OBJECT ORIENTED PROGRAMMING IN PHP

By
Agubata Odinaka

What you will learn:

- Why you should consider switching to OOP.
- Basics of OOP in php.
- How to work with MYSQLi (object oriented) and PDO.

Content:

Why Learn OOP?

For people new to OOP and are comfortable with 'classic' procedural php, you may be wondering why you should even bother to switch to object oriented programming… why go through the trouble?

The Pros of Learning PHP OOP

- PHP is moving in an OOP direction. For example, many important PHP extensions like PEAR and Smarty are OO based.
- OOP is more suited for complex group projects where so many programmers are involved in one project.
- OOP is the modern way of software development and all the major languages (Java,PERL, PHP, C#, Ruby) use this method of programming.
- It makes your code more maintainable. Instead of dividing your program into **tasks** (functions), you divide it into **objects**.

Consider this…

Procedural

```
function strength($kind)
{
  switch($kind){
   case 'man':
     manKind();
     break;
   case 'woman':
     womankind();
     break;
   }
}
```

OOP

```
function checkStrength($kind)
{
  $kind->strength();
}
```

But…

If we do not have many functions performing similar actions in our project, OOP may not really be necessary.

Basic OOP Principles

Understanding Some OOP Concepts

First of all, there's confusion in OOP: seasoned developers talk about objects and classes like they are interchangeable terms. This is not the case however, though the difference can be tough to wrap your head around at first.

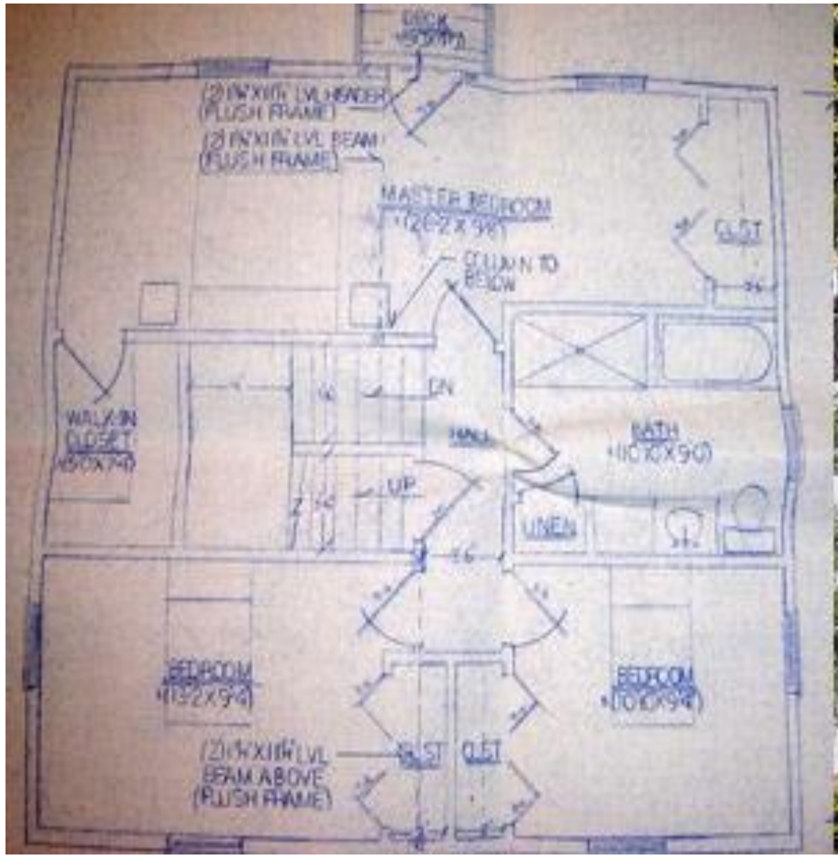
A class is like a blueprint for a house. It defines the shape of the house on paper, with relationships between the different parts of the house clearly defined and planned out, even though the house doesn't exist.

An object, then, is like the actual house built according to that blueprint.

