

# Module 5

---

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

## Module 5 Study Guide and Deliverables

Readings: Murach's PHP and MySQL: Chapters 5, 14, & 19

Assignments: Assignment 5 due Tuesday, February 21 at 6:00 AM ET

Live Classroom:

- Tuesday, February 14, 6:00–8:00 PM ET
- Facilitator live office: TBD

## ■ Object Oriented Programming Paradigm

## Learning Objectives

---

By reading the lectures and textbook, participating in the discussions, and completing the assignments, you will be able to do the following:

- Describe the object oriented programming paradigm.
- Define classes and instantiate objects with PHP.
- Use advanced object oriented programming concepts.
- Implement programs using the MVC design pattern.
- Identify some of the more popular PHP MVC frameworks available today.

## Introduction

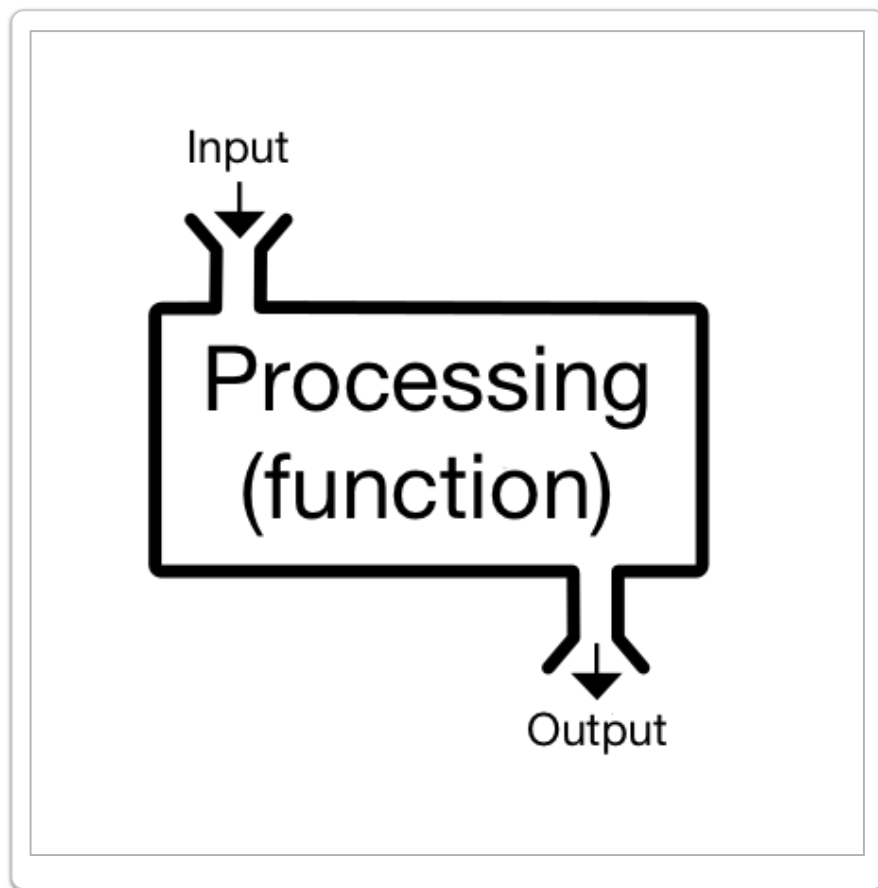
---

In this lecture, you'll learn about the object-oriented approach to building web applications. You'll start with the very basics and then move on to some more advanced features of this paradigm. We will also discuss the Model View Controller design/architectural pattern and implement a small stock quoting application using MVC. Finally, we will briefly cover some of the more popular MVC frameworks that are available for PHP programmers.

# Object Oriented Programming Paradigm

---

A paradigm a way of approaching or thinking about things, you could also call it a thought process. One way to explain object oriented programming (OOP) is to contrast it against something we already know. In the previous module, you learned how to create PHP programs in a “procedural” manner. The variables and functions were normally separated from each other. We took a variable, gave it to a function as input, and the function gave us back some type of output after it was done processing. It looked something like this:



With OOP, we take our variables and functions and wrap them together into a single object. We then refer to the variables as properties and/or attributes and our functions as methods, and sometimes even behaviors.

Note: You'll also hear the terms member variables and member functions. These terms refer to properties and methods, respectfully.

Here is a high level look:

Let's take a dog and use it as an example for this explanation. Our dog is going to be an object. We know that objects have attributes and behaviors. What attributes could we derive from our dog? How about:

- Breed
- Color
- Age
- Weight
- Friendliness

- Name

Those are just a few, we could certainly think of more if needed, but I hope you are getting the general idea here. Now let's look at our dog's behaviors (methods):

- Eat
- Sleep
- Run
- Jump
- Bark
- Lick
- Fetch

Those are all things that dogs can do.

So an object contains both properties and behaviors. This is the important distinction between OOP and procedural programming.

## Benefits of Object Oriented Programming

---

It is important to note that you don't necessarily "need" to use OOP. It does have its advantages, especially when our programs become larger. Switching to OOP can bring a sense of simplicity to complex programs. OOP gives us the ability to create modular, manageable, and organized code. It also gives us the structure needed to help with scalability.

Probably the most noticeable benefits come with abstraction and encapsulation. Encapsulation is very important as it allows us to define our data and program structures together in an object and hide the implementation from whoever or whatever needs to use our objects. Instead, we provide an interface that exposes the functionality while restricting direct access. An analogy would be that you shouldn't need to directly interact with an engine in order to operate a vehicle. You just need to be able to control the speed and direction of the vehicle and you can do that with a gas pedal, brakes, and a steering wheel. In this case, our engine is the implementation and our gas pedal, brakes, and steering wheel are the interfaces provided to us to operate the vehicle.

We also gain some very powerful concepts with inheritance and polymorphism. We'll see how all of this works as we start building our programs with an OOP approach.

