design Patterns
- Intro to MVC Design Pattern

Object Oriented Programming (OOP)

Problem

- PHP is Very Easy!
  - PHP was designed to be written directly into web pages.
    - You get immediate results, and are tempted to keep trying out new things.
- These are all great! So what's the problem?

Problem

- Code written in an ad hoc fashion becomes a nightmare to maintain, especially as the project gets bigger.
- Bugs are hard to track down.
- Things that are second-nature to you may be causing other programmers to spend hours to solving a trivial task.

Why?

- Why would it take other programmers longer than you to work on your code?
- Programming can be like a brain dump.
- Other people think of solving the problem in a completely different way.
- Code is duplicated all over the place and changing it once, doesn't fix all the bugs.
- etc...

Solving the Problem

- OOP allows you to abstract implementation details behind a clean API: encapsulation.
- OOP improves reusability through inheritance.
- There are dozens (if not, hundreds) of design patterns that will solve common problems for you (ex MVC).

OOP: Words of Advice

Object Oriented Programming requires a paradigm shift in your thinking. It's fundamentally different thinking about how to solve a problem using OOP vs. writing it procedurally.

There will be several "ah ha" moments along the way where your understanding will grow.

Classes and Objects

Classes and Objects

Understanding the relationship between classes and objects will give you the "ah ha!" moment that gets you excited about OOP.

Classes

```php
<?php

class Car
{
    protected $color;
    protected $brand;
    protected $topSpeed;

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

    public function getColor() {
        return $this->color;
    }

    // etc...
}
```

A class is a code template used to generate objects.

Variables are called “properties”.

Functions are called “methods”.

Objects

- If a class is a template for generating objects, an object is data that is structured according to the template defined by the class.
- An object is an *instance* of a class.
- An object is of the type defined by the class.

Car Class

```php
<?php

class Car
{
    protected $color;
    protected $brand;
    protected $topSpeed;

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

    public function getColor() {
        return $this->color;
    }

    // etc...
}
```

Car Objects

Each object is an instance of the Car class.

```php
$car1 = new Car();
$car2 = new Car();

$car1->setColor('black');
$car2->setColor('red');

echo $car1->getColor();   // prints out 'black'
echo $car2->getColor();   // prints out 'red'
```

Car Objects

```
var_dump($car1)
```

```
object(Car)#1 (3) {
  ["color":protected]=>
  string(5) "black"
  ["brand":protected]=>
  NULL
  ["topSpeed":protected]=>
  NULL
}
```

Classes

Think of a class as a mold in which objects are created from.

A Look at the Syntax

Syntax

- The $this keyword.
- Defining a class. class Keyword.
- Defining class properties.
- Defining class methods.
- Creating an object by instantiating a new class instance.
- Accessing class properties.
- Accessing class methods.

$this

- Inside of classes the $this keyword refers to an instance of the object itself.
- The $this keyword has access to most levels of visibility (public, protected, private).
- Used to reference properties and methods.
- More on this at inheritance.

```php
class Car
{
    const NUM_TIRES = 4;

    public $model;

    protected $color;
    protected $brand;
    protected $topSpeed;

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

    public function getColor() {
        return $this->color;
    }
}

$car = new Car();
$car->setColor('black');
$car->model = 'm3';
echo $car->getColor(        );
echo $car->model;
```

Constructor

- A constructor is a special function that gets called when an object is instantiated.
- Used to ensure required properties are set, perform preliminary setup operations.
- public function __construct([$args...]) {...}

```php
class Car
{
    const NUM_TIRES = 4;

    public $model;

    protected $color;
    protected $brand;
    protected $topSpeed;

    public function __construct($color, $brand) {
        $this->setColor($color);
        $this->brand = $brand;
    }

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

    public function getColor() {
        return $this->color;
    }
}
```

Constructor

```
$car = new Car('black', 'BMW');
echo $car->getColor(); // returns 'black'
```

Both constructor arguments were required.

They must both be specified when creating a new instance of the class.

Constructor

```php
public function __construct($color = null) {
    if (null !== $color) {
        $this->setColor($color);
    }
}
```

Works like any other function or method.

Specifying a default value makes the argument optional.