Take a look at this…

Class

Object



Creating a class

You define your own class in php by starting with the keyword 'class' followed by the name you want to give your new class. The naming convention for classes is a capital first letter and a camel case. For example:

```
class MyClass{
    …
}
```

Creating/Instantiating an object

Like we have said earlier, an object is created or instantiated from a class. An object is instantiated with the keyword ‘new’, followed by the name of the class. For example:

```
$obj=new MyClass();
```

OR

```
$obj=new MyClass;
```

Adding properties to your class

A class contains both data (variables) and functions that form a package called an 'object'. When you create a variable inside a class, it is called a 'property'. For example, lets give our class a property called name:

```
class MyClass{
    var name;
}
```

Adding methods to your class

In the same way that variables get a different name when created inside a class, functions are also referred to by a different name when created inside a class - they are called 'methods'. A classes' methods are used to manipulate its' own properties.

The naming convention for methods is a small first letter and a camel case or an underscore in-between the joined words like getName or ‘get_name’. For example:

```
class MyClass{
    var name;
    function set_name($input){
        …
    }
    function get_name(){
        …
    }
}
```

Constants…

A constant is somewhat like a variable, in that it holds a value, but it is *immutable*. Once you declare a constant, it does not change.

```
class MyClass {
    const MAIDEN_NAME = “Ikpeama”;
    …
}
```

In the class above, MAIDEN_NAME is a constant. It is declared with the keyword const, and under no circumstances can the value be changed to anything other than “Ikpeama”. Note that the constant’s name does not have a leading $, as variable names do and it is in uppercase.

Abstract classes…

An abstract class is one that cannot be instantiated, only inherited.

You declare an abstract class with the keyword abstract, like this:

```
abstract class MyClass {
    abstract function myFunction(){
        …
    }
}
```

Note that for you to make a class an abstract class, it must contain at least one abstract method. You cannot have an abstract method inside a non-abstract class.

The constructor method…

A constructor is a special built-in method that allows you to initialize your object's properties (give your properties values) when you instantiate (create) an object. This is optional and is solely dependent on choice. The 'construct' method starts with two underscores (__) and the word 'construct'.

```
class MyClass {
    var $name;
    function __construct($input){
        …
    }
}
```

The $this variable…

The $this is a built-in variable (built into all objects) which points to the current object. In other words, $this is a special self-referencing variable. You use $this and the '->' operator to access properties and to call other methods of the current class.

```
class MyClass {

    var $name;

   function __construct($input){

        $this->name=$input;

   }

}
```

Note that there is no '$' before the variable name in '$this->name'.

Calling a method in a class…

In OOP, a scenario may arise in a class where one may want to call either a method of the same class or that of another class. To call a method of another class, we do this…

```
class FirstClass {
    var $name;
    function set_name($input){
        $this->name=$input;
    }
}
```

```
class SecondClass {
    function __construct($name){
        FirstClass::set_name($name);
    }
}
```

To call a method of a class in another method in the same class, we do this…

```
class MyClass{
    const NAME="Onyema";
    function get_name(){
        echo self::NAME;
    }
    function check_name(){
        self::get_name();
    }
}
```

The keyword “static”…

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

```
class FirstClass {
     public static $name="Makata";
    function get_name(){
         return self::$name;
    }
}
echo FirstClass::$name;
$someone=new FirstClass();
echo $someone->get_name();
```

The keyword “final”…

The ‘final’ keyword which prevents child classes from overriding a method by prefixing the definition with final. If the class itself is being defined final then it cannot be extended.

The following example results in Fatal error: Cannot override final method BaseClass::test()…

```
class BaseClass {
    final public function test(){
        echo “BaseClass”;
    }
}
```

```
class ChildClass extends BaseClass{
    public function test(){
        echo “overridden”;
    }
}
$object=new ChildClass();
$object->test();
```

Creating a Class

Before delving into the principles of OOP and to help us become more familiar with some concepts in OOP, we shall…

- ✓ create a class with properties and methods.
- ✓ instantiate an object and call a method of the class.

```
class Register
{
    var $name;
    var $age;

    function student_name()
    {
        return $this->name;
    }
    function student_age()
    {
        return $this->age;
    }

    function nickName()
    {
        $nick=substr ($this->name,6);
        return $nick;
    }
}
```

Explaining what we have done…

- We created a class called “Register”.
- The class has two properties, “name” and “age”.
- It also has three methods “student_name()”,”student_age()”, and “nickName()”.

Note: When accessing methods and properties of a class, you use the arrow (->) operator.