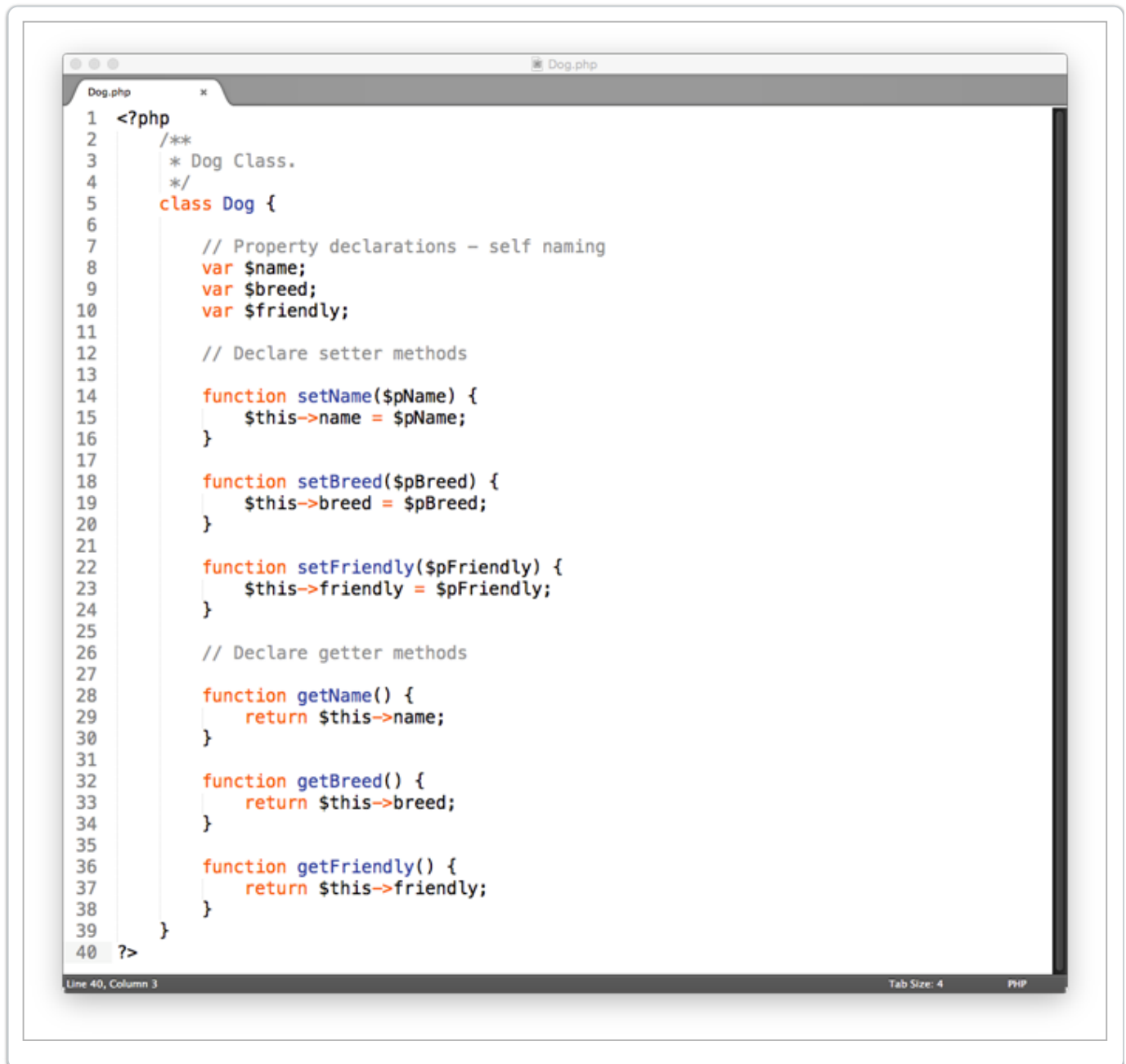
## Classes

---

Before we start using objects, we need to talk about classes. A class is simply a blueprint for an object. When we create our objects, we use a class as the template that specifies how the object is built. A class is not an object, it is just used to describe one. An object cannot be created without a class.

The class defines the attributes and behaviors that all objects derived from the class will have. Classes must define the basic building blocks of the objects.

Let's look at a class definition in PHP:

A screenshot of a code editor window titled 'Dog.php'. The code defines a PHP class named 'Dog'. It includes property declarations for \$name, \$breed, and \$friendly, and sets of setter and getter methods for each property. The code is as follows:

```
1 <?php
2 /**
3  * Dog Class.
4  */
5 class Dog {
6
7     // Property declarations - self naming
8     var $name;
9     var $breed;
10    var $friendly;
11
12    // Declare setter methods
13
14    function setName($pName) {
15        $this->name = $pName;
16    }
17
18    function setBreed($pBreed) {
19        $this->breed = $pBreed;
20    }
21
22    function setFriendly($pFriendly) {
23        $this->friendly = $pFriendly;
24    }
25
26    // Declare getter methods
27
28    function getName() {
29        return $this->name;
30    }
31
32    function getBreed() {
33        return $this->breed;
34    }
35
36    function getFriendly() {
37        return $this->friendly;
38    }
39 }
40 ?>
```

The editor interface shows line numbers from 1 to 40 on the left. The status bar at the bottom indicates 'Line 40, Column 3', 'Tab Size: 4', and 'PHP'.

The file was saved as Dog.php.

You can see that we use a keyword class and then choose a name for the class, which in this case, we selected a name of Dog. We have an opening and closing set of curly braces in which we put all of the member code of the class within.

We create our member variables, or properties, prefixing it with the var keyword. The variables are \$name, \$breed, and \$friendly.

Note: We will eventually discuss visibility or access modifiers. The var keyword was originally depreciated in PHP 5, but has made a comeback as of PHP 5.3. In this case, var is an alias for the public access modifier.

We then declare a few methods. The types of functions we have created here are known as setters and getters. Setters allow us to set the value of properties for an object and getters allow us to get the value of the properties from an object. We've created a setter and getter method for each property within the class. Just as mentioned above, we'll cover access modifiers later and for now, we will just use the function keyword for declaring our methods.

You may have noticed some new syntax inside of our methods. Let's focus on these a bit:

- `$this` : A special variable that is always available to an object within its scope. It refers to the current object.
- `->` : The object operator. It is simply a hyphen (or dash) and a greater than sign combined together to form an arrow. It is placed between the object and the property or method that you want to access.

If we were to view this file in our web browser, we would get a blank page. All we have done is create the blueprint for an object, we haven't actually created one yet. We'll do that in the next section.

Some things to keep in mind about classes in PHP:

- Class name must start with a letter or underscore.
- The remaining characters must be letters, numbers, or underscores.
- Best practice encourages you to name your classes using UpperCamel notation, being sure to capitalize the first letter of your class name.
- Properties should be written in regular camel case or have words separated with underscores.

