WEB 306 – PHP: Databases and Framework

Day 5 – Object-Oriented Programming

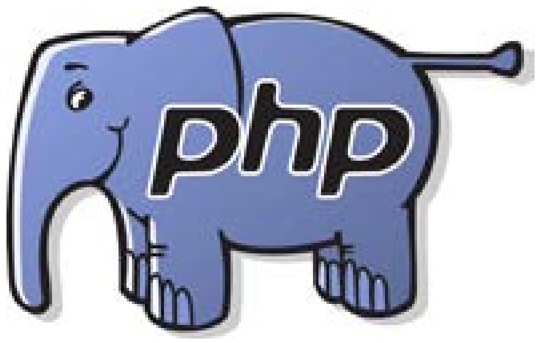
1. Object-Oriented Programming

1.1 About OOP

The set of programming conventions which structures code following the principles of objects is known as Object-Oriented Programming (or OOP).

> **"Object-oriented programming takes the view that what we really care about are the objects we want to manipulate rather than the logic required to manipulate them. Examples of objects range from human beings (described by name, address, and so forth) to buildings and floors (whose properties can be described and managed) down to the little widgets on a computer desktop (such as buttons and scroll bars)."**
>
> **— From object-oriented programming (OOP), TechTarget**

However, you don't need to use objects to be using object-oriented programming and just because you are using objects it doesn't mean that you are writing object oriented code.



1.2 The Four Principles of OOP

Object-Oriented programming is defined by four principles:

1. **Encapsulation**
2. **Abstraction**
3. **Inheritance**
4. **Polymorphism**

The goal of these principles is to make the use of an object easy for a potential third party to understand without needing to understand the structure of the class or how its methods work as well as to make the program more modular and reusable.

1.2.1 Encapsulation

Encapsulation is the idea that an object should include

PHP Resources

1. PHP Website
2. XAMPP
3. Documentation
4. W3Schools
5. CSS Tricks
6. Scotch.io
7. Stack Overflow

everything which it needs to function and that the object's properties and data should only be altered by the object itself using its methods and even restricting access to those methods. Essentially, object data should be "encapsulated" in objects.

> **"In general, encapsulation is the inclusion of one thing within another thing so that the included thing is not apparent."**
>
> **— From encapsulation, TechTarget**

1.2.2 Abstraction

Abstraction is the idea that objects should be simplified and that their data and processes should be hidden from the end user.The goal of abstraction is to reduce complexity by hiding the way which data is actually processed from the user of an object in order to place focus on what the object is actually for and prevent them from using an object incorrectly.

> **"Through the process of abstraction, a programmer hides all but the relevant data about an object in order to reduce complexity and increase efficiency. In the same way that abstraction sometimes works in art, the object that remains is a representation of the original, with unwanted detail omitted."**
>
> **— From abstraction, TechTarget**

Some people do not include "abstraction" as a principle of object-oriented programming because it is technically a principle of all programming. While this stance is true, it is a bit pedantic and one of the major goals of object-oriented programming is to increase the level of abstraction in programs. You can acheive higher levels of abstraction with object-oriented programming than without it.

1.2.3 Inheritance

Inheritance is the idea that any child classes should inherit the properties and methods of their parent classes. PHP allows child classes to be defined which inherit all of the qualities of parent classes, meaning that it supports the inheritance principle.

> **"In object-oriented programming, inheritance is the concept that when a class of objects is defined, any subclass that is defined can inherit the definitions of one or more general classes."**
>
> **— From inheritance, TechTarget**

1.2.4 Polymorphism

Polymorphism is the idea that objects should be able to do different things depending on the context they are being used without changing the core interface.

> **"In object-oriented programming, polymorphism (from the Greek meaning "having multiple forms") is the characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a variable, a function, or an object to have more than one form."**
>
> **— From polymorphism, TechTarget**