The next step is to add a constructor to the class.

```
function __construct($name,$age){
        $this->name=$name;
        $this->age=$age;
  }
```

On completion, our “register” class will look like this

```
class Register{
    var $name;
    var $age;
    function __construct($name,$age){
        $this->name=$name;
        $this->age=$age;
    }
    function student_name(){
        return $this->name;
    }
    function student_age(){
        return $this->age;
    }
    function nickName(){
        $nick=substr ($this->name,6);
        return $nick;
    }
}
```

Now, we instantiate an object.

You 'feed' the constructor method by providing a list of arguments (like a function)after the class name.

```
$student=new Register(“SplashActivity”,”40”);
```

Then, we call a method in the object.

```
echo $student->nickName();
```

Principles of OOP

Inheritance- this is a principle in oop whereby, a new class takes all the properties of an already existing class(commonly referred to as a base class or a super class). The keyword “extends” is used to initiate inheritance.

```
Class Welcome extends Register{
    …
}
```

Polymorphism-this is a situation whereby the same method performs different functions in different objects created from the same class.

```
Class Human{
    function talk(){
        echo “speech”;
    }
}
```

```
Class Englishman extends Human{
    function talk(){
        echo “English”;
    }
}
Class Igboman extends Human{
    function talk(){
        echo “Igbo”;
    }
}
$man=new Englishman; $man->talk();
$boy=new Igboman;$boy->talk();
```

Encapsulation- One of the fundamental principles in OOP is 'encapsulation'. The idea is that you create a cleaner and better code if you restrict access to the data structures (properties and methods) in your objects.

There are 3 access modifiers:

- • public
- • private
- • protected

Public is the default modifier.

- Properties or methods declared as 'public' have no access restrictions, meaning anyone can access them. Every constructor method is automatically a public method.
- When a property or method in a class is declared 'protected', only the class and classes derived from that class can access it - this has to do with inheritance.
- When you declare a property or method in a class as 'private', only the same class can access it.

The power of OOP comes mainly from inheritance and polymorphism. If you use classes, but never use either of those two concepts, you probably don't need to be using a class in the first place.

```
Class Register{
      private $accountNum;
     function __construct($input){
          self::set_account($input);
     }
     private function set_account($num){
           $this->accountNum=$num;
     }
     public function get_account(){
          echo $this->accountNum;
     }
     protected function checkAccnt(){
          echo $this->accountNum;
     }
}
```

Trying to access the methods…

```
$acc=new register("234");
$acc->get_account();
$acc->checkAccnt();
```

The Autoload Function

When writing object-oriented code, it is often customary to put each class in its own source file. The advantage of this is that it’s much easier to find where a class is placed, and it also minimizes the amount of code that needs to be included because you only include exactly the classes you need.

The downside is that you often have to include tons and tons of source files, which can be a pain, often leading to including too many files and a code-maintenance headache.

__autoload() solves this problem by not requiring you to include classes you are about to use.

If an __autoload() function is defined (only one such function can exist per application) and you access a class that hasn't been defined on that page, it will be called with the class name as a parameter. This gives you a chance to include the class just in time.

If you successfully include the class, your source code continues executing as if the class had been defined. If the class doesn't exist, the scripting engine raises a fatal error.

Here’s a typical example using __autoload():

Create a php file and name it MyClass.php:

```
<?php
    class MyClass {
        function checkAuto(){
            echo “Cauchy is great";
        }
    }
?>
```

Create another php file and name it LinkClass.php:

```
<?php
    function __autoload($class_name)
    {
        require_once “$class_name.php";
    }
?>
```

Then, coming to your main page:

```
<?php
     require_once “LinkClass.php";
    $obj = new MyClass();
    $obj->checkAuto();
?>
```

Notice that MyClass.php was not explicitly included here but implicitly by the call to __autoload(). You will usually keep the definition of __autoload() in a file that is included by all of your main script files and when the amount of classes you use increases, the savings in code and maintenance will be great.

OOP and Database

There are two ways we can relate with our database(which is mysql in our case) via php oop. They are…

- **MYSQLi** - simply ‘improved’ MYSQL. The MySQLi Extension (MySQL Improved) is a relational database driver used in the PHP programming language to provide an interface with MySQL databases.

- **PDO** - the acronym for PHP Data Objects. The PHP Data Objects (PDO) extension defines a lightweight, consistent interface for accessing databases in PHP.

Make your choice…

MYSQLi supports both procedural and object oriented php but PDO is purely object oriented.

The major advantage of PDO is that it presently supports around 18 different databases which makes for flexibility and easy migration(when necessary).