## Class Constants

We can also include constants in our class declarations. You already know that a constant is like a variable and can hold any value, but it can't be changed after the program starts to run. Another way to say that is that constants are immutable. In our class declarations, constant declarations are preceded by the `const` keyword.

## Objects

---

We have already learned a little bit about objects when we covered classes in the previous section. One thing to understand about an object is that it is a data structure that is more complex than the primitive data types we learned about in Module 1. As you've learned, an object can contain multiple data types.

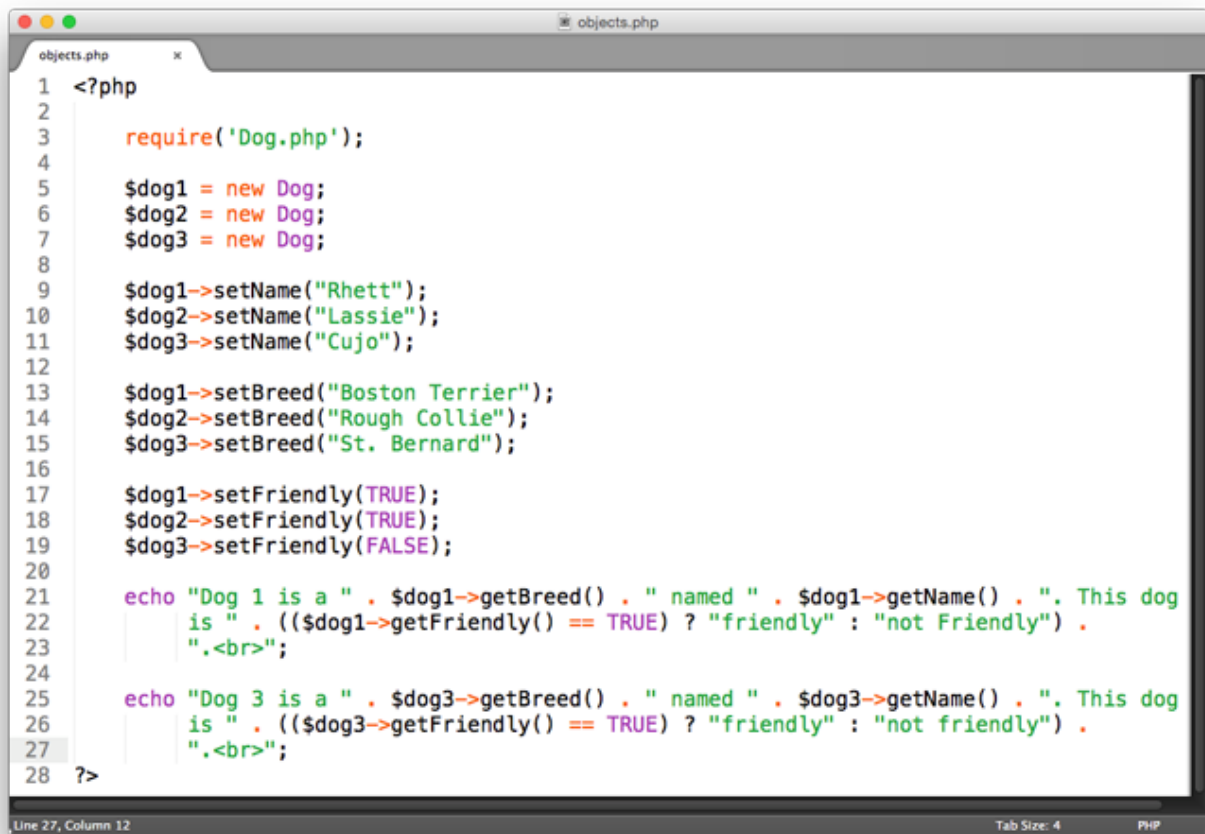
Once you have your class created, it is fairly simple to create objects belonging to that class. In order to create a new instance of an object, declare a regular variable and assign it to: `new ClassName`;

It will look something like this:

```
$dog1 = new Dog;
```

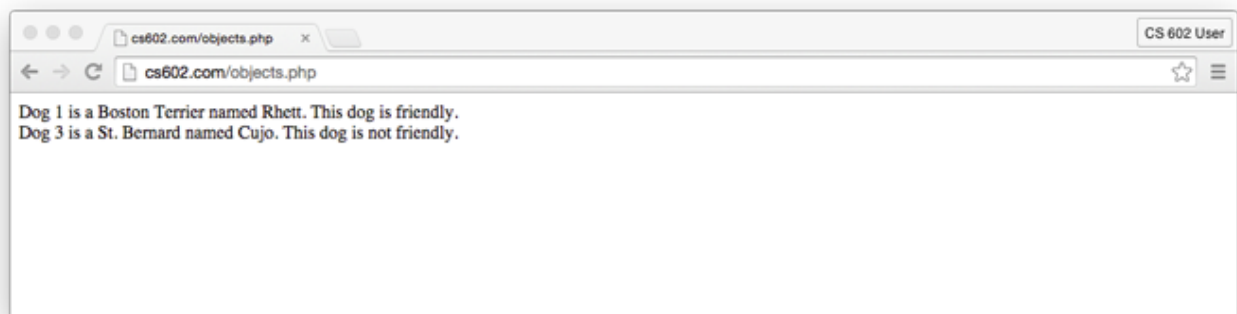
From there, we can access the properties and methods of the object as needed.

Let's create some Dog objects based off of our Dog class:



```
1 <?php
2
3     require('Dog.php');
4
5     $dog1 = new Dog;
6     $dog2 = new Dog;
7     $dog3 = new Dog;
8
9     $dog1->setName("Rhett");
10    $dog2->setName("Lassie");
11    $dog3->setName("Cujo");
12
13    $dog1->setBreed("Boston Terrier");
14    $dog2->setBreed("Rough Collie");
15    $dog3->setBreed("St. Bernard");
16
17    $dog1->setFriendly(TRUE);
18    $dog2->setFriendly(TRUE);
19    $dog3->setFriendly(FALSE);
20
21    echo "Dog 1 is a " . $dog1->getBreed() . " named " . $dog1->getName() . ". This dog
22    is " . (($dog1->getFriendly() == TRUE) ? "friendly" : "not friendly") .
23    ".<br>";
24
25    echo "Dog 3 is a " . $dog3->getBreed() . " named " . $dog3->getName() . ". This dog
26    is " . (($dog3->getFriendly() == TRUE) ? "friendly" : "not friendly") .
27    ".<br>";
28 ?>
```

This is what we will see in our browser window:



You can see that we've created a new file and saved it as objects.php. In this file, we start off with:

```
require('Dog.php');
```

This line tells PHP that it needs to copy the contents of the specified file and include it in the current file. You can also use `include()`, `include_once()`, or `require_once()`. `require` and `include` do the same thing, but they react differently if there is an error. `require()` halts execution if an error is encountered while `include()` simply provides a warning message. The `include_once` and `require_once` variations do the same thing as their simpler counter parts with the additional feature of having PHP check to see if the file has already been included, and if so, it will not include (or require) it again.