2. Using Encapsulation and Abstraction

We can use encapsulation and absctraction in order to make our `Car` class from the previous lesson more efficient and object-oriented.

```
class Car {
  public $name;
  public $driver;
  public $image;
  public $honk;
  public $top_speed;
  private $current_speed;
  private $gas;

  public function __construct($some_array) {
       $this->name = $some_array[‘name’];
       $this->driver = $some_array[‘driver’];
       $this->image = $some_array[‘image’];
       $this->honk = $some_array[‘honk’];
       $this->top_speed = $some_array[‘top_speed’];
       $this->current_speed = 0;
       $this->gas = 100;
  }

  public function start($speed) {
       // Set the speed of this car to a value between 1 and
       // $top_speed
       if ($speed > 0 && $speed < $this->top_speed) {
            $this->current_speed = $speed;
       } else if ($speed < 1) {
            $this->current_speed = 1;
       } else if ($speed > $this->top_speed) {
            $this->current_speed = $this->top_speed;
       }

       // Set the gas to $gas minus 10%
       $this->gas = $this->gas - 10;

       // Display the Car’s current speed and the amount of gas
       echo ‘<p><span class=”fa fa-tachometer-alt”></span> ‘  .
            $this->current_speed .
            ‘ <span class=”fa fa-battery-half”></span> ‘ .
            $this->gas . ‘</p>’;

       // Display an image of the Car driving
       echo ‘<p><img src=”img/start/’ . $this->image . ‘” alt=”’ .
            $this->name . ‘”></p>’;
  }

  public function honk() {
        echo '<audio autoplay><source src="audio/honk/' .
        $this->honk . '"></audio>';
    }

    public function doughnuts() {
        // Set the current speed to the top speed
        $this->current_speed = $this->top_speed;

        // Set the gas to the gas minus 20
        $this->gas = $this->gas - 20;

        // Display the Car's current speed and the amount of gas
        echo '<p><span class="fa fa-tachometer-alt"></span> '  .
            $this->current_speed .
            ' <span class="fa fa-battery-half"></span> ' .
            $this->gas . '</p>';

        // Display an image of the Car doing doughnuts
        echo '<p><img src="img/doughnuts/' . $this->image .
            '" alt="' . $this->name . '"></p>';
    }

    public function fill_up() {
        // Stop the car by setting the current speed to 0
        $this->current_speed = 0;

        // Set the gas back to 100%
        $this->gas = 100;

        // Display the Car's current speed and the amount of gas
        echo '<p><span class="fa fa-tachometer-alt"></span> '  .
            $this->current_speed .
            ' <span class="fa fa-battery-half"></span> ' .
            $this->gas . '</p>';
    }
}
```

2.1 Private and Protected Scope

The purpose of private and protected property and method scope is to keep the code object-oriented by keeping the data encapsulated inside of the object and to keep the processing of that data abstracted from the user. You would use private properties and methods for properties and methods which you don't want your user to be able to have direct access to.

In an effort to follow the encapsulation and abstraction principles as closely as possible, many developers, especially those who come to PHP with exerience using other object-oriented languages such as C++ or Java, would argue that every non-static (see ***2.2 The static Keyword***) property which is defined inside of a class should be a private or protected property. To define a property as private, use the `private` keyword.

```
  private $name;
  private $driver;
  private $image;
  private $honk;
  private $top_speed;
  private $current_speed;
  private $gas;
```

Methods would be defined as private or protected less frequently but when private or protected methods are used they would be used for internal data processing within a class. We have some repeated code in our public methods which could be moved to private methods which are reused inside of the public methods.

For example, we use code to display our car's current speed and the amount of gas in multiple places throughout our class so we could store this code in a private method and call the method whenever we want to use it. As with properties, to define a method as private, use the `private` keyword.

```
  private function dashboard() {
    // Display the Car's current speed and the amount of gas
    echo '<p><span class="fa fa-tachometer-alt"></span> '  .
        $this->current_speed . ' <span class="fa fa-battery-half">
        </span> ' . $this->gas . '</p>';
  }
```

We also display an image of our car in multiple places. While we display different images, the only difference in our code is that we change the name of the folder the image is in depending on what the car is doing. This means that we can create a private method which displays an image of the car and simply alter the type of image by using a `$folder` parameter as a placeholder for the folder name.

```
  private function show($folder) {
    // Display an image of the Car
    echo '<p><img src="img/' . $folder . '/' . $this->image . '" alt="'
        .    $this->name . '"></p>';
  }
```

2.2 The static Keyword

2.2.1 Defining Static Properties

Static properties, also known as "class properties," are properties which can be accesed without creating a new instance of a class. These properties cannot be accessed from objects, or instances of a class, and are instead accessed using the name of the class itself. As a result, they act more like variables which are associateed with a specific class instead of as properties.

