The City Lit Institute

Department of Computing

Keeley Street, Holborn, London WC2B 4BA

OO PHP

(with Apache server)

LECTURER: **ALEXANDER ADU- SARKODIE**

MSc. Telematics (IT & Telecom), MSc. Eng., Dip. Russ. Lang., Teach. Cert, AMIAEng (UK), MBCS (UK), MIfL(UK)

Blog: http://www.blogger.com/profile/14800490193632788559

iStudio: http://www.goldhawk-college.com/istudio/

Object Oriented PHP

Object-oriented programming (OOP) is a programming paradigm that uses "**objects**" data structures consisting of **data fields** and **methods** together with their interactions- to design applications and computer programs. Programming techniques may include features such as data **inheritance**, **encapsulation**, **polymorphism, abstraction**, and **modularity**. OOP was not commonly used in mainstream software application development until the **early 1990s**

Benefits of OOP:

Inheritance

Inheritance is a fundamental capability construct in OOP where you can use **one class**, as the **base** or **basis** for **another class** or **many other classes**. Doing this allows you to **efficiently re-use the code found in your base class.**

Inheritance can make your code **lighter**, because you are **re-using the same code** in a number of different classes. This a sort of PHP **includes**.

The object-oriented development methodology places great stock in the concept of **inheritance**. This strategy promotes **code re-usability** and assumes that one will be able to use **well-designed classes within numerous applications (design pattern).**

A **design pattern** is the **ability** to have a **reusable solution** in **solving a common occurrence problem.**

'extends' is the keyword that **enables** inheritance. For example, having created the ***person*** class you could extend properties of this class to an ***employee*** class that will **inherit some properties** of the *person* class

A prototype will look like this:

```
class employee extends person {
        function __construct($employee_name) {
                $this->set_name($employee_name);
        }
}
```

where **person** is the **base** class

Encapsulation

Programmers enjoy putting little **things together as well as taking things apart**. Although self satisfying, such **in-depth knowledge of an item's inner working is normally not a requirement**. Millions of people use radio sets and computers, however not knowing the **underlying architecture** and **procedures** of how they work. Typical example is the radio. By changing a station, what you are actually doing is changing the **signal** to a **different frequency** associated with **another radio station**. Failing to understand this concept should not stop you from using the radio, because the ***interface*** takes care to hide such details.

In OOP, the practice of **separating the user from the true inner workings of an application through well-known interfaces is known as encapsulation**. The embodiment of the application is a number of **independent components**. These components are known as ***objects*** and objects **are created from a blue print or templates called *class*.**

Classes specify what **sort of data (**properties**)** the object might contain and the **behavior** (method) one would expect. This strategy offers the following advantages:

- The developer can **change** the application implementation **without affecting the object user**, because the user's only interaction with the object is via an **interface**.
- **Potential of user error is reduced** because of the **control** exercised over the user's interaction with the application.

Polymorphism

Polymorphism in OOP is the ability to **re-define** a *class 's* **characteristics or behavior** depending upon the **context** in which it is used.

For example, a ***clockIn*** behavior in an Employee definition for a Clerk and Programmer will be dependent upon the context in which the "*clockIn*" is implemented. The class Clerk could simply define *clockIn* as using a **time clock** to timestamp a card. Whiles the class Programmer will accomplish this by **signing** onto the corporate network.

Another classic example of polymorphism uses an inheritance tree rooted in the Animal class. All Animal's have a `makeNoise()` method, but the Dog class and the Cat class implement it differently. This allows you to refer to any Dog's and Cat's using an Animal reference type.

```
Animal a = new Dog();
Animal b = new Cat();
```

Now you can call `makeNoise()` on either Animal instance and know that it will make the appropriate noise. This is particularly useful if you have a Collection of Animals, and you don't know at run time exactly what type each of them really is.

Abstraction

An abstract class is a class that **cannot be instantiated**, it exists extensively for **inheritance** and it **must be inherited**. There are scenarios in which it is useful to define classes that is not intended to instantiate; because such classes **normally are used as base-classes in inheritance hierarchies, we call such classes abstract classes**.