After the require statement, you can see that we instantiate three different dog objects. We then set the names, breed, and friendly properties using the object's setter methods.

Finally, we string together an echo statement that accesses the properties of the objects using the getter methods.

We've also introduced some new code here, the ternary operator, which isn't for exclusive use with objects. We can use this anywhere we would normally use an if else statement, it's just a shorter way to express the same thing.

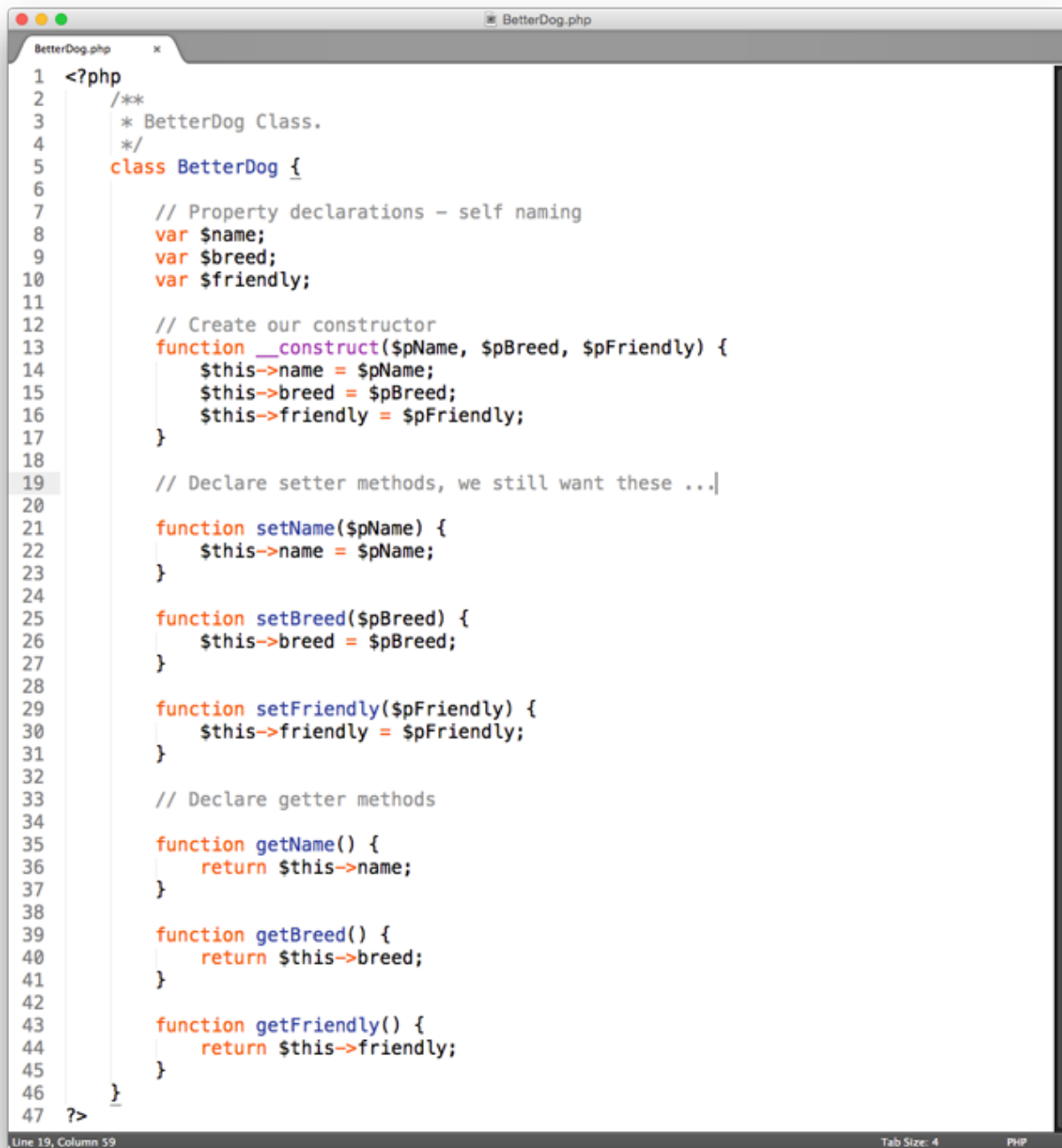
## Constructors

---

When we covered classes and objects in the previous section, we intentionally did not use a constructor. A constructor is a special method that is called when we instantiate an object. Technically, this special method is known as a magic method in PHP and we denote this with two underscores at the beginning of its name.

The constructor is automatically called when we instantiate an object. It is used to set up or configure the object.

Let's go ahead and revise the previous class and object files in order to show how constructors are implemented and used.



```
1 <?php
2 /**
3  * BetterDog Class.
4  */
5 class BetterDog {
6
7     // Property declarations - self naming
8     var $name;
9     var $breed;
10    var $friendly;
11
12    // Create our constructor
13    function __construct($pName, $pBreed, $pFriendly) {
14        $this->name = $pName;
15        $this->breed = $pBreed;
16        $this->friendly = $pFriendly;
17    }
18
19    // Declare setter methods, we still want these ...|
20
21    function setName($pName) {
22        $this->name = $pName;
23    }
24
25    function setBreed($pBreed) {
26        $this->breed = $pBreed;
27    }
28
29    function setFriendly($pFriendly) {
30        $this->friendly = $pFriendly;
31    }
32
33    // Declare getter methods
34
35    function getName() {
36        return $this->name;
37    }
38
39    function getBreed() {
40        return $this->breed;
41    }
42
43    function getFriendly() {
44        return $this->friendly;
45    }
46 }
47 ?>
```

Line 19, Column 59 Tab Size: 4 PHP

In the example above, we saved the file as BetterDog.php and have also changed our class line signature to reflect BetterDog as the name of the class.

