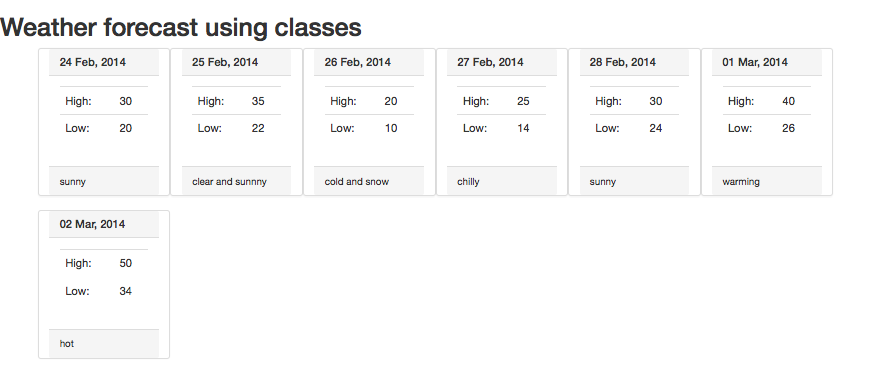
|  |  |  |  |
| --- | --- | --- | --- |
| LAB 21 (Chapter 12,15) PHP Classes and Objects  |  | | --- | | What You Will Learn   * To create and use your own classes in PHP * Some design principals to help your write good code * Some standard Object oriented design patterns |  |  | | --- | | Approximate Time  The exercises in this lab should take approximately 60 minutes to complete. | |  | |
| Web Application Development  COP3834  Professor Navarro |
| Textbook by Pearson  http://www.funwebdev.com  Adapted from Pearson, modified 11/2/21 |

|  |
| --- |
| Preparing Directories |
| 1 | Download zip folder.. |
| 2 | Uncompress to your desktop. |

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 13. — 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 12. We used parallel arrays of values to store information, and had to put all functionality where it was used. |
| 2 | Examine lab13-exercise01.php. In that file there is a reference to a *Forecast* class, and a file named Forecast.class.php. At present this page will generate errors, which this exercise will fix. |
| 3 | Create a file named Forecast.class.php. Inside that file create a new class named 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">  <tr><td>High:</td><td>'.$this->high.'</td></tr>  <tr><td>Low:</td><td>'.$this->low.'</td></tr>  </table>  </div>  <div class="panel-footer">'.$this->description.'</div>  </div>';  } |
| 5 | Test lab13-exercise01.php in the browser.  You should see output similar to that shown in Figure 13.1. If not, it is because you are using the bootstrap styling. Rename the xbootstrap directory to bootstrap. |