Visibility

- Defines **“who”** can access properties and methods from **“where”**.
- Keywords: public, protected, private.
- Properties and methods declared public are available to any code that is using an instance of an object.
- Protected and private are not.
- More on protected and private at inheritance.

Rule of Thumb: Properties Never Public

- Properties should be declared protected or private.
- Use “setters” and “getters” to access them.
- This is a best practice so that object interfaces can be kept consistent.
- Sometimes you want to operate on a property before setting or getting it.

Using Setters

The color property is declared protected.

A public setter is used to set the value and perform additional operations (validation).

```
public function setColor($color) {
    $okColors = array('black', 'red', 'blue');
    if(!in_array($color, $okColors)) {
        throw new Exception('Invalid color');
    }

    $this->color = $color;
}
```

Inheritance

Inheritance

- Mechanism in which one or more classes can be derived from a base class.
- A class that inherits from another is said to subclass it.
- The relationship is often described in terms of parents and children.
- A child class is derived from and inherits characteristics from the parent.

Inheritance

- Children inherit properties and methods from the parent.
- Children classes often add additional functionality or change functionality of the parent.
- A child class “extends” the parent class.

Inheritance Use Case

- Imagine an online store that sells multiple types of products.
- Music albums and basketballs are two of the products.
- Some properties they have in common, others are different.
- Some functionality is the same, others are different.

Inheritance Use Case

Album

- Name
- Category
- Description
- Price
- Length
- Artist
- Producer

Basketball

- Name
- Category
- Description
- Price
- Color
- Brand
- IsRegulation

Inheritance Use Case

- We can create one class called ShopProduct that has all of these properties.
- Now we have Basketball stuff mixed with MusicAlbum stuff.
- What happens when we add more products? Maintainability nightmare.

Inheritance Use Case

- We can create two distinct classes to represent these objects, w/no parent class.
- We'd be duplicating a lot of work: getPrice(), setPrice(), etc...
- And remember, code duplication is bad.

Inheritance Use Case

- We can create a parent class called ShopProduct that has everything around price, name, description, category, etc...
- We can have two distinct classes, Basketball and MusicAlbum that *extend* or *inherit from* the ShopProduct class.
- There are several benefits to doing this.

```php
class ShopProduct
{
    protected $name;
    protected $description;
    protected $category;
    protected $price;

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

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

    // etc...
}
```

```php
class MusicAlbum extends ShopProduct
{
    protected $length;
    protected $artist;
    protected $producer;

    public function setLength($length) {
        $this->length = $length;
    }

    public function getLength() {
        return $this->length;
    }

    // etc...
}
```

```php
class Basketball extends ShopProduct
{
    protected $isRegulation;
    protected $brand;
    protected $maxPressure;

    public function setBrand($brand) {
        $this->brand = $brand;
    }

    public function getBrand() {
        return $this->brand;
    }

    // etc...
}
```

```php
$album = new MusicAlbum();
$album->setName('A. Bird - Mysterious...');
$album->setLength('42 minutes');
```

```php
$ball = new Basketball();
$ball->setName('NBA Basketball!');
$ball->setBrand('Nike');
```

Inheritance Benefits

- Benefit 1: Less code, all the common functionality is done *once* in the parent class.
- Benefit 2: Encapsulate code inside of classes where the code belongs and is relevant: high cohesion.
- Benefit 3: Type hinting! More on that...

Type Hinting objects

- We have 3 product related classes: ShopProduct, MusicAlbum, Basketball.
- Imagine we have a Cart object, with a add($product) method.
- The add method signature can look like:

```
public function add(ShopProduct $product) {
    //...
}
```

Type Hinting objects

- The Cart object doesn't care about what type of product, it can add any product.
- What about a class that manages Inventory.
- We may need to keep track of types of products that get sold:

```
public function sellBasketball(Basketball $product) {
    //...
}
```

More on Visibility

- If a property or method is marked as **private** a child class *does not* have access to it.
- If a property or method is marked as **protected** a child class *does* have access to it, but it's not available outside of the class (can only be accessed with $this).

More on Visibility

- When do you know whether or not something should be private, protected, or public?
- Private when the value or functionality should never be changed by anything.
- Protected when children may want to make modifications.