The choice of which to use is based purely on the project in question and also based on choice but here, we shall just use the two for the purpose of learning.

Connecting to database and creating table…

- MYSQLi OOP

```
// Create connection

$connection=new MySQLi($servername,$username,$password,$database)

$myQuery = "CREATE TABLE MyTable (id INT AUTO_INCREMENT PRIMARY KEY,
surname VARCHAR(255) NOT NULL, picture VARCHAR(255) NOT NULL)”;

if ($connection->query($myQuery)) {
    echo "Table created";
} else {
    echo $connection->error;
}
$connection->close();
```

❑ PDO

```
try {
$connection = new PDO("mysql:host=$servername;dbname=$database", $username, $password);
  $connection->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
  $myQuery = "CREATE TABLE MyTable (id INT AUTO_INCREMENT PRIMARY KEY,
    surname VARCHAR(255) NOT NULL, picture VARCHAR(255)NOT NULL)";
  $connection->exec($myQuery);
  echo "Table created";

}
catch(PDOException $e){
    echo $e->getMessage();
    }
$connection= null;

?>
```

Inserting data into the database…

- MYSQLi OOP

```
$query="INSERT INTO MyTable(name, picture) VALUES(?,?)";

$statement=$connection->prepare($query);

$name="Ogbuefi; $picture="great";

$statement->bind_param("ss",$name,$picture);

$statement->execute();

echo "inserted";

$connection->close();
```

❑ PDO

```
try{
	$connection->setAttribute(PDO::ATTR_ERRMODE,PDO::ERRMODE_EXCEPTION);
	$query="INSERT INTO MyTable(name, picture)  VALUES(:name, :picture)";
	$statement=$connection->prepare($query);
	$array=array(':name'=>$name, ’:picture’=>$picture);
	$statement->execute($array);
	echo “inserted";
}catch(PDOException $e){
	echo $e->getMessage();
}
$connection=null;
```

Selecting data from database(with conditions)…

- MYSQLi OOP

```
$query="SELECT name FROM MyTable WHERE id=?";
$statement=$connection->prepare($query);
$statement->bind_param("i",$id);
$statement->execute();
$statement->bind_result($name);
while($statement->fetch()){
    echo "name=".$name<br>";
}
$statement->close();
$connection->close();
```

- PDO

```
$connection=

new PDO("mysql:host=$server;dbname=$database",$username,$password,

array(PDO::ATTR_EMULATE_PREPARES => false, PDO::ATTR_ERRMODE =>
PDO::ERRMODE_EXCEPTION));

try{

    $query="SELECT * FROM linktable WHERE id=:id";

    $statement=$connection->prepare($query);

    $id=“1";

    $statement->bindParam(':id',$id);

    $statement->execute();
```

PDO

```
    $result=$statement->fetchAll();
     foreach($result as $row){
         echo $row[‘id'].“<br>";
    }
}catch(PDOException $e){
    echo $e->getMessage();
}
$connection=null
```

Selecting data from database(without conditions)…

- MYSQLi OOP

```
$query=“SELECT * FROM MyTable";

$statement=$connection->query($query);

While($row=$statement->fetch_assoc()){

      echo “name=”.$row[“name”]

}
```

- PDO

```
$query="SELECT * FROM MyTable";

$statement=$connection->query($query);

//You can either use

while($fetch=$statement->fetch()){

      echo "name=$fetch["name"]<br>";

}

//OR

Foreach($statement->fetchAll() as $records){

      echo $records["name"];

}
```

Prepared Statements

You must have noticed this line of code

```
$statement=$connection->prepare($query);
```

Whenever you interact with a database using user input, you should **ALWAYS** use prepared statements.

A prepared statement is a feature used to execute the same (or similar) SQL statements repeatedly with high efficiency. Prepared statements are very useful against SQL injections.

Just think of it as a magic method(just like your “clean-up code” in Android Studio).

How does it work?

Prepared statements basically work like this:

- Prepare: An SQL statement template is created and sent to the database. Certain values are left unspecified, called parameters (labeled "?“ or “:name”).
- The database parses, compiles, and performs query optimization on the SQL statement template, and stores the result without executing it.
- Execute: At a later time, the application binds the values to the parameters, and the database executes the statement. The application may execute the statement as many times as it wants with different values

Why choose it over direct SQL statement execution?