Abstract classes cannot be used to instantiate objects; because abstract classes are **incomplete**, it may contain only definition of the properties or methods and derived classes that inherit this implements it's properties or methods. **Polymorphism is a form of abstraction.**

Generally considered, **abstraction** is a focus to **essential qualities, disregarding, ignoring or hiding details to capture some kind of commonality between different instances.**

Key OOP Concepts:

Classes

OOP revolves around a construct called a **'class'**. Classes are **blue prints or templates that are used to define objects.** They provide **basis** from which you can create **specific instances of the entity the class models**. For example, an employee management application may include an **Employee class**. You can then call upon this class to create and maintain **specific instances** or **entities**, say **Peter** and **Josh** (objects), for example.

Classes are intended to represent **those real-life items that you'd like to manipulate** within an application. PHP's generalized class creation syntax follows:

```
Class Class_Name {
        //fields declarations and properties defined here
        //method declarations defined here
}
```

The example class titled ***Employee*** below defines three fields, ***name***, ***title*** and ***wages***, in addition to two methods, ***clockIn*** and ***clockOut***. Most of these syntaxes will become more clearer when we begin to have a look at later exercises.

```
Class Employee {
        private $name;
        private $title;
        protected $wage;
        protected function clockIn(){
                echo"Member $this -> name  clocked in at"  .date("h:i:s");
        }
        protected function checkout() {
                echo"Member $this -> name  clocked out  at"  .date("h:i:s");
        }
}
```

Objects

A class provides a **basis from which you can create specific instances of the entity of the class model**, better known as *object.* Objects are created using the ***new*** keyword, like this:

$employee = **new** Employee(); // process of creating an object is called **instantiation.**

Here "Employee" is the class

Once the object is created, all of the characteristics and behaviors defined within the class are made available to the newly instantiated object (i.e. inherited)**.**

Fields

Fields are **attributes** that are intended to **describe** some aspect of a class. Unlike most languages, in PHP fields don't **necessarily need to be declared** as it is a loosely typed language. Can be generated on the fly. You can optionally **declare and assign them initial values**. An example follows:

```
Class Employee  {
        public $name = “John”; // declaring and assigning
        private $wage; //declare
}
```

In the example above the two fields **$name** and **$wage** are prefixed with a **scope descriptor** (**public** or **private**). This is a common practice when declaring fields. **Once declared, each field can be used under the terms accorded it by the scope descriptor**. (More on scope descriptors later**).**

Invoking Fields (Rendering their values**)**

Fields are referred to using the (**arrow/access**) **->** operator and, unlike variables, are **not prefaced** with a dollar sign (**Very important**). Furthermore, because field’s value typically is **specific** to a given object, it is correlated to that object like **this** (We call this **self referencing**);

```
$object -> field
```

For example, the Employee class includes the field ***name***, ***title***, and ***wage***. If you create an object named $employee of type *Employee (the **class**)*, you would refer to these fields like this:

```
$employee -> name
$employee -> title
$employee -> wage
```

They (fields) are attributes or specifics of the object

When you refer to a field from within the class in which it is defined, it is still prefaced with -> operator, furthermore you use the ***$this*** keyword to signify the **action of self referencing**. This implies that you are referring to the field residing in the same class in

which the field is being accessed or manipulated. Therefore if you were to create a method (**a function within a class**) for setting the name field in the *Employee* class, it might look like this;

```
function setName ($name){
        $this -> name=  $name
}
```

Re-cap:

- An object is created in a class by the process known as **instantiation**. When this happens, the object **inherits** all the **properties** and **methods** of the class within which it is created.
- Objects are created as **virtual entities by the PHP engine when the server runs**. To output any actions of the object you create methods in the class and use the keyword ***$this*** and the arrow operator to point to the field you are trying to access within the object. The *$this* keyword is self referencing. Also called a **pseudo-variable.**