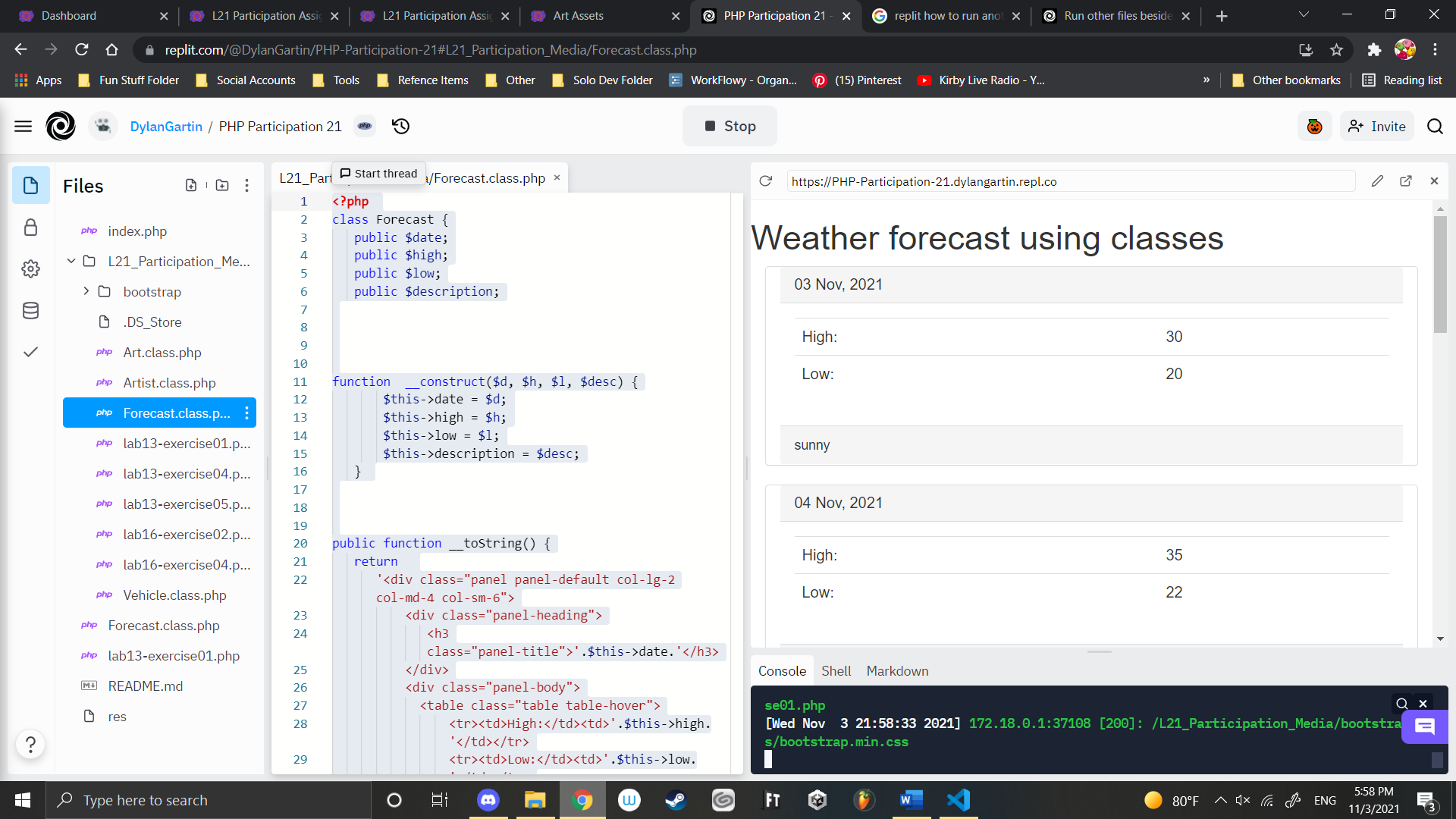
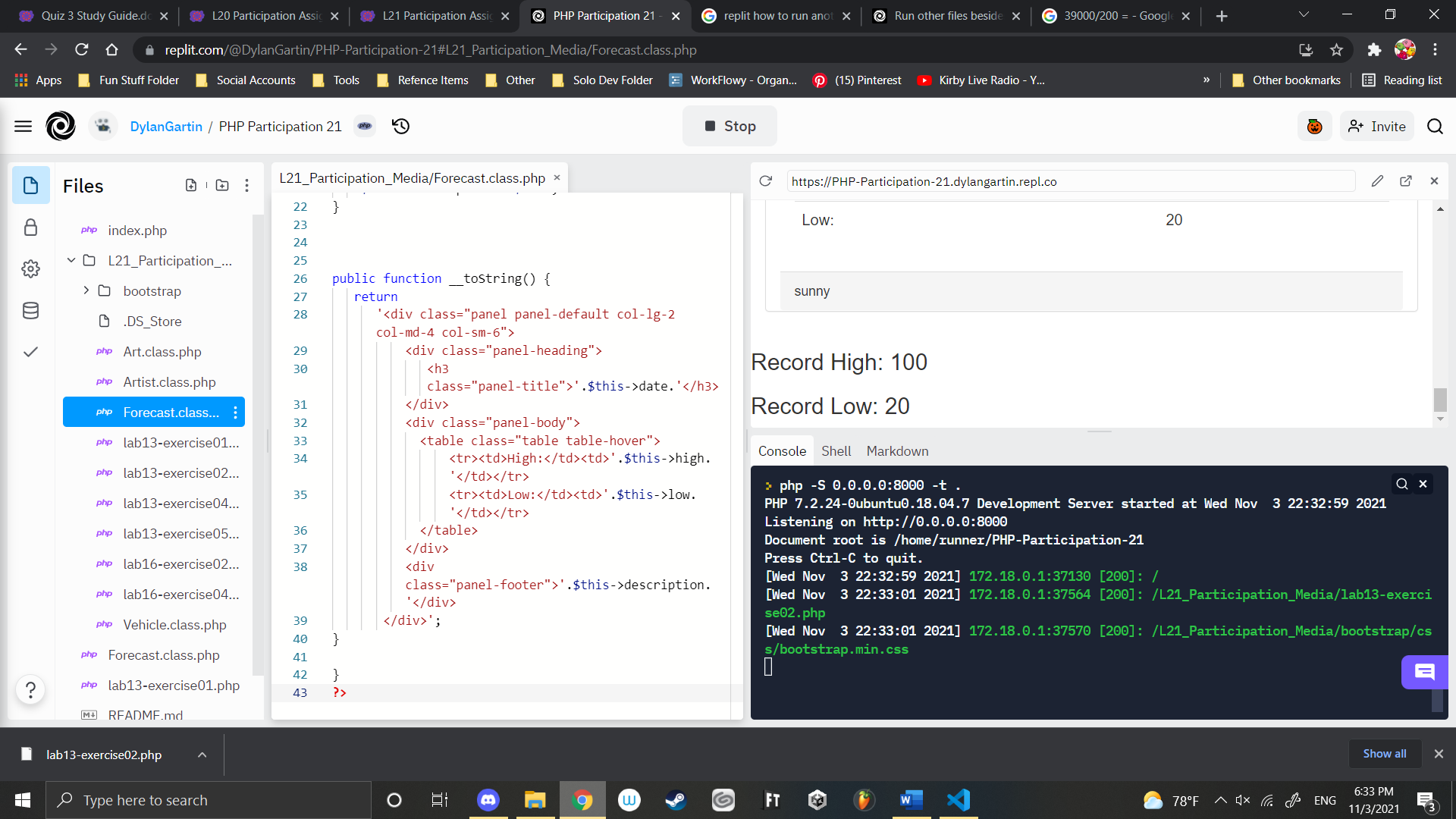