Let's go ahead and look at our new objects file that utilizes the constructor we created in our BetterDog class. We've changed our objects file name to objects2.php





```
1 <?php
2
3     require('BetterDog.php');
4
5     $dog1 = new BetterDog("Rhett", "Boston Terrier", TRUE);
6     $dog2 = new BetterDog("Lassie", "Rough Collie", TRUE);
7     $dog3 = new BetterDog("Cujo", "St. Bernard", FALSE);
8
9     echo "Dog 1 is a " . $dog1->getBreed() . " named " . $dog1->getName() . ". This dog
10         is " . (($dog1->getFriendly() == TRUE) ? "friendly" : "not friendly") .
11         "<br>";
12
13     echo "Dog 3 is a " . $dog3->getBreed() . " named " . $dog3->getName() . ". This dog
14         is " . (($dog3->getFriendly() == TRUE) ? "friendly" : "not friendly") .
15         "<br>";
16 ?>
```

When we load this version in our browser, this is what we will see:



cs602.com/objects2.php x CS 602 User

cs602.com/objects2.php

Dog 1 is a Boston Terrier named Rhett. This dog is friendly.  
Dog 3 is a St. Bernard named Cujo. This dog is not friendly.

You should be able to determine that this is the same result as our previous example. We just had to write a lot less code in the latter because our constructor allows us to give it a few parameters and it sets up our object for us.

## Destructors

`__destruct()` is ran when an object needs to be destroyed. PHP calls a destructor as soon as all references to an object are freed up or when the script terminates. You don't need to call this explicitly and you really shouldn't unless you have a specific need to do so.

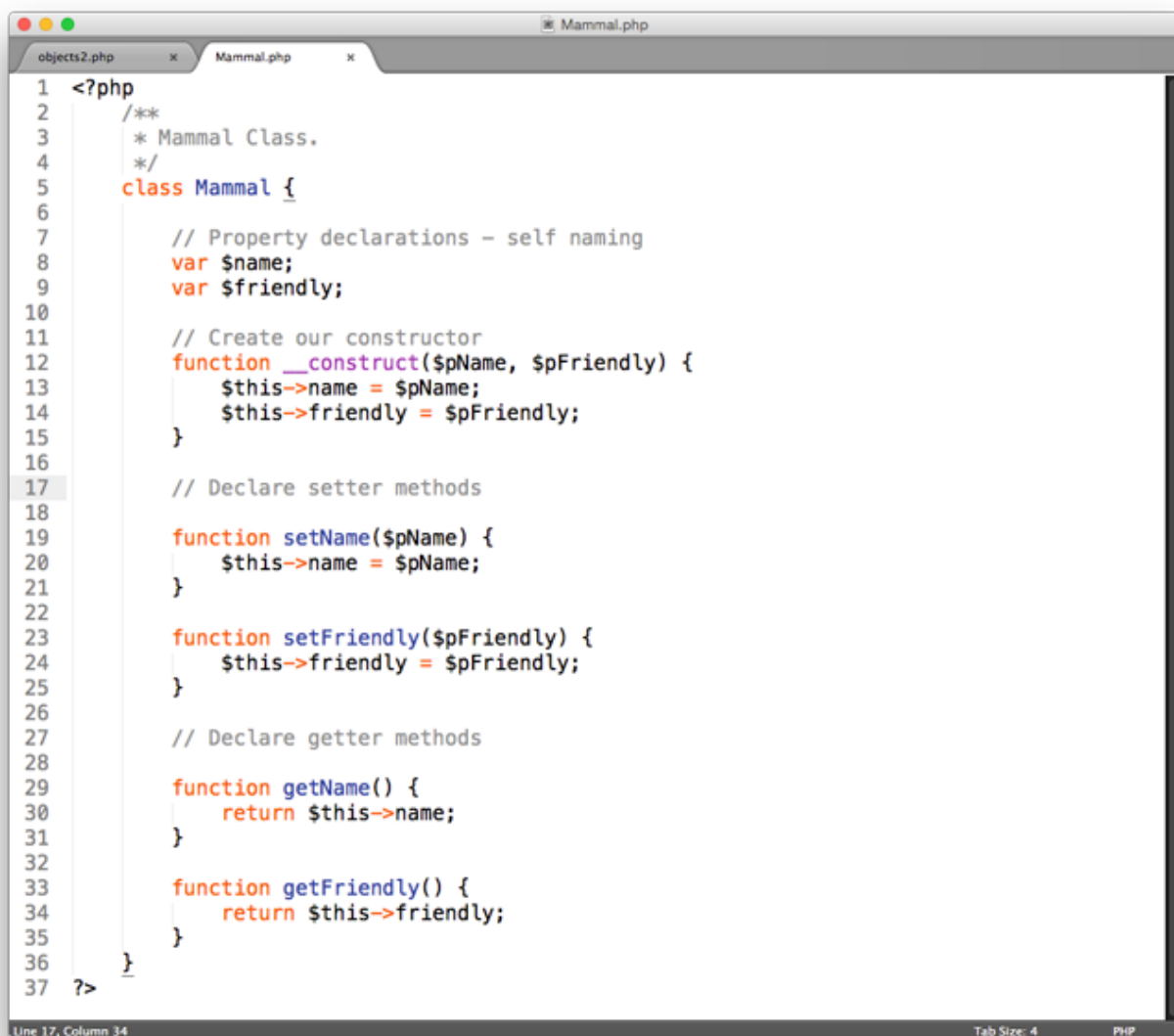
## Inheritance

Another powerful feature of object oriented programming is the ability to reuse code. OOP allows us to define relationships between classes. We can organize our classes and take into consideration the commonalities between similar classes. This is the core of what “Inheritance” is all about.

Put another way, with inheritance, a child class can inherit the properties and methods of a parent class. Because this is available to us, we should always be “thinking more abstractly” when we start designing our programs with OOP.

Let’s return back to our example involving dogs. We were very specific when we created the dog class, so much so that it really isn’t feasible to create child classes from the Dog class. We could create a subclass (child) named small dog to try and show a distinction between large dogs and small dogs, but that really doesn’t give us the best example of inheritance. What would make sense is to find a way to create a parent class of Dog that we could then create other child classes from. What do dog’s and cat’s have in common? They are both mammals, so let’s create a Mammal class and we can then recreate our dog class as a child of Mammal and also create a Cat child class as well.