Field Scopes (scope descriptors)
PHP supports five class fields scopes: *public, private, protected, final* and *static.*

public:
You can declare fields in the *public* scope by **prefacing** the field with the keyword *public.* An example follows;

```
class Employee {
        public $name;
        //other field and method declaration follow
}
```

Public fields can then be manipulated and accessed **directly** by a corresponding object, like so;

```
$employee = new Employee();//Instantiating an object
$employee -> name = “Peter Taylor”;//accessing direcly
$name = $employee -> name;
Echo “New employee : $name”
```

Executing this code produces the following;

New employee : Peter Taylor

private

Private fields are only **accessible** from **within the class in which they are defined**. An example follows;

```
class Employee {
        private $name;
        private $telephone;
}
```

Note: Fields designated as private are not **directly accessible by an instantiated object**, nor are they available as **sub-classes**. If you want to make the fields available then it is best to consider using the ***protected*** scope instead.

Private Fields must be accessed via **publicly exposed interfaces**. This practice of **encapsulation** allows separating the user from the true inner workings of the application.

Below is an example of a class, in which a private field is manipulated by a **public method.**

```
class Employee {
        private $name;
//accessed via a public exposed interface (encapsualtion)
        public function setName ($name){
                $this -> name =$name;
        }
}
$staff = new Employee(); //process of instantiation
$staff -> setName("John Smith"); //pass value
```

Note

Encapsulating the management of such fields within a method enables you to **maintain tight control over how that field is set**. For example, you could add to the setName method's capabilities to **validate** that the name is set to **solely alphabetical characters and further ensure that it is not blank**. This strategy is more reliable for future proofing your application than leaving it to the end user to provide valid information.

protected

They often require variables intended **for use only within the function**. Classes can include fields used for solely internal purposes.

An example follows

```
class Employee {
        protected $wage;
}
```

Final:

Marking a field as final prevents it from being **overridden by a subclass**. A *finalized* field is declared as follows;

```
class Employee {
        final $isbn;
}
```

You can also declare methods as final.

Summary:

- In OOP, you think of software as being a series of **virtual objects**. A **series** of **mini programs inside a larger one**. The larger program is the **entire script** for the application that you are writing. This could be a **blog** or **message board**. A list of mini programs within the application could be;
  - ⇨ An object to **connect to a database** for the application
  - ⇨ An object to **handle validation**
  - ⇨ An object for handling **user input**
  - ⇨ An object to **build a table for the result of a query** from a database

    All these help to give **you better organization and management of your code base**
- OOP allows you to create **re-usable block/components of code** if designed properly.
- Easy to **update** OOP system as they are **modular based**. OOP systems are designed to be **plug and play**.
- Since the whole application is **segmented**, you can adopt **division** of labor. Developers can work co-currently on different **models/components** to build objects for the application **without interfering with one another**. In short, it is easier to have multiple programmers working on code base and not worrying about touching or breaking code in other parts of the system.

- You can **eliminate** a distinctive functional component from the system and should not affect other functional components of the system.
- You can use the components to **build a library**. Most of the components of the library you build can be imported into other projects.
- OOP principles are consistent in other languages, so it is **easy to move from one OOP language say PHP** to say Ruby or Java. Though syntaxes may vary and are quite minimal. The same is for frameworks like Zend, Pear and Smarty templates.

It is worth knowing the few disadvantages of using OOP as against traditional **event driven procedural programming**:

- It is **verbose**. More code needed to write to get job done. If your project requires say one to three pages, then you will be better of using the old procedural event driven way using **functions** which are immutable.

- The application's speed is compromised. When you have OOP code there are a number of things the interpreter has to deal with. OOP code is **slower at run time**. If this is a problem you can put **more processors** behind the code and have in place a strategy for **load balancing via analytic tests.**

  Using a **RESTful** (Representional State Transfer) web services will allow a topology of cloud connected servers (cloud computing) to handle data transfer managing load balancing seamlessly. This behaviour is known as **stateless**. RESTful services renders explicit HTTP requests, by exposing the application to a structure-like URI which is hierarchical e.g. An example of a URI discussion on a blog will have the URI http://codeunique/discussions/discusion/{topic} . They can use both XML and JSON data formats, and support CRUDE.