Figure 13.1 Screenshot of the output from Exercise 1

|  |
| --- |
| EXERCISE 13. — Instantiate Objects |
| 1 | In the previous exercise you wrote a constructor, but the instantiation of the Forecase objects was already done for you. This exercise will walk you through calling your own constructors. Copy the 13.1 file and create a new 13.2 file with the same code. Work on Exercise 13.2 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; |
| 3 | 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 13. — Add Static Variables |
| 1 | Continuing from our last exercise, we will now add some static variables to your Forecast class. |
| 2 | 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, $desc) {  ... |
| 3 | These static variables can now be printed, even without an instance of the class. We can therefore add the all time high and low to our page by adding a footer like:  <footer>  <h3>Record High: <?php echo Forecast::$allTimeHigh; ?></h3>  <h3>Record Low: <?php echo Forecast::$allTimeLow; ?></h3>  </footer>  Refresh the page and you will see your initialized values output. |



|  |  |
| --- | --- |
| 4 | 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 13. — 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 lab13-exercise04.php, which like the previous exercise, outputs a forecast, but unlike the previous exercise, uses dot notation to update values before outputting.  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  This happens because lab13-exercise04.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 = $d;  } |
| 4 | Now in lab13-exercise04.php replace the former accessing of variables directly with calls to the appropriate setter as follows:  for ($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 13.2, but will manage and validate the all time high and low, including every time any instance of a Forecast is updated! |

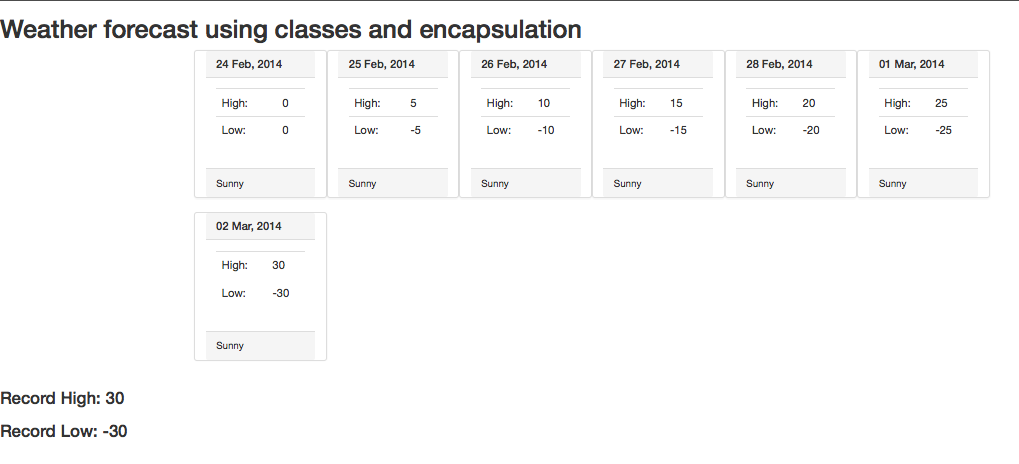


Figure 13.2 Screenshot of the completed Exercise 4, with encapsulated getters and setters.

|  |
| --- |
| EXERCISE 13. — Inheritance |
| 1 | Open and run lab13-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 lab13-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">  <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>  </table>  </div>  </div>';  }  Notice we are accessing the $wheels element of this class, and the elements in the parent class. Now out 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">  <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>  </table>  </div>  </div>';  }  } |
| 6 | In lab13-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 13.3. |