Let’s start off with creating the Mammal class:

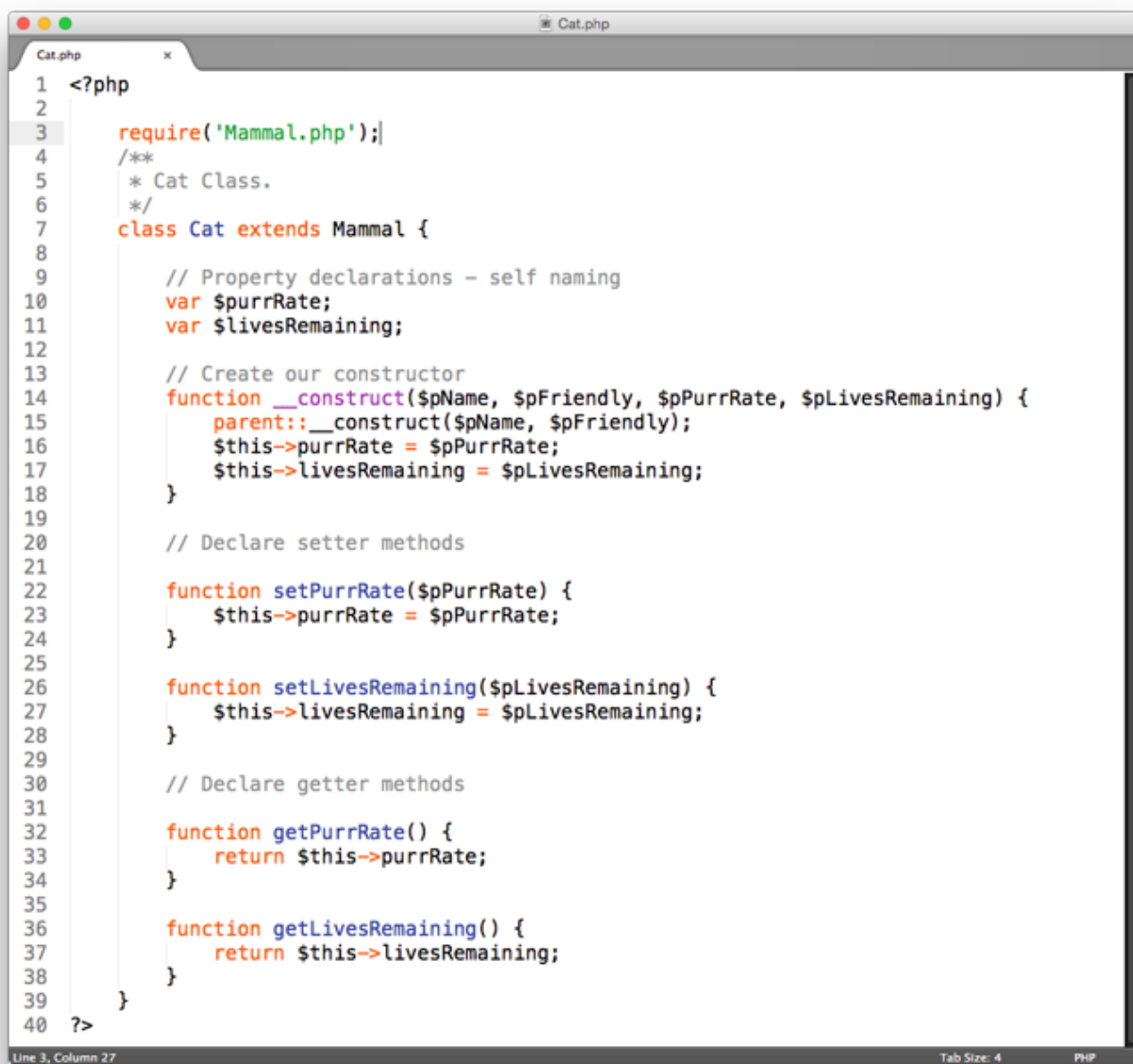


```
1 <?php
2 /**
3  * Mammal Class.
4  */
5 class Mammal {
6
7     // Property declarations - self naming
8     var $name;
9     var $friendly;
10
11     // Create our constructor
12     function __construct($pName, $pFriendly) {
13         $this->name = $pName;
14         $this->friendly = $pFriendly;
15     }
16
17     // Declare setter methods
18
19     function setName($pName) {
20         $this->name = $pName;
21     }
22
23     function setFriendly($pFriendly) {
24         $this->friendly = $pFriendly;
25     }
26
27     // Declare getter methods
28
29     function getName() {
30         return $this->name;
31     }
32
33     function getFriendly() {
34         return $this->friendly;
35     }
36 }
37 ?>
```

Line 17, Column 34 Tab Size: 4 PHP

It is probably worth noting that even though we can, we probably won't be creating any Mammal objects. Instead, we will be creating Dog and Cat objects. So let's go ahead and create these two child classes.