- Prepared statements reduces parsing time as the preparation on the query is done only once (although the statement is executed multiple times).
- Bound parameters minimize bandwidth to the server as you need to send only the parameters each time, and not the whole query.
- Prepared statements are very useful against SQL injections, because parameter values, which are transmitted later using a different protocol, need not be correctly escaped. If the original statement template is not derived from external input, SQL injection cannot occur.

Code Example…

We are going to write a code to upload picture, create a thumbnail from the picture and display the thumbnail on our browser with php OOP.

What do we need…

To do this, we are going to break our code down into objects…think about the objects we may possibly need to get this done…

- We are definitely going to need an object to insert data into our database.
- We are also going to need an object to read the image.
- We are going to need an object that actually creates the thumbnail

NOTE: the number of objects you are going to need depends on the method you will employ to achieve the objective. Different people may require different objects to achieve the same thing.

First, we create the interface…

Create a php file and call it interface.php.

Type the following inside the body tag…

```
<form action="handle.php" method="post"
enctype="multipart/form-data">
<input name="surname" placeholder="surname" type="text"/>
<input name="othernames" placeholder="othernames" type="text"/>
<input type="file" name="pic" />
<button type="submit" name="btn-upload">upload</button>
</form>
```

Then, we create the classes we require…

Create another php file and call it classes.php.

Here, we shall create the classes we need. Then we shall include this file in all other files we shall create so we can easily make use of the classes.

Attempt to create a class to insert the name, surname and picture path into the database.

Clue…

Create a class, add a constructor to it and paste our PDO insert code inside the constructor.

```
class DatabaseInsert{
    const SERVER="localhost";
    const USERNAME="root";
    const PASSWORD="";
    const DATABASE="link";
    function __construct($surname,$othernames,$picture){
try{
    $connection=
new
PDO("mysql:host=".self::SERVER.";dbname=".self::DATABASE,self::US
ERNAME,self::PASSWORD,
array(PDO::ATTR_EMULATE_PREPARES => false,
PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
```

…

```
$query="INSERT INTO linktable(surname, othernames, picture)
VALUES(:surname,:othernames,:picture)";
    $statement=$connection->prepare($query);
    $array=array(':surname'=>$surname,':othernames'=>$othername
s,':picture'=>$picture);
    $statement->execute($array);
}catch(PDOException $e){
    echo $e->getMessage();
}
    $connection=null;
    }
```

Then we create our image class…

```
class image{
    protected $image;
    protected $width;
    protected $height;
    private $mimetype;
    function __construct($filename){
            $this-
>image=imagecreatefromstring(file_get_contents($filename));
$info = getimagesize($filename);
```

```
      $this->width = $info[0];
       $this->height = $info[1];
       $this->mimetype = $info['mime'];
      }
      public function display() {
       header("Content-type: {$this->mimetype}");
       switch($this->mimetype) {
          case 'image/jpeg': imagejpeg($this->image); break;
          case 'image/png': imagepng($this->image); break;
          case 'image/gif': imagegif($this->image); break;
       }
    }
}
```

Then we create our thumbnail class…

```
class Thumbnail extends image{

     function __construct($image, $th_height) {

      // call the super-class contructor

      parent::__construct($image);

          $th_width = ($th_height*$this->width)/$this->height;

      // modify the image to create a thumbnail

      $thumb = imagecreatetruecolor($th_width, $th_height);

      imagecopyresampled($thumb, $this->image, 0, 0, 0, 0, $th_width,
$th_height, $this->width, $this->height);

      $this->image = $thumb;

   }

}
```

We now create a file to process the form…

Let us call this file handle.php

```
require_once(“LinkClasses.php");

     if(isset($_POST['btn-upload']))

{

        $surname=$_POST['surname'];

          $othernames=$_POST['othernames'];

      $pic = rand(1000,100000)."-".$_FILES['pic']['name'];

      $pic_loc = $_FILES['pic']['tmp_name'];

      $folder="picture/";

     $picture=$folder.$pic;
```

```
 if(move_uploaded_file($pic_loc,$picture))
        {
            $insert=new DatabaseInsert($surname,$othernames,$picture);
            $image=new Thumbnail ($picture,100);
            $image->display();
        }
}
```

Notice that we included the file with all our classes here with the__autoload() and we created and used the objects we needed with ease.

Run the page on your browser and test your work…

Thanks…

Questions