To define a static property use the `static` keyword before the property name but after the scope keyword (i.e. `public`, `private`, `protected`).

```
  private static $chars = '/([!@#$%^&*()[]{}<>:;])/';
  private static $name_chars = '/([!@#$%^&*()[]{}<>:;0123456789])/';
```

If no privacy level is specified for a static property, it is assumed that it is a public property. Static properties should be used sparingly. When they are used they are frequently used as counters.

2.2.3 Defining Static Methods

Static methods process data which is not dependant on the creation of an instance of the class. Unlike static properties, static methods can be accessed from either the class name itself, or from individual objects, or instances of a class. However, because static methods are not associated with a specific instance of a class, and hold the same data regardless of which instance they are being accessed from, they do not have access to the `$this` keyword. While static methods are used fairly frequently in PHP, it is considered a bad practice to overuse them and it is best to only create them when you are sure that nothing in the method is dependant on an instance of the class or the `$this` variable.

To define a static method use the `static` keyword before the method definition but after the scope keyword (i.e. `public`, `private`, `protected`).

```
public static function filter_input($string, $special_chars = '//') {
   // this method would be static
}
```

If no privacy level is specified it will be assumed that the method is public. The order of the static keyword in relation to the scope keyword does not make a technical difference but it is considered best practice to place the `static` keyword after the scope keyword.

2.2.4 The self Keyword

Because static properties and methods do not rely on an instance of a class, they are not accessed from the instance's variable name outside of the class or the `$this` variable inside of the class. Static properties cannot be accessed from individual objects, or instances a class, at all. In theory, static methods could be accessed this way but it would not be common practice to do so.

Instead, static properties and methods are accessed using the class name, when accessing them outside of the class, and the `self` keyword when accessing them inside of the class. The class name or `self` keyword is then followed by the scope resolution operator.

2.2.5 The Scope Resolution Operator (a.k.a. Paamayim Nekudotayim)

The scope resolution operator is written as `::` two colons, is also known as ***Paamayim Nekudotayim***, which means "double colon" in Hebrew and is used to access static properties and methods. It is comparable to the `->` arrow which is used with instantiated classes.

```
// Outside the class
echo Car::filter_input("some string", Car::$name_chars);
```

```
// Inside the class
public static function format_name($name) {
   $sanitized = self::filter_input($name, self::$name_chars);
}
```

2.3 Sanitization and Validation Methods

Our user input needs to be sanitized. If we're following the encapsulation and abstraction principles, this should be done inside of our class. We can create sanitization methods for our `Car` class. Before we do so, let's diffrentiate some common terminology.

2.3.1 Filtering vs. Escaping vs. Validation vs. Sanitization

These terms tend to be used interchangably but are generally used to describe the following processes.

1. **Filtering: Removing potentially malicious characters from user input. Filtering functions include `trim()`, `strip_tags()` and `preg_replace()`.**
2. **Escaping: Converting potentially malicious characters to their escape sequence code i.e `<` to `&lt;` or `&` to `&amp;` so that they are read as plain text instead of code. Escaping functions include `htmlspecialchars()` and `htmlentities()`.**
3. **Validation: The act of checking user input to ensure that it has been filled out, is of the correct data type and that it matches authorized characters and values. Validation functions include `isset()`, `empty()`, `is_string()`, `is_numeric()` and `is_array()`.**
4. **Sanitization: The act of using filtering, escaping, validation or a combination of those processes to ensure that user input is not malicious and will not provoke unwanted or unauthorized behaviour. The `filter_var()` and `filter_input()` functions can be used for any type of sanitization.**

2.3.2 Input vs. Output

So far we have done validating, filtering and escaping of data at the same time, in the same function. It is best practice to separate these processes and perform them at different stages. User input should be validated and filtered when it is collected from a superglobal and escaped when it is retreived from a database or file and displayed as output.

2.3.3 Filtering Generic Input

To filter generic input we will create a `filter_input()` method using the `trim_space()` function which removes white space from the beginning and end of a string, the `strip_tags()` function which tries to remove any HTML tags from a string and the `preg_replace()` function which can remove any remaining special characters from the string using regular expressions.

```
public static function filter_input($string, $special_chars = ‘//’) {
   $trim_space = trim($string);
   $strip_tags = strip_tags($trim_space);
   $remove_chars = preg_replace($special_chars, ‘’, $strip_tags);
   return $remove_chars;
}
```

2.3.4 Filtering and Formatting Names

We would want to validate and filter each type of data using unique processes. To filter names we would want to run them through our generic `filter_input()` method to remove unwanted characters but may want to remove additional characters such as numbers from names.