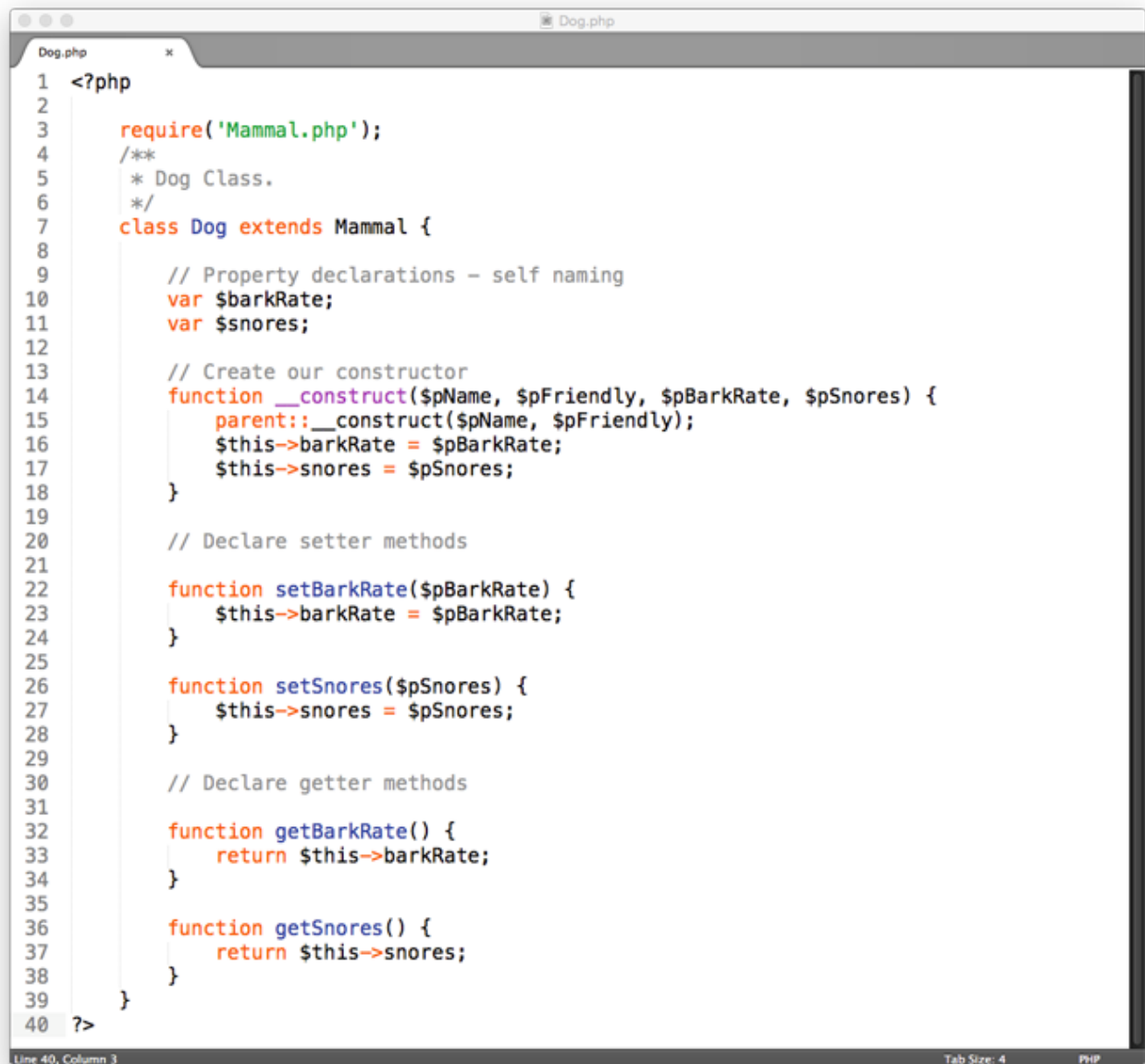
Cat.php



```
1 <?php
2
3 require('Mammal.php');|
4 /**
5  * Cat Class.
6  */
7 class Cat extends Mammal {
8
9     // Property declarations - self naming
10    var $purrrRate;
11    var $livesRemaining;
12
13    // Create our constructor
14    function __construct($pName, $pFriendly, $pPurrRate, $pLivesRemaining) {
15        parent::__construct($pName, $pFriendly);
16        $this->purrrRate = $pPurrRate;
17        $this->livesRemaining = $pLivesRemaining;
18    }
19
20    // Declare setter methods
21
22    function setPurrRate($pPurrRate) {
23        $this->purrrRate = $pPurrRate;
24    }
25
26    function setLivesRemaining($pLivesRemaining) {
27        $this->livesRemaining = $pLivesRemaining;
28    }
29
30    // Declare getter methods
31
32    function getPurrRate() {
33        return $this->purrrRate;
34    }
35
36    function getLivesRemaining() {
37        return $this->livesRemaining;
38    }
39 }
40 ?>
```

Line 3, Column 27 Tab Size: 4 PHP

Dog.php



```

1  <?php
2
3  require('Mammal.php');
4
5  /*
6   * Dog Class.
7   */
8  class Dog extends Mammal {
9
10     // Property declarations - self naming
11     var $barkRate;
12     var $snores;
13
14     // Create our constructor
15     function __construct($pName, $pFriendly, $pBarkRate, $pSnores) {
16         parent::__construct($pName, $pFriendly);
17         $this->barkRate = $pBarkRate;
18         $this->snores = $pSnores;
19     }
20
21     // Declare setter methods
22
23     function setBarkRate($pBarkRate) {
24         $this->barkRate = $pBarkRate;
25     }
26
27     function setSnores($pSnores) {
28         $this->snores = $pSnores;
29     }
30
31     // Declare getter methods
32
33     function getBarkRate() {
34         return $this->barkRate;
35     }
36
37     function getSnores() {
38         return $this->snores;
39     }
40 }
41 ?>

```

Line 40, Column 3      Tab Size: 4      PHP


You can see in both examples that we require the Mammal.php file. We then use the extends keyword after the class name and then indicate the parent class name. Everything else within these files should be familiar to you except for this line of code within the constructor:

```
parent::__construct($pName, $pFriendly);
```

That line calls the parent constructor and passes along the two parameters it needs to setup the object. The double colon is known as the Scope Resolution Operator. This operator provides access to static, constant, and overridden properties or methods of a class.

Now let's look at the files where we declare our Cat and Dog objects:

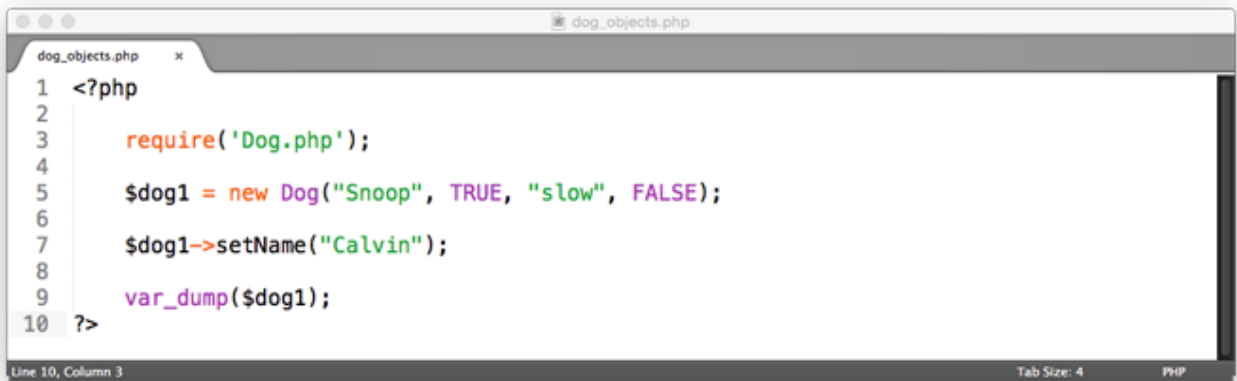
cat\_objects.php



```
1 <?php
2
3     require('Cat.php');
4
5     $cat1 = new Cat("Fred", TRUE, "fast", 2);
6
7     var_dump($cat1);
8 ?>
```

