

# LAB 10

---

## PHP CLASSES AND OBJECTS

### What You Will Learn

- To create and use your own classes in php
- Some design principals to help your write good code
- Some standard Object oriented design patterns

### Approximate Time

The exercises in this lab should take approximately 90 minutes to complete.

## Fundamentals of Web Development

Randy Connolly and Ricardo Hoar

Textbook by Pearson  
<http://www.funwebdev.com>

### PREPARING DIRECTORIES

- 1 If you haven't done so already, create a folder in your personal drive for all the labs for this book. Like Labs 8 and 9, this set of labs requires a functioning webserver to interpret the PHP code. If you do not have a webserver running go back to Lab 8 exercises and get one set up.
- 2 From the main labs folder (either downloaded from the textbook's web site using the code provided with the textbook or in a common location provided by your instructor), copy the folder titled lab10 to your course folder created in step one.

Objects and classes are a technique used by software developers to improve code readability, increase code reuse and modularity, and support the design of solutions for any domain.

## CLASSES AND OBJECTS IN PHP

### EXERCISE 10.1 — DEFINE A CLASS

- 1 To make your first class easy enough to complete, but meaningful enough to illustrate why classes are good, consider the weather examples you did for Lab 9. We used parallel arrays of values to store information, and had to put all functionality where it was used.
- 2 You are given a file [lab10-exercise01.php](#). In that file there is a reference to a *Forecast* class, and a file named `forecast.class.php`. As provided the file will generate errors, which this exercise will fix.
- 3 Create a file named [Forecast.class.php](#). Inside that file create a new class names *Forecast* that will contain a date, high temperature, low temperature, and a text description:

```
class Forecast{
    public $date;
    public $high;
    public $low;
    public $description;

    function __construct($d, $h, $l, $desc){
        $this->date = $d;
        $this->high = $h;
        $this->low = $l;
        $this->description = $desc;
    }
}
```

- 4 Now add a `toString()` method to your class that makes use of the internal class variables to format and return nice output forecast like in Lab 9. As follows:

```
public function __toString(){
    return '<div class="panel panel-default col-lg-2 col-md-4 col-sm-6">
        <div class="panel-heading">
            <h3 class="panel-title">'. $this->date. '</h3>
        </div>
        <div class="panel-body">
            <table class="table table-hover"><tbody>
                <tr><td>High:</td><td>'. $this->high. '</td></tr>
                <tr><td>Low:</td><td>'. $this->low. '</td></tr>
            </tbody>
        </table>
        </div>
        <div class="panel-footer">'. $this->description. '</div>
    </div>';
}
```

When you refresh [lab10-exercise01.php](#) you will see that it outputs a Forecast using the Forecast Class we just defined to create output like that shown in Figure 10.1.

### Weather forecast using classes

24 Feb, 2014	25 Feb, 2014	26 Feb, 2014	27 Feb, 2014	28 Feb, 2014	01 Mar, 2014
High: 30	High: 35	High: 20	High: 25	High: 30	High: 40
Low: 20	Low: 22	Low: 10	Low: 14	Low: 24	Low: 26
sunny	clear and sunny	cold and snow	chilly	sunny	warming

  

02 Mar, 2014
High: 50
Low: 34
hot

Figure 10.1 Screenshot of the output from Exercise 1

## EXERCISE 10.2 — INSTITUTE OBJECTS

- 1 In the previous exercise you wrote a constructor, but the use of the constructors was done for you. This exercise will walk you through calling your own constructors. Continue working on the previous Exercise for this Exercise.
- 2 Find the code that defines that array for Forecast objects and remove it. Now we will walk through instantiating objects, and adding them to an array manually. Add the following code to define a single Forecast Object and output it.

```
$dayOne= new Forecast (date("d M, Y", $today),30,20,"sunny");
echo $dayOne;
```

- Now bring up a local forecast for your city. Forecast information is widely available on the WWW. Instantiate 7 forecast objects to represent the next 7 days forecast.

Add each of the Forecast objects to the array named forecast as follows:

```
$forecast = array ();
$forecast[] = $dayOne; //assuming $dayOne defines as above.
```

Your output will now look similar to the output from Exercise 1, but will be relevant to your city, and make use of instantiated objects of type Forecast.