Inheritance

As applied to PHP, class inheritance is accomplished by using the **extends** keyword.

A class that inherits from another class is known as **child** class, or a **sub-class**. The class from which the child class inherits is known as the **parent**, or **base** class.

Example 1

```
<?php
	//Define a base Employee class
	class Employee {
		private $name;
		//Define a setter for the private $name member.
		function setName($name){
			if ($name == "") echo "Name cannot be blank!";
			else $this -> name =  $name;
		}
		//Define a getter for the private $name member to return the value
		function getName() {
			return "My name is ".$this -> name." <br />";
		}
	}//End of Employee class
```

```
//Now define an Executive class that inherits   from Employee
class Executive extends Employee {
//Define a method unique to Executive
        function pillageCompany() {
                echo "I am selling the company assets to finance my yatch";
        }
}//End Executive class
//Create a new executive object
$exec = new Executive();
//Call the setName() method defined in the Employee class
$exec -> setName ("Alexander");
//Call the Employee getName () method
echo $exec -> getName ();
//Now call the pillageCompany method for association
$exec -> pillageCompany ();
?>.
```

Example 2

```
<?php
//Define a base Employee class
	class Employee {
		private $name;
		//Define a setter for the private $name member.
		function setName ($name){
			if ($name == "") {
				echo "Name cannot be blank!";
			}
			else $this -> name =  $name;
		}
		//Define a getter for the private $name member to return the value
		function getName () {
			return "My name is ".$this -> name." <br />";
		}
	}//End of Employee class
	//1 Now define an Executive class that inherits from Employee
	class Executive extends Employee {
		//Define a method unique to Employee
			function pillageCompany () {
				echo "I am selling  company assets to finance my yatch.";
			}
```

```
}//End Executive class


//Create a new executive object
$exec = new Executive();
//Call the setName () method defined in the Employee class
$exec -> setName ("Alexander");
//Call the Employee getName () method on the $exec object
echo $exec -> getName();
//2. Extending Executive: class CEO extends Executive
class CEO extends Executive {
        function getFaceLift () {
                echo " I need some square jaws";
        }
}



$ceo = new CEO();
        $ceo  -> setName ("Marvin");
        //Now call the pillageCompany method for association
        $ceo  -> pillageCompany();
        //Now call the getFaceLift in-built method for ceo
        $ceo  ->getFaceLift ();
?>
```

Constructors and Destructors

Every class has two special functions, **constructor** and **destructor**. Even if you do not **explicitly** declare and define them, they exist (they are **empty**).

A **constructor** is a **special function of a class** that is **automatically executed** whenever an object of a class **gets instantiated**.

It is called right after you create a new object. This makes constructor suitable for any object **initialization** you need.

Conversely, a **destructor** is a function which is called right after you **release** an object. Releasing object means that you **do not need it or use it anymore**. This makes destructor suitable for any **final actions** you want to perform.

In PHP5 constructor is a function called __**construct**.

Destructor is a function called __**destruct**(). Unlike a constructor, this function cannot be called **directly**. It will be called **implicitly** when you release your object.

When you create **inherited class**, its constructor will be executed. It may be logical to you that the constructor of a parent class executes too, but this is not a case. **If you want to execute parent constructor you have to call it explicitly in a subclass constructor with**:

parent::__construct

classname::__construct()

.

Why do we need a Constructor?

It is needed as it provides an opportunity for doing **necessary setup operations** like **initializing class variables**, **opening database connections** or **socket connections**, etc. In simple terms, it is **needed** to **setup the object and its environment before it can be used**.