Formatting of names using `strtolower()` and `ucwords()` can be done when user input is collected as well. We will create a `format_name()` method to do this.

```
public static function format_name($name) {
   $filtered = self::filter_input($name, self::$name_chars);
   $lowercase = strtolower($filtered);
   $capitalized = ucwords($lowercase);
}
```

2.3.5 Filtering and Validating Select, Radio and Checkbox Fields

We are going to allow our user to select image and audio files using an HTML `<select>` tag with preset options.

```
<select name=”image” id=”image”>
   <option value=”car.gif”>Car</option>
</select>
```

Even preset data which comes from select fields, radio inputs and checkbox inputs should be filtered and sanitized. Even though the data is preset, it could still be easily altered by the user using modern browser tools such as the web inspector.

To filter file names from select fields we would first run them through our generic `filter_input()` method to remove unwanted characters. We could then shift all letters to lowercase to ensure that the casing of the input matches with the names of our files, as servers are case sensitive.

Finally we can loop through an array of authorized file names and check each one to ensure that the provided file name matches one of them. We will do this using a `filter_file()` method.

```
public static function filter_file($input_file, $files_array) {
   $filtered = self::filter_input($input_file, self::$chars);
   $file_name = strtolower($filtered);

   foreach ($files_array as $item_name) {
         if ($item_name == $file_name) {
               return $file_name;
         } // end if
   } // end foreach
} // end filter_file
```

We could then use this method like this.

```
$image_names = [‘car.gif’, ‘homer.gif’, ‘mcqueen.gif’,
   ‘studebaker.gif’];

$this->image = self::filter_file($value, $image_names);
```

2.3.6 Filtering and Validating Numbers

All data is initially read as a string when submitted using an HTML form, even if you use an HTML input with a `type=”number”` attribute. The HTML input type could also be easily altered using browser tools such as the web inspector which would allow a user to enter whatever strings they want. To filter and validate integers we would first want to remove all characters except for numbers and arithmatic operators.

We could do this using the `filter_var()` function and the `FILTER_SANITIZE_NUMBER_INT` constant. Finally, to convert the string into an integer we could then cast the string as an integer by using the `int` keyword. We will do this using a `filter_number()` method.

```
public static function filter_number($number) {
   $filter_int = filter_var($number, FILTER_SANITIZE_NUMBER_INT);
```

```
    return (int) $filter_int;
  }
```

2.3.7 Escaping Output

Escaping special characters to convert them to plain text using functions such as `htmlspecialchars()` or `htmlentities()` should normally only be done after retreiving data from a database or file and displaying it as output. This is because you can not properly escape the characters in your data unless you know the final format which your data will be used in.

If you are sure that that your data will only ever be used in HTML then you could escape your input using `htmlspecialchars()` before saving it to a database. However, if it is possible that it could be used in a different format then you would want to avoid doing so until you display the data.

This way you can use the escape characters of whichever language you are using the data as output in on a case by case basis.

We will make an `escape_html()` method to escape output to be displayed in our HTML. The best practice is to convert all characters to the UTF-8 character set using `mb_convert_encoding()` before using `htmlspecialchars()` to ensure that non UTF-8 characters are not missed.

```
  public static function escape_html($string) {
    $convert_charset = mb_convert_encoding($string, ‘UTF-8’, ‘UTF-8’);
    $encode_html = htmlspecialchars($string, ENT_QUOTES, ‘UTF-8’);
    return $encode_html;
  }
```

2.4 Getter and Setter Methods

If we are following the encapsulation and abstraction principles strictly and have made all of our properties private or protected then that means that we can not currently access or alter any of them outside of our class. This is the goal of encapsulation and abstraction as it means that the data can not be directly altered by the user and they can not use that data incorrectly.

However, sometimes it is necessary to allow users to access and alter properties outside of our class. The common solution to this is to create methods which are dedicated to accessing and defining properties called ***“getters”*** and ***“setters”***.