### EXERCISE 10.3 — ADD STATIC VARIABLES

- Continuing from our last exercise, we will now add some static variables to your Forecast class.
- In your Forecast class, add two static variables called allTimeHigh and allTimeLow . Initialize them with unreasonable values as shown in the highlighted lines below:

```
class Forecast{
    public $date;
    public $high;
    public $low;
    public $description;
    public static $allTimeHigh=100;
    public static $allTimeLow=1000;

    function __construct($d, $h, $l, $desc){
//...
```

- These static variables can now be printed, even without an instance of the class. We can therefore add the all time high and low to our page by adding a footer like:

```
<footer>
<h3>Record High: <?php echo Forecast::$allTimeHigh; ?></h3>
<h3>Record Low: <?php echo Forecast::$allTimeLow; ?></h3>
</footer>
```

Refresh the page and you will see your initialized values output.

- To demonstrate updating the variables, let us add a conditional check to our constructors for each value (new lines in red). If the low is lower than the all time low, update, and similarly for the high. Inside the class we can use the self:: syntax as follows:

```
function __construct($d, $h, $l, $desc){
    $this->date = $d;
    $this->high = $h;
    if($h>self::$allTimeHigh){
```

```

        self::$allTimeHigh=$h;
    }
    $this->low = $l;
    if($l<self::$allTimeLow){
        self::$allTimeLow=$l;
    }
    $this->description = $desc;
}

```

Your page will now update the static variable and output the actual highs and lows from your forecast data to the bottom footer.

## OBJECT-ORIENTED DESIGN

### EXERCISE 10.4 — DATA ENCAPSULATION

- 1 Our exercises in this chapter to date have used a class where all the members are public. To demonstrate the definition, usage and purpose of data encapsulation, open, examine, and test [labio-exerciseo4.php](#), which like the previous exercise outputs a forecast, but unlike the previous exercise, uses dot notation to update values before printing.

To make this code work you will have to add default values to every parameter in the constructor for Forecast.

```
function __construct($d=0, $h=0, $l=0, $desc=0){
```

- 2 As you can see your current output no longer captures the correct high and low temperatures. This is because in Exercise 3 the check to update the static variables happened in the constructor. Now, because we set the values outside the constructor, the updated highs and low don't update the static allTimeHigh and allTimeLow variables. This will be solved using encapsulation.

Change all the public (non static) variables to private.

```
//...
private $date;
private $high;
private $low;
private $description;
public static $allTimeHigh=0;
public static $allTimeLow=100;
```

Reload the page and you will get a fatal error output

Fatal error: Cannot access private property

because [labio-exerciseo4.php](#) uses -> notation to access and update the variables in the class.

- 3 To fix this problem we will write public *getter* and *setter* methods for each private variable as follows:

```

    public function getDate(){
        return $this->date;
    }
    public function setDate($d){
        $this->date = $d;
    }
    public function getHigh(){
        return $this->high;
    }

    public function setHigh($h){
        $this->high = $h;
    }
    public function getLow(){
        return $this->low;
    }
    public function setLow($l){
        $this->low = $l;
    }
    public function getDescription(){
        return $this->description;
    }
    public function setDescription($d){
        $this->description = $d;
    }
}

```

- 4 Now in [lab10-exercise04.php](#) replace the former accessing of variables directly with calls to the appropriate setter as follows:

```

for ($i=0;$i<7;$i++){
    $dayOne = new Forecast();
    $dayOne->setHigh($i*5);
    $dayOne->setLow($i*-5);
    $dayOne->setDate(date("d M, Y", $today+$i*$oneday));
    $dayOne->setDescription("Sunny");
    $forecast[]=$dayOne;
}

```

Your final page should almost look correct now, but the `allTimeHgh` and `allTimeLow` variables are still not being updated.

By adding the check for the lowest and highest values (as we did in our constructor back in Exercise 3) into our setter functions, we will ensure data is correct. Now we can rest assured that however our class is used, the `allTimeHigh` and `allTimeLow` will be updated correctly, since they can only be modified by the constructor or the setter.

Correcting this oversight will serve as a proof of concept about why encapsulation matters. The code below shows how to add to our setters for high and low with changes in red

```
public function setHigh($h){
    if($h>self::$allTimeHigh){
        self::$allTimeHigh=$h;
    }
    $this->high = $h;
}
public function setLow($l){
    if($l<self::$allTimeLow){
        self::$allTimeLow=$l;
    }
    $this->low = $l;
}
```

Now your page will contain not look like Figure 10.2, but will manage and validate the all time high and low, including every time any instance of a Forecast is updated!

