COMP 2920

Server Side Web Scripting with PHP

Lesson 1

Object Oriented Programming in PHP

So far with COMP 1920, we've only explored procedural style programming. In the modern world of PHP, as of PHP 5, object-oriented programming is the preferred style of programming. It helps the programmer model real life *things* and translate them into code.

The benefits of object-oriented programming include

- ease of translating user requirements into code
- reusability of code
- modularize and isolate code
- testability of code

Before we begin, there are a few key terms we must understand.

Classes

A class can be thought of as a blueprint. It is used to build objects. Multiple objects can be created from one class. This promotes modularization and isolation of logic.

A class defines properties and methods. For example, think of a person. The properties of a person can include their age and name. Methods perform actions such as altering the properties or retrieving properties. A class should also represent one *thing*, one *responsibility*. This is known as the **single responsibility principle**.

Objects

Objects are created from classes. After a class has been defined, an instance of that class can be created. These instances are called objects. Each object has it's state defined by the properties in the class. For example, we can create multiple person objects but each person object can have a different name. Each person can alter it's state with their defined methods.

Example by Code

To best illustrate the person example above, lets dive into the code

```
<?php

class Person
{
    private $age;     // age property
    private $name;    // name property

    // a method to alter the name
    public function setName($name)
    {
        // $this refers to the current instance
        // -> refers to properties in scope
        $this->name = $name;
    }

    // a method to retrieve the name
    public function getName()
    {
        return $this->name;
    }
}
```

This is a class definition of a `Person`. Typically each class is saved into its own file, so the above should be saved into `Person.php`. By convention, classes start with an uppercase. If a class has multiple words, **PascalCase** is used – first letters of every word is a capital.

Method definitions inside the class are named using the **camelCase**. First letter of every word is capital except the very first letter remains lowercase.

So far we have only defined a class. If you attempt to run that file, you will notice nothing happens.

Lets write code to create multiple objects, alter their state, and print out their properties.

```
<?php

require("Person.php");

// create instances of the Person class
$person1 = new Person();
$person2 = new Person();

// using their methods, alter their state
$person1->setName("John");
$person2->setName("Jane");

// print out each object's name property
echo $person1->getName();
echo "<br>";
echo $person2->getName();
```

Save the above in a separate file and run it through your browser. You should see we have created 2 instances of `Person`. If we were to alter the `Person` class, both instances would share the same altered behaviour.

Constructors and Destructors

When objects are created, constructors are used and executed right off the bat. This is often used to set default values of properties or perform default actions that all objects of this class must do. To handle this, the **magic method** `__construct()` is used. This is called everytime an object is created. Likewise, when an object is destroyed, `__destruct()` is called.

For example

```
<?php

class Person
{
    private $age;     // age property
    private $name;    // name property

    public function __construct()
    {
        echo "A person was created!";
    }

    // ... rest of class definition left out

    public function __destruct()
    {
        echo "A person was destroyed!";
    }
}
```

Inheritance

Object-oriented programming in PHP also allows for inheritance. This allows classes to inherit properties and methods from other classes. This allows for the **open/close principle** – classes are open for extension but closed for modification. Through extension, you can also overwrite methods of the parent class. You may choose to overwrite the constructor to perform a new set of default actions. Or you may also call the parenting constructor as well. Take the example below, note the `extends` keyword

```
<?php

class Child extends Person
{
    public function __construct()
    {
        // call the parent constructor first
        parent::__construct();
        echo 'A child was created!';
    }

    // new method defined in the Child class
    public function sayHi()
    {
        echo "Hi";
    }
}
```

Create a new PHP file and make use of the classes above. Lets call it `driver.php` with the following code,

```
<?php

require("Person.php");
require("Child.php");

$child = new Child();
$child->setName("Johnny"); // inherit from Person
$child->sayHi();           // from Child
```

Visibility

We also have the ability to **encapsulate** and hide properties or methods, using a scope modifier. You will notice we have used `public` and `private` keywords. In this section we will discuss it in more detail.

Public

Methods and properties with the `public` scope are accessible internally and externally. Child classes can access the parent `public` methods. Using the `->` , methods and properties that are *in scope* can be accessed.

Protected

Methods and properties with the `protected` scope are only accessible within itself or child classes. In other words, a child class can access the parent `public` and `protected` methods.

Private

Methods and properties declared `private` are only accessible within itself. Child classes can not access `private` methods and properties of the parent.

Static

Methods and properties can be declared `static`. `static` allows these methods and properties to be called without instantiating a class. To call `static` types, you use the `::` operator. For example, if `$count` is a static property in the Person class, you would call it like so

```
Person::$count;
```

`static` types also belong to the class, not each object. In other words, all objects created from a class with a `static` property will share the same value for that property. This is best illustrated with a counter.

Lets first modify our `Person` class

```
<?php

class Person
{
    private $age;     // age property
    private $name;    // name property

    protected static $count = 0;  // static property

    public function __construct()
    {
        echo "A person was created!";
    }

    // ... rest of class definition left out
}
```

Now lets modify our `Child` class. Notice the scope is `protected`, so a `Child` class extending `Person` will have visibility.

```
<?php

class Child extends Person
{
    public function __construct()
    {
        // call the parent constructor first
        parent::__construct();
        echo 'A child was created!';
    }

    public function increaseCount()
    {
        // call the static $count variable from Person
        Person::$count++;
    }

    public function getCount()
    {
        return Person::$count;
    }
}
```

Lets modify our `driver.php` class to demonstrate the persistent value of `$count` across multiple `Child` objects

```
<?php

require("Person.php");
require("Child.php");

// first child object, increase the count twice
$child1 = new Child();
$child1->increaseCount();
$child1->increaseCount();

// new child object, it'll retrieve the $count value
// that was done with child1
$child2 = new Child();
echo $child2->getCount();
```

Take note that even though the increase of count was done in another object, it is shared with the second object.

If you modify the `$age`, of the objects, you will notice those values are independent.

Type Hinting

To demonstrate type hinting, lets create a new class called `Application`. The `Application` class will take in a `Person` as a parameter through the constructor. When we run the application, we will print out the `Person`'s name.

Create a file called `Application.php`

```
<?php

class Application
{
    private $person;

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

    public function run()
    {
        echo $this->person->getName();
    }
}
```

Let's create file called `index.php` to drive our application

```
<?php
require "Person.php";
require "Application.php";

$person = new Person();
$person->setName("Gary");

$app = new Application($person);
echo "Application started.";

$app->run();
echo "Application finished.";
```

This forces the application to be initialized with a `Person` as the parameter. The application will not initialize and will throw a *catchable* fatal error when the parameter is not of type `Person`. For example, if you create an `Animal` class and tried to pass that into the constructor of `Application`, PHP will throw an error.

Without type hinting, if a developer attempts to pass in an `Animal` into the constructor for `Application`, the application will continue to run and at the `run()` method, it will attempt to call the `getName()` method. Of course, because `Animal` does not have a `getName()` method, PHP will cause a fatal error.

For those paying close attention, you may be wondering, what's the difference? So what if PHP throws a *catchable* fatal error at the `__construct()` or a fatal error at `run()`? The key here is with a *catchable* fatal error, it can be handled with an error handler and the `index.php` script will continue to run. See this link on writing your own error handler

http://docs.php.net/manual/en/function.set-error-handler.php

A fatal error by calling an undefined method is not recoverable. Depending on your requirements, being able to handle an error and continue running may be desirable.

In PHP version 7, type hinting will actually throw an `exception` if the wrong type was passed. In the next lesson we will cover exception handling, among other topics.