2.4.1 Getter Methods

A getter method would get the value of a private property by simply returning it like this:

```
  public function get_name() {
    return $this->name;
  }
```

You would then access the private variable data by calling the getter method.

```
  echo batmobile->get_name();
```

2.4.2 Setter Methods

A setter method would be used to define and redefine a private property by taking in a value as a parameter, doing any necessary data processing and defining the variable inside of the method.

```
public function set_name(string $value) {
    $this->name = self::format_name($value);
}
```

You would then set the value of the variable by calling the setter method.

```
$batmobile->set_name("Adam West's Batmobile");

echo batmobile->get_name(); # echos Adam West's Batmobile
```

This might seem pointless or overly complex at first but the benefit of this approach is that you can make sure that any data which is entered into an object is appropriately processed and that users are prevented from using your class' properties incorrectly. In this example, we are sanitizing any data the user enters by removing unwanted characters and are formatting it as a proper name.

5.6.3 Setter Examples

There are a lot of useful ways which setter methods can be used to filter and process data that users provide. Each property may have its own processing requirements. The `$driver` property requires the same type of processing as the `$name` property.

```
public function set_driver(string $value) {
    $this->driver = self::format_name($value);
}
```

The `$image` setter should filter the string, check to ensure that the referenced image file exists and use a default image if it does not.

```
public function set_image(string $value) {
    $image_names = ['car.gif', 'homer.gif', 'mcqueen.gif',
          'studebaker.gif'];

    $this->image = self::filter_file($value, $image_names);

    if (empty($this->image)) {
          $this->image = $image_names[0];
    }
}
```

The `$honk` setter would need to follow the same process but for MP3 files.

```
public function set_honk(string $value) {
    $mp3_names = ['honk.mp3', 'homer.mp3', 'kermit.mp3', 'wow.mp3'];

    $this->honk = self::filter_file($value, $mp3_names);

    if (empty($this->honk)) {
          $this->honk = $mp3_names[0];
    }
}
```

The `$top_speed` setter could check if the input includes only numbers (integers, floats, or a string with numbers) using the `is_numeric()` function and run it through the `filter_number()` method if it does not.

```
  public function set_top_speed(string $value) {
     if (is_numeric($value)) {
          $this->top_speed = (int) $value;
     } else {
          $this->top_speed = self::filter_number($value);
     }
  }
```

We would want to rewrite our internal methods and especially the `__construct()` method so that they use our setter methods to define properties to ensure that the appropriate data processing is applied to them.

```
  public function __construct($some_array) {
     $this->set_name($some_array['name'];);
     $this->set_driver($some_array['driver'];);
     $this->set_image($some_array['image'];);
     $this->set_honk($some_array['honk'];);
     $this->set_top_speed($some_array['top_speed'];);
     $this->current_speed = 0;
     $this->gas = 100;
  }
```

6. Using Inheritance and Polymorphism

6.1 Creating Child Classes

We can create child classes of classes which inherit all of the properties and methods of the parent class. Child classes should represent subcategories of a class which would share all of the properties and methods of that base class.

For example, our `Car` class could have child classes of `Honda`, `Subaru` and `Batmobile`.

6.1.2 Requiring Parent Classes

At some point, either within the child class file or when the new instance of the child class is created, it will be necessary to require the file which the parent class was defined in to use it. If you are only using the child class then it would make sense to require the parent class inside the child class file.

```
  require_once 'Car.php'; // If we are requiring inside the child file
```

If you are using both the parent class and the child then it would make more sense to require both before you create new instances of them.

```
  require_once '../classes/Car.php'; // Requiring with a new instance
  require_once '../classes/Batmobile.php';
```

Either way, you may only require each class once.

6.1.3 Defining Child Classes

You would define a child class in almost the same way as a regular class except you would add the extends keyword at the end of the class name followed by the parent class name.

```
class Honda extends Car {}
```

```
class Subaru extends Car {}
```

```
class Batmobile extends Car {}
```

6.2 Inheriting Parent Properties and Methods

All public and protected properties and methods of parent classes are automatically inherited by child classes. You would also want to define any properties which you want to be inherited into the child class as protected in the parent class instead of private.

```
protected $name;
protected $driver;
protected $image;
protected $honk;
protected $top_speed;
protected $current_speed;
protected $gas;
```

6.2.1 Child __construct() Methods

This class would have its own `__construct()` method which would override the parent method. Polymorphism describes a scenario where a child class with the same name as a method from the parent class overrides that method to behave in a different way. Our child constructor will have the same parameters as the parent class but could have additional parameters as well.

```
class Batmobile extends Car {
   public function __construct($bat_array) {
   }
}
```

6.2.2 The parent Keyword

To have the child class inherit the values of the parent class' properties, use the `parent` keyword to refer to the parent class.