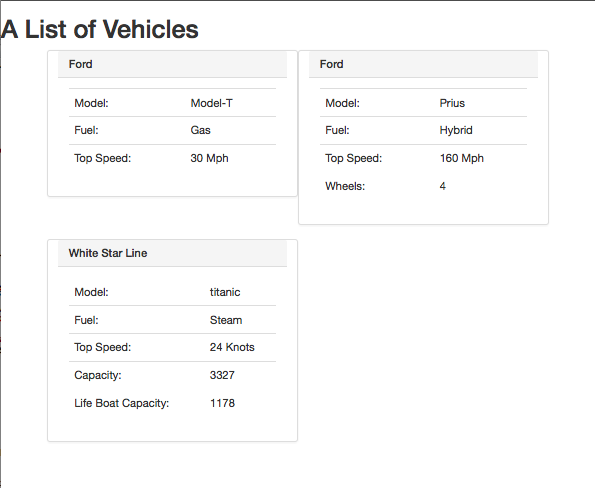


Figure 13.3 The output of Exercise 5

Broken Can’t do 16.2

|  |
| --- |
| BROKE |
| Exercise 16.2 — Serialize your objects |
| 1 | Serialization is the means by which objects and data structures can be stored as files, and then resurrected as objects when the need arises. Serialization allows us to transfer objects through a network by converting it into a byte stream. It also helps in preserving the state of the object. Open lab16-exercise02.php and the referenced Artist.class.php. This file creates several objects and prints them out to the screen. |
| 2 | To illustrate how serialization can help you with your own classes, make the Artist class implement the Serializable interface. This has been done with the class declaration, and implement two methods serilaize() and unserialize() in Artist.class.php  class Artist implements Serializable{  //...  public function serialize() {  return serialize(array("earliest" => self::$earliestDate,  "first" => $this->FirstName,  "last" => $this->LastName,  "bdate" => $this->BirthDate,  "ddate" => $this->DeathDate,  "bcity" => $this->BirthCity,  "works" => $this->artworks  ));  }  public function unserialize($data) {  $data = unserialize($data);  self::$earliestDate = $data['earliest'];  $this->FirstName = $data['first'];  $this->LastName = $data['last'];  $this->BirthDate = $data['bdate'];  $this->DeathDate = $data['ddate'];  $this->BirthCity = $data['bcity'];  $this->artworks = $data['works'];  }  //...  } |
| 3 | Now, since the Artist itself has instances or Art, we must ensure the Art classes and subclasses are also serializable.. In Art.class.php these changes are already made to serialize and unserialize those classes.  class Art implements Serializable{  //...  public function serialize() {  return serialize(  array("date" =>$this->dateCreated,  "name" => $this->name  )  );  }  public function unserialize($data) {  $data = unserialize($data);  $this->dateCreated = $data['date'];  $this->name = $data['name'];  }  //...  }  Note that the Painting and Sculpture subclasses do not need to declare that they implement Serializable, but they need to override the serialize() and unserialize() methods in order to ensure additional fields are properly stored  class Painting extends Art{  //...  public function serialize() {  return serialize(  array("med" => $this->medium,  "artData" => parent::serialize()  )  );  }  public function unserialize($data) {  $data = unserialize($data);  $this->medium = $data['med'];  parent::unserialize($data['artData']);  }  //..  }  class Sculpture extends Painting{  //..  public function serialize() {  return serialize(  array("weight" =>$this->weight,  "paintingData" => parent::serialize()  )  );  }  public function unserialize($data) {  $data = unserialize($data);  $this->weight = $data['weight'];  parent::unserialize($data['paintingData']);  }  //...  } |
| 4 | Now add the following lines to your lab16-exercise02.php to demonstrate the serialization for objects, and then their subsequent unserialization.  $picassoAsFile = serialize($picasso);  echo "<pre width='100%'>$picassoAsFile</pre>";  $picasso = unserialize($picassoAsFile);  The output from the object is unchanged, but in between a serialized version of the object is output as shown in Figure 16.2. Note only step 4 required manual update from you. |
|  |  |

Graphical user interface, text, application, email

Description automatically generated

Figure 16.2– Exercise 16.2 completed, showing the serialized objects.