Example 3

```
class Customer {
        private $first_name;
        private $last_name;
        private $outstanding_amount;
        public function __construct() {
                $first_name = "";
                $last_name = "";
                $outstanding_amount = 0;
        }

        public function setData($first_name, $last_name, $outstanding_amount) {
                $this->first_name = $first_name;
                $this->last_name = $last_name;
                $this->outstanding_amount = $outstanding_amount;
        }
        public function printData() {
                echo "Name : " . $first_name . " " . $this->last_name . "\n";
                echo "Outstanding Amount : " . $this->outstanding_amount . "\n";
        }

}
$c1 = new Customer();
$c1->setData("Sunil","Bhatia",0);
```

In the above example, we create a **new object** of the **Customer class**. The 'new' operator is responsible for creating the Customer object. The process of creating the new instance (an) object in called **instantiation**.

At this point PHP **searches** the Customer class to see if a **constructor** has been defined. Therefore it calls the constructor method i.e __construct() defined earlier. The __construct() method sets the $firstname and $lastname to blank and sets the $outstanding_amount to zero. It can also be used for validation.

Also, once the process of instantiation has been complete, then you can use the **method(s) inside the class to set a property or pass a value to the instantiated object**.

```
$c1 = new Customer();


$c1->setData("Sunil","Bhatia",0);
```

Parent Constructors

Parent constructors are not called implicitly. They are called if the child class defines a constructor. In order to run a parent constructor, a call to **parent::__construct()** within the child constructor is required.

Invoking Parent Constructors

PHP does not automatically call the parent constructor; you must call it explicitly using the **parent** keyword. Note**::** which is part of the notation.

Example 4 using new unified constructors

```
<?php
class BaseClass {
   function __construct() {
       print "In BaseClass constructor\n";
   }
}

class SubClass extends BaseClass {
   function __construct() {
       parent::__construct();
       print "In SubClass constructor\n";
   }
}

$obj = new BaseClass();
$obj = new SubClass();
?>
```

Example 5

```
<?php
	class Employee {
		protected $name;
		protected $title;
		function __construct() {
			echo  “<p>Staff constructor called!</p>”;
		}
	}

	class Manager extends Employee {
		function   __construct() {
			parent::__construct();
			echo  “<p>Manager constructor called!</p>”;
		}
	}
?>
```

This results in the following:

Staff constructor called!

Manager constructor called

Parameterized Constructor or Argument Constructor

A parameterized or argument constructor is a constructor which accepts **values** in the form of **arguments** in the constructor. Unlike other programming languages where overloaded arguments constructors are possible, in PHP5 you cannot overload constructors.

Example 6

```
class Customer {
        private $first_name;
        private $last_name;
        private $outstanding_amount;
        public function __construct($first_name, $last_name, $outstanding_amount) {
                $this->setData($first_name, $last_name, $outstanding_amount);
               $this->printData();
        }
        public function setData($first_name, $last_name, $outstanding_amount) {
                $this->first_name = $first_name;
                $this->last_name = $last_name;
                $this->outstanding_amount = $outstanding_amount;
        }
        public function printData() {
                echo "Name : " .$this-> first_name . " " . $this->last_name . "\n";
                echo "Outstanding Amount : " . $this->outstanding_amount . "\n";
        }
}
$c1 = new Customer("Sunil","Bhatia",0);
```

In the above example, we create a new object $c1 and pass the values “Sunil”, “Bhatia” and zero to the constructor defined. The constructor now takes three arguments and stores them in the internal private variable $first_name, $last_name and $outstanding_amount respectively.