### Weather forecast using classes and encapsulation

24 Feb, 2014	25 Feb, 2014	26 Feb, 2014	27 Feb, 2014	28 Feb, 2014	01 Mar, 2014
High: 0	High: 5	High: 10	High: 15	High: 20	High: 25
Low: 0	Low: -5	Low: -10	Low: -15	Low: -20	Low: -25
Sunny	Sunny	Sunny	Sunny	Sunny	Sunny

  

02 Mar, 2014
High: 30
Low: -30
Sunny

Record High: 30

Record Low: -30

Figure 10.2 Screenshot of the completed Exercise 4, with encapsulated getters and setters.

## EXERCISE 10.5 — INHERITANCE

- 1 Open and run [lab10-exercise05.php](#). Notice that it prints out a list of vehicles. Since students readily understand vehicles as a concept, they make a decent way to explain inheritance.
- 2 Open [Vehicle.class.php](#), to see the given definition for a Vehicle. Add one more class below the definition of Vehicle: *LandVehicle* as follows:

```
class LandVehicle extends Vehicle{
    private $wheels;
```

```

        function __construct($mk, $md, $f, $spd,$whlCount){
            parent::__construct($mk, $md, $f, $spd);
            $this->wheels = $whlCount;
        }
    }
}

```

- 3 Now modify the instantiation of the second vehicle in [lab10-exercise05.php](#), to be a LandVehicle, and pass 4, to say it has 4 wheels.

```
$hybridCar = new LandVehicle("Ford", "Prius", "Hybrid","160",4);
```

When you run the page, you will get error messages about the properties defined in the parent class not being accessible. To fix this, modify the declaration of the properties in the vehicle from private to protected:

```

protected $make;
protected $model;
protected $fuel;
protected $topSpeed;

```

- 4 To make the LandVehicle output a different output then the generic vehicle we must override the \_\_toString Method. For our LandVehicle add the following:

```

function __toString(){
    return ' <div class="panel panel-default col-lg-3 col-md-4 col-sm-6">
        <div class="panel-heading">
            <h3 class="panel-title">'.$this->make.'</h3>
        </div>
        <div class="panel-body">
            <table class="table table-hover"><tbody>
                <tr><td>Model:</td><td>'.$this->model.'</td></tr>
                <tr><td>Fuel:</td><td>'.$this->fuel.'</td></tr>
                <tr><td>Top Speed:</td><td>'.$this->topSpeed.' Mph</td></tr>
                <tr><td>Wheels:</td><td>'..$this->wheels.</td></tr>
            </tbody>
            </table>
        </div>
    </div>';
}

```

Notice we are accessing the \$wheels element of this class, and the elements in the parent class. Now our page will output a different box for a Land Vehicle than it does a regular base Vehicle (namely it will tell us how many wheels it has).



- 5 Finally we will create a second subclass to represent Water Vehicles. Much like LandVehicle we will add some properties for water vehicles and override the `__toString()` method. (In this case we are adding the boat's capacity, lifeboat capacity, and changing the speed to be in Knots rather than Miles per hour)

```
class WaterVehicle extends Vehicle{
    private $capacity;
    private $lifeBoatCapacity;

    function __construct($mk, $md, $f, $spd,$cap, $lifeboat){

        parent::__construct($mk, $md, $f, $spd);
        $this->capacity = $cap;
        $this->lifeBoatCapacity = $lifeboat;
    }

    function __toString(){
        return ' <div class="panel panel-default col-lg-3 col-md-4 col-sm-6">
            <div class="panel-heading">
                <h3 class="panel-title">'. $this->make. '</h3>
            </div>
            <div class="panel-body">
                <table class="table table-hover"><tbody>
                    <tr><td>Model:</td><td>'. $this->model. '</td></tr>
                    <tr><td>Fuel:</td><td>'. $this->fuel. '</td></tr>
                    <tr><td>Top Speed:</td><td>'. $this->topSpeed. '
Knots</td></tr>
                    <tr><td>Capacity:</td><td>'. $this->capacity. '</td></tr>
                    <tr><td>Life Boat Capacity:</td><td>'. $this-
>lifeBoatCapacity. '</td></tr>
                </tbody>
            </table>
            </div>
        </div>';
    }

}
```

In [lab10-exercise05.php](#) instantiate a water vehicle and echo it as follows:

```
$boat = new WaterVehicle("White Star Line", "titanic",
    "Steam", "24", 3327, 1178);
echo $boat;
```

Your output will look similar to Figure 10.3.