The `parent` keyword just acts as a placeholder for the parent class. This means that we can use it to access properties and methods of the parent class. This includes the parent class' `__construct()` contructor method which we can use to define the parameters of that method using the parameters of the child method.

When we define those parameters, they will then set the properties of the object to map to the parameters as specified in the parent class' `__construct()` method. If desired, we could also set hardcoded values for the parameters of the parent class' `__construct()` method.

Properties and methods of the parent class are accessed using the `::` scope resolution operator (or the ***Paamayim Nekudotayim***).

```
class Batmobile extends Car {
   public function __construct($bat_array) {
         parent::__construct($bat_array);
   }

}
```

6.3 Unique Child Properties

Our child class can also have its own unique properties.

```
class Batmobile extends Car {
   private $bat_tech;

   public function __construct($bat_array) {
         parent::__construct($bat_array);

         $this->set_bat_tech($bat_array['bat_tech']);
   }
}
```

6.4 Unique Child Methods

Our child class can also have its own unique methods. Any properties which are unique to the child class should have their own getter and setter methods.

```
public function get_bat_tech() {
   return $this->bat_tech;
}

public function set_bat_tech(string $value) {
   $this->bat_tech = parent::filter_input($value);
}
```

6.5 Polymorphic Methods

Polymorphism allows us to use the same properties and methods to acheive different results depending on the context. For example, all of our `Car` classes have images and honks, but the Batmobile might have different images and honks to choose from.

```
public function set_image(string $value) {
   $image_names = ['batmobile.gif', 'tumbler.gif', 'animated.gif',
'lego.gif'];

   $this->image = self::filter_file($value, $image_names);

   if (empty($this->image)) {
         $this->image = $image_names[0];
   }
```

```
  }

  public function set_honk(string $value) {
     $mp3_names = ['holy.mp3', 'lego.mp3'];

     $this->honk = self::filter_file($value, $mp3_names);

     if (empty($this->honk)) {
          $this->honk = $mp3_names[0];
     }
  }
```

7. Creating New Object Instances From User Input

Now that we know how to create new object instances manually, let's allow our users to create objects using a form. In a file called `add-car.php`, we're going to create a form with an action which will point to a seperate include file in our includes folder called `process-add-car.inc.php` where we will process the data.

```
  <form action="includes/process-add-car.inc.php" method="post">
     <label for="name">Name:</label>
     <input type="text" name="name" value="" id="name">

     <label for="driver">Name:</label>
     <input type="text" name="driver" value="" id="driver">

     <select name=”image” id=”image”>
          <option value=”car.gif”>Car</option>
     </select>

     <select name=”honk” id=”honk”>
          <option value=”honk.gif”>Car</option>
     </select>

     <label for="top_speed">Name:</label>
     <input type="number" name="top_speed" value="170" id="top_speed">

     <input type="hidden" name="type" value=" <?php echo
          Car::escape_html($_GET['type']); ?>">

     <button type="submit" name="button"><span class="fa fa-plus">
          </span> Add Car</button>
  </form>
```

This form will have inputs for our car name and driver, select fields for our image and honk sound and a number field for our top speed.

In `process-add-car.inc.php` we're going to import our form data using the `$_POST` superglobal. First we will check to see if our required fields have been filled out. If so, redirect to the "view" page with a success message, else go back to the "add" page with an error message.

```
  if (!empty($_POST['name']) && !empty($_POST['driver'])) {
     header('Location: ../view-cars.php?add=success');
     exit();
  } else {
     header('Location: ../add-cars.php?add=error');
     exit();
  }
```

To create a new instance of an object, use the `new` keyword. Rather than manually creating an array we're going to use the `$_POST` array as the parameter of our `__construct()` method.

```
  if ($_POST['type'] == 'batmobile') {
     $car = new Batmobile($_POST);
  } else {
     $car = new Car($_POST);
  }
```

In order to store and transport objects, we sometimes need to use the `serialize()` function to convert the object data to a non-relational database in the form of one string.

```
  $serialized_car = serialize($car);
```