Example 7

As an example, suppose you want to immediately populate certain book fields with information specific to a supplied ISBN. For example, you might want to know the title and author of a book, in addition to how many copies the library owns and how many are presently available for loan. The code for this task will look something like this:

```
<?php
        class Book {
                private $title;
                private $isbn;
                private $copies;
                public function __construct($isbn) {
                        $this ->setIsbn($isbn);
                        $this ->getTitle();
                        $this->getNumberCopies();
                }
                public function setIsbn($isbn) {
                        $this ->isbn = $isbn;
                }
                public function getTitle() {
                        $this ->title = “Beginning Python”;
                        print  “Title: ” .$this->title.<”br />”;
                }
```

```
        public function getNumberCopies() {
            $this ->copies = “5”;
            print  “Number copies available: ” .$this->copies.<”br />”;
        }
    }
    $book= new Book(“159055199x”)
?>
```

This results in the following:

Title : Beginning Python

Number copies available: 5

Destructor

The destructor method will be called as soon as **all references** to a particular object are **removed** or when the object is **explicitly destroyed** or in any order in **shutdown sequence.** Destructors are created like any other method but must be titled **__destruct()**

Example 8

```
<?php
class MyDestructableClass {
   function __construct() {
      print "In constructor\n";
      $this->name = "MyDestructableClass";//this will be destroyed on using __destruct function
   }

   function __destruct() {
      print "Destroying " . $this->name . "\n";
   }
}

$obj = new MyDestructableClass();
?>
```

Example 9

```
<?php

        class Boook {

                private $title;

                private $isbn;

                private $copies;

                function __construct($isbn) {

                        echo “<p>Book instance class created.</p>”;

                }

                function __destruct($isbn) {

                        echo “<p>Book instance class destroyed.</p>”;

                }

                $book= new Book(“159055199x”);

        }

?>
```

The result will be;

Book class instance created.

Book class instance destroyed

When the script is complete, PHP will **destroy** any objects that reside in memory.

Note, therefore that, if the instantiated class and information created as a result of the instantiation **reside in memory**, you’re not required to explicitly declare a destructor. However, if less volatile data is created (say, stored in a database) as a result of the instantiation and should be destroyed at the time of the object destruction, you’ll need to create a custom destructor.

Static Class Members

Sometimes it is useful to create **fields** and **methods** that are **not invoked by any particular object** but rather are **global** and are **shared** by all class instances.

Declaring class members or methods as static **makes them accessible without needing an instantiation of the class**. A **member** declared as static **cannot be accessed** with an instantiated class object (**though a static method can**).

The big difference between php 4 and php 5 is that a method declared as "**static**" does not have $**this** set (which is for self referencing). You'll get a fatal error, in fact, if you try to use $this in a static method.

For example, suppose you are writing a class that **tracks the number of Web page visitors**. What you wouldn't want to do, is to want the **visitor count reset to zero every time the class is instantiated (**a user signs in**)**, and therefore you would set the field to be of **static** scope. Static scope are **recursive** in nature. Each time the execution takes place.

Example 10

```
<?php
        class Visitor {
                private static $visitors = 0;
                function  __construct() {
                        self::$visitors++
                }
                static function getVisitors() {
                        return self::$visitors
                }
                /*Now we instantiate  2  Visitor classes  */
                $visits = new Visitor;
                        echo Visitor::getVisitors . “<br />”;//class name::method name
                $visits2 = new Visitor;
                        echo Visitor::getVisitors . “<br />”;
        }
        ?>
```

Because the $visitors field was declared as **static**, any changes made to its value **are reflected across all instantiated objects**. Also note that static fields and methods are referred to using **self** keyword and **class** name, rather than via $this and arrow operators. This is not possible with static fields. Any attempts to do that will result in a syntax error. This is because static fields are re-cursive. They call themselves repeatedly.

Magic methods

A quicker technique for accessing array elements

Exercise 11

```
<?php


class Person11 {
    private $fields = array( a=>'50 Paul',
                             b=>'3 James',
                             c=>'15 Gibbs',
                             d=>'5 Arthur',
                             e=>'4 Steve');


    public function __get($k) {
        if(isset($this->fields[$k])) {
        //factorise to use for each. Use argument as an array reference
            return $this->fields[$k];
        }
        return null;
    }
}
$p = new Person11();
echo $p->a ."<br />".$p->b ."<br />".$p->c."<br />".$p->d."<br />".$p->e;
?>
```

Ecommerce

- Widgets
- Transactions

MVC Framework

- Model
- View
- Controller