## A List of Vehicles

Ford	
Model:	Model-T
Fuel:	Gas
Top Speed:	30 Mph

  

Ford	
Model:	Prius
Fuel:	Hybrid
Top Speed:	160 Mph
Wheels:	4

  

White Star Line	
Model:	titanic
Fuel:	Steam
Top Speed:	24 Knots
Capacity:	3327
Life Boat Capacity:	1178

Figure 10.3 The output of Exercise 5

### EXERCISE 10.6 — ITERATING POLYMORPHIC OBJECTS

- 1 Our last exercise illustrated inheritance, so we will use that framework in this example to illustrate how inheritance allows for Polymorphism.

Open lab10-exercise05.php and replace the instantiation and echo statement with the following code which instantiates 6 vehicles.

```
$allVehicles = array(
    new Vehicle("Ford", "Model-T", "Gas", "30"),
    new LandVehicle("Ford", "Prius", "Hybrid", "130", 4),
    new WaterVehicle("White Star Line", "titanic", "Steam", "24", 3327, 1178),
    new LandVehicle("Honda", "CR-V", "Gas", "165", 4),
    new WaterVehicle("Cunard Line", "Queen Mary 2", "Diesel", "30", 2620, 2620),
    new LandVehicle("Mercedes-Benz", "Actros", "Diesel", "140", 8)
);
```

- 2 Now, since the vehicles (and sub classes) are in a single array, we can loop through the array and output each one.

```
foreach($allVehicles as $v){
    echo $v;
}
```

- 3 Your output should look like Figure 10.4. Notice that when we echo the object, the appropriate `__toString()` method is called.

## A List of Vehicles

Ford	
Model:	Model-T
Fuel:	Gas
Top Speed:	30 Mph

Ford	
Model:	Prius
Fuel:	Hybrid
Top Speed:	130 Mph
Wheels:	4

White Star Line	
Model:	titanic
Fuel:	Steam
Top Speed:	24 Knots
Capacity:	3327
Life Boat Capacity:	1178

Honda	
Model:	CR-V
Fuel:	Gas
Top Speed:	165 Mph
Wheels:	4

Cunard Line	
Model:	Queen Mary 2
Fuel:	Diesel
Top Speed:	30 Knots
Capacity:	2620
Life Boat Capacity:	2620

Mercedes-Benz	
Model:	Actros
Fuel:	Diesel
Top Speed:	140 Mph
Wheels:	8

Figure 10.4 Completed Exercise 6. Polymorphism means the correct `__toString()` method is called for output.

## EXERCISE 10.7 — USING INTERFACES

- 1 To illustrate the use of Interfaces we will continue using the vehicles example from the past few Exercises. Define the following interface:

```
interface MovingObject{
    public function getDistanceOverTime($time);
}
```

- 2 Now, in your main file modify the main output to print the new function on each object instead of the `__toString()` method.

```
echo $v->getDistanceOverTime(10);
// echo $v;
```

At first, nothing will be printed since none of the classes have *implemented* the interface.

- 3 Next, modify your Vehicle class to make it implement the Interface. This means saying you implement it, and actually providing an implementation for the function.

```
class Vehicle implements MovingObject{

    //...
    public function getDistanceOverTime($time){
        return $this->model." can travel " .($this->topSpeed*$time). "
        miles in ".$time. "hours<br/>";
    }
    //...
```

- 4 Now each of the vehicles will output something for the function. However, since some vehicles store the speed in knots and others in miles per hour, the WaterVehicle subclass will need to override this method.

First define the ratio of knots to miles per hour:

```
define("KNOTSTOMPH", 1.15078);
```

Now in the WaterVehicle subclass override the getDistanceOverTime method:

```
public function getDistanceOverTime($time){
    return $this->model." can travel " .($this->topSpeed*KNOTSTOMPH*$time). "
    miles in ".$time. "hours<br/>";
}
```

Finally, each type of class and subclass has a correct implementation for the Interface. Interfaces are nice for organizing code.