|  |  |
| --- | --- |
| Exercise 16.4 — HTML5 Web Storage | |
| Open lab16-exercise04.php and notice that it is conducting a 3 question survey using sessions. In order to get the next question the last question must be submitted, requiring multiple requests to the server to conduct the survey. Nothing is done with the results in this example. | |
| We will change the page to post all 3 questions and use HTML5 storage to run the survey, collects the answers, and post at the end.  First we will remove the need for sessions to handle which question we are on (easier to debug as well). Look at line #76. This means replacing the complicated session logic with the very straightforward:  <?php  echo "<h1 id='questionNumber'>Question #1</h1>";  echo "<h2>".getSurveyQuestion(0)."</h2>";  ?>  Similarly, simplify the form to add a id (for Javascript) and change the input submit field to a button attached to a JavaScript function.  function getSurveyQuestion($i){  $questions = array("What is your favorite color?", "In what city were you  born?", "Your favorite drink is:");  $form= "  <form action='' method='get' role='form'>  <div class ='form-group'>  <label for='answer' id='question'>".$questions[$i]."</label>  <input type='text' name='answer' id='answer' class='form-control'/>  </div>";  $form.="<input type='button' value='Next' class='form-control' onclick='nextQ();'/>";  $form.="</form>";  return $form;  } | |
| Now add JavaScript tags and code to initialize all the questions in the HTML5 sessionStorage to the head section:  <script type="text/javascript">  if (typeof (localStorage) === "undefined" || typeof (sessionStorage) === "undefined") {  alert("Web Storage is not supported on this browser...");  }  else {  //gets serialized to a comma separated list of strings.  sessionStorage.setItem("Questions",  new Array("What is your favorite color?", "In what city were you born?", "Your favorite drink is:"));  sessionStorage.setItem("Answers", "");  sessionStorage.setItem("currentQuestion",0);  // document.write("web storage modified");  }  </script> | |
| Finally we will write the function that handles the button press, stores the answer and changes over to the next question.  function nextQ(){  var currentIndex = sessionStorage.getItem("currentQuestion");  var answerNode = document.getElementById("answer");  var answer = answerNode.value;  var oldAnswers = sessionStorage.getItem("Answers");  if(oldAnswers!="")  sessionStorage.setItem("Answers",oldAnswers+","+answer);  else  sessionStorage.setItem("Answers",answer);  //Now increment to Next Question  currentIndex = parseInt(currentIndex) + 1;  sessionStorage.setItem("currentQuestion",currentIndex);  var allQs = sessionStorage.getItem("Questions").split(",");  if(allQs.length<=currentIndex){  //echo for now – survey completed.  var allAs = sessionStorage.getItem("Answers").split(",");  for (var i=0;i<allQs.length;i++){  document.write(allQs[i]+":"+allAs[i]+"</br>");  }  }  else{  //Update the questions from SessionStorage  var questionNode=document.getElementById("questionNumber");  questionNode.innerHTML=("Question #"+(parseInt(currentIndex)+1));  var questionNode=document.getElementById("question");  questionNode.innerHTML=(allQs[currentIndex]);  }  }  5. Finally you will see that after the last question the values are echoes out to the screen, and still not handled by the server. Reflect on how HTML5 storage has allowed the entire survey to be conducted with only 1 request. View storage in Chrome Dev Tools/Application/Session Storage | |
| Finally you will see that after the last question the values are echoes out to the screen, and still not handled by the server.  To see where the data is stored, open up Dev tools in Google. Select Application and then Session Storage on the left. | |
| Exercise 16.5 — Writing to a file | |
| 1. Create a file called poly\_log and place in the web root or current path of your index.php. Add the following text to error\_log .   Florida Poly has about 1,500 undergraduate and graduate students and offers more than 31 programs of study. The University is the only state university in Florida dedicated exclusively to STEM education.   1. Create an index.php file to read the poly\_log file and append it to a new file. 2. Add the following code to read the file contents and assign to $text.   <?php  //write to file  $text = file\_get\_contents( poly\_log );  ?>   1. Add the following line. Where does the file output to?   echo "<h3>$text<h3>";   1. Comment out echo line and add this line next.   echo file\_put\_contents( 'poly-copy.txt', $text, FILE\_APPEND );   1. Open up the newly created file called poly-copy.txt. Does it contain the text from poly\_log? 2. Add the following code to read the contents from the Nintendo website. Then it checks if the website mentions ‘marioparty’. Checks the numeric position of the first occurrence. Is Mario Party available?   $nintendo = file\_get\_contents( 'https://www.nintendo.com' );  if( strpos( $nintendo, 'marioparty' ) ) {  echo "<p>This website has Mario Party!</p>"; }  else {  echo "<p> Mario Party not available</p>";  } | |
| 1. Add a final line to index.php to append the content from Nintendo.com to poly-copy.txt then run the program. Does it append the html code of Nintendo.com to poly-copy.txt? | |