Serialized objects look like this.

```
  O:3:"Car":7:{s:4:"name";s:8:"Some Car";s:6:"driver";s:8:"Some
  Guy";s:10:" Car image";s:7:"car.gif";s:9:" Car
  honk";s:8:"honk.mp3";s:9:"top_speed";s:3:"170";s:13:"current_
  speed";i:0;s:3:"gas";i:100;}
```

We are going to store our object in a `.txt` file where we can access it at a later point using the `file_put_contents()` function. The first parameter of this function is our file path, The second parameter is our object data concatenated with our new line. `PHP_EOL` is a PHP keyword which just moves text to a new line. The third parameter is a keyword which specifies that the content should be "appended" to the previous content rather than replacing it.

```
  $file_path = '../cars.txt';
  $add_new_line = PHP_EOL;
  $text_line = $serialized_car . $add_new_line;
  $append = FILE_APPEND;
  file_put_contents($file_path, $text_line, $append);
```

The combined processing code looks like this.

```
  if (!empty($_POST['name']) && !empty($_POST['driver'])) {

     if ($_POST['type'] == 'batmobile') {
          $car = new Batmobile($_POST);
     } else {
          $car = new Car($_POST);
     }

     $serialized_car = serialize($car);

     $file_path = '../cars.txt';
```

```
    $add_new_line = PHP_EOL;
    $text_line = $serialized_car . $add_new_line;
    $append = FILE_APPEND;

    file_put_contents($file_path, $text_line, $append);

    header('Location: ../view-cars.php?add=success');
    exit();
  } else {
    header('Location: ../add-cars.php?add=error');
    exit();
  }
```

8. Displaying Object Data

In order to access our `Car` data we must require our `Car` classes.

```
  require_once '../classes/Car.php';
  require_once '../classes/Batmobile.php';
```

We can use the `file()` function to import the data in the `.txt` file where we saved our serialized objects.

```
  $car_array = file('cars.txt');
```

We will loop through each `$car_index` and `$car_value` in `$car_array`. Because we used the `serialize()` function to convert our object data to a string we must use the `unserialize()` function to convert the data back to its initial format.

```
  foreach ($car_array as $car_index => $car_value):
    $unserialized_car = unserialize($car_value);
  endforeach;
```

We can now use the `->` arrow to access our object properties and methods. We'll use our car's index number to create a unique ID.

```
  <div class="car" id="car_<?php echo $car_index; ?>"></div>
```

Now we'll display the car's name and driver.

```
  <div class="car" id="car_<?php echo $car_index; ?>">
    <h3><span class="fa fa-car"></span> <?php echo
        Car::escape_html($unserialized_car->get_name()); ?></h3>
    <h4><span class="fa fa-car"></span> <?php echo
        Car::escape_html($unserialized_car->get_driver()); ?></h4>
  </div>
```

We will check if a `$_GET` key with our car's ID is set and if the `$_GET` value matches this action, we will perform this method.

```
  if (isset($_GET['car_' . $car_index])):
    if ($_GET['car_' . $car_index] == 'start'):
        $unserialized_car->start(80);
    elseif ($_GET['car_' . $car_index] == 'honk'):
        $unserialized_car->honk();
    elseif ($_GET['car_' . $car_index] == 'doughnuts'):
        $unserialized_car->doughnuts();
    elseif ($_GET['car_' . $car_index] == 'fill_up'):
        $unserialized_car->fill_up();
    endif;
  endif;
```

Using anchor tags, we can add **$_GET** data to trigger each method.

```
  <a href="<?php echo $this_file . '?car_' . $car_index . '=start' .
  '#car_' . $car_index; ?>" class="button"><span class="fa fa-key"></
  span> Start Car</a>
  <a href="<?php echo $this_file . '?car_' . $car_index . '=honk'
  . '#car_' . $car_index; ?>" class="button"><span class="fa fa-
  bullhorn"></span> Honk Horn</a>
  <a href="<?php echo $this_file . '?car_' . $car_index . '=doughnuts' .
  '#car_' . $car_index; ?>" class="button"><span class="fa fa-sync"></
  span> Do Doughnuts</a>
  <a href="<?php echo $this_file . '?car_' . $car_index . '=fill_up' .
  '#car_' . $car_index; ?>" class="button"><span class="fa fa-battery-
  full"></span> Fill Up Gas</a>
```