More on Visibility

- Think of a camera class.
- It may have a method called snapPhoto($time) which opens and closes the shutter for as long as $time specifies.
- We may want to extend this class with CompactCamera and SLRCamera.
- Should children be able to modify the snapPhoto() method?

Advanced Topics

- Calling methods belonging to a parent, using $this->method() or parent::method().
- PHP doesn't support true *method overloading*.
- *Polymorphism*: working with different objects behind a unified interface: getSummary(ShopProduct $product). Print description then length for MusicAlbum and for Basketball print whether it's regulation.

Advanced Topics

- Static methods and properties.
- Abstract Classes and Interfaces.
- Error Handling and Exceptions.
- Final classes and methods.
- Cloning objects.
- See the link for some good reading...

http://www.amazon.com/Objects-Patterns-Practice-Matt-Zandstra/dp/1590599098

Design Patterns

Design Patterns

- Problems tend to recur and we must solve them time and time again.
- How to route incoming requests, how to retrieve data from a database and model the data in code (ORM, ODM), etc...
- Eventually we develop patterns for solving these different types of problems.

Design Patterns

- Defines a problem. First step to solving a problem is to recognize the problem.
- Defines a solution. Once problem is recognized the solution is presented along with consequences of use.
- Language independent.

Design Patterns - Composition

- Eventually certain strategies break down, inheritance cannot solve every problem.
- Sometimes we must use composition over inheritance.
- Composition: remove functionality from the inheritance tree, and put it into classes outside of the inheritance tree.

Design Patterns - MVC

- First, let's recognize our problem.
- PHP code is doing a lot of things:
  - deciding what to present to the user and how
  - performing business logic (validation, authentication)
  - Persistence: database, session, file system storage
- We're mixing it all together and it is a problem.

Design Patterns - MVC

- Breaks up the application into three areas of responsibility: Model, View, Controller.
- Model: Domain objects and structures representing some knowledge.
- View: The presentational layer, visual representation of a model or models.
- Controller: The link between the user and the system, brings everything together.
- The MVC pattern gives us a clear *separation of concers*.

MVC - Model

- We need a class that knows how to get users from the database.
- This class should also be able to find individual users by their ID.
- This class should also know how to Authenticate a User.
- This class is called a Service class.

MVC - Model

- We need another class that represents a User.
- This class should just have properties of a user (username, email, first and last names).
- All business logic (authentication, validation, etc...) should be in the Service class.

MVC - Model

```php
class UserService
{
    /**
     * Return a user by their ID.
     */
    public function findById($id) {
        return $this->db->getUserById($id);
    }

    /**
     * Return an array of all users.
     */
    public function findAll() {
        return $this->db->getAllUsers();
    }

    public function authenticateUser(User $user) {
        // perform auth logic...
    }
}
```

MVC - Model

```php
class User
{
    protected $firstName;
    protected $lastName;
    protected $email;
    protected $username;

    public function getFirstName() {
        return $this->firstName;
    }

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

    public function getFullName() {
        return $this->first_name . ' ' . $this->lastName;
    }

    // etc...
}
```

MVC - View

- The view should know how to represent a user and/or a list of users.
- It should know about the User class, but NOT the UserService class.
- The view is just a mix of HTML and PHP that doesn't know how to do business logic, only display logic.

MVC - View

```
<body>
    <h1>List of Users</h1>
    <ul>
        <?php foreach($users as $user) { ?>
        <li><?php echo $user->username ?></li>
        <?php } ?>
    </ul>
</body>
```

MVC - Controller

- The controller brings everything together.
- Some routing system knows how to call the right controller class and the right method in the controller class.
- The controller should know how to retrieve a user or a list of users based on the incoming request.
- /users or /users?userId=456

MVC - Controller

```php
class UserController
{
    /**
     * The router routes /users to this method.
     */
    public function getAllUsers() {
        $service = new UserService();
        $users = $service->findAll();
        $this->view->users = $users;
    }

    /**
     * The router routes /users/{id} to this method.
     */
    public function getUser($id) {
        $service = new UserService();
        $user = $service->findById($id);
        $this->view->user = $user;
    }
}
```