Line 1, Column 1 Tab Size: 4 PHP

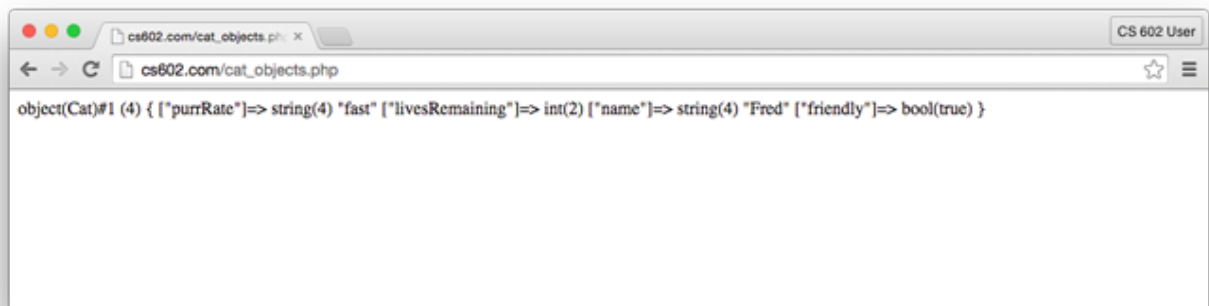
dog\_objects.php



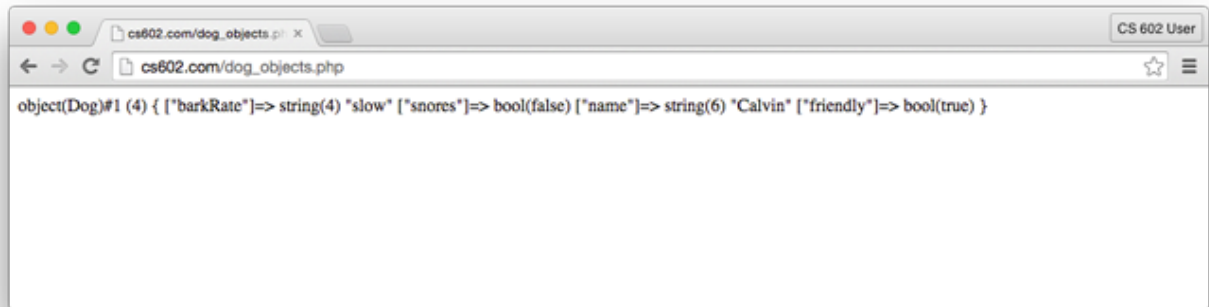
```
1 <?php
2
3     require('Dog.php');
4
5     $dog1 = new Dog("Snoop", TRUE, "slow", FALSE);
6
7     $dog1->setName("Calvin");
8
9     var_dump($dog1);
10 ?>
```

Line 10, Column 3 Tab Size: 4 PHP

And here are their respective screen shots when viewed in the browser:



```
object(Cat)#1 (4) { ["purrRate"]=> string(4) "fast" ["livesRemaining"]=> int(2) ["name"]=> string(4) "Fred" ["friendly"]=> bool(true) }
```



You can see that we did something different when we created our Dog object compared to when we created our Cat object. For the Dog object, we used the setName() method on it to change the name to a different value from what we used to create the object. There is no setName() method in the Dog class, however there is one in the Mammal class. Since Dog is a subclass of Mammal, it can call the methods that are contained within the Mammal class.

## Overriding Methods

You can override method declarations in a parent class by declaring a method with the same name in the child class. In our Dog example, let's say that we originally named our dog when we instantiated the object, but we now want to pass the string "later" to setName() as an indication that we have changed our mind and haven't come up with a name yet for our Dog.

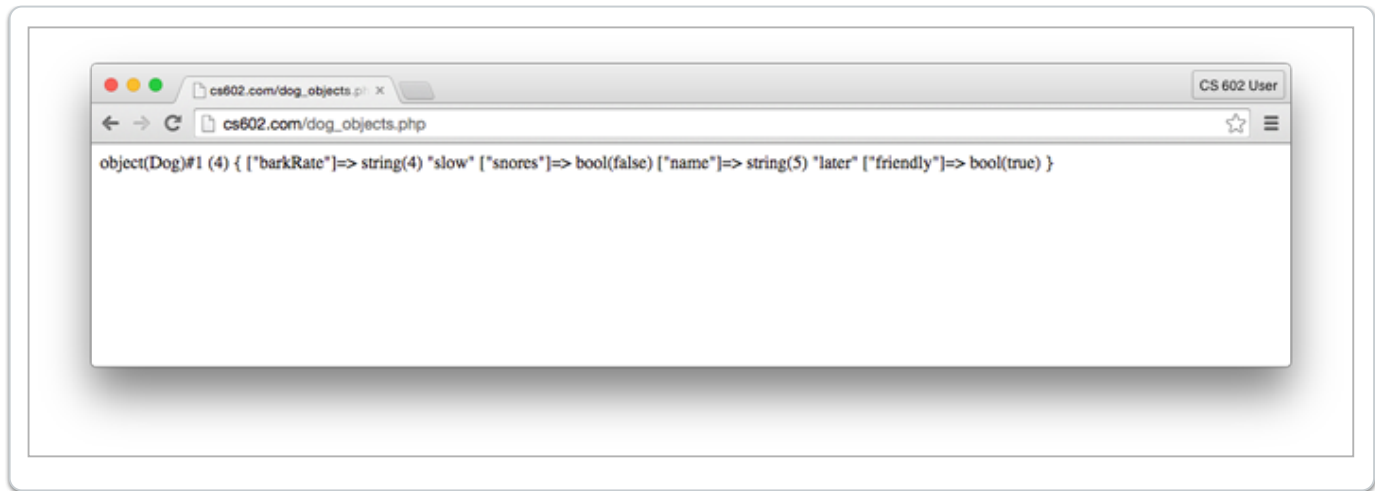
We can accomplish that like this:

A screenshot of a code editor window titled 'dog\_objects.php'. The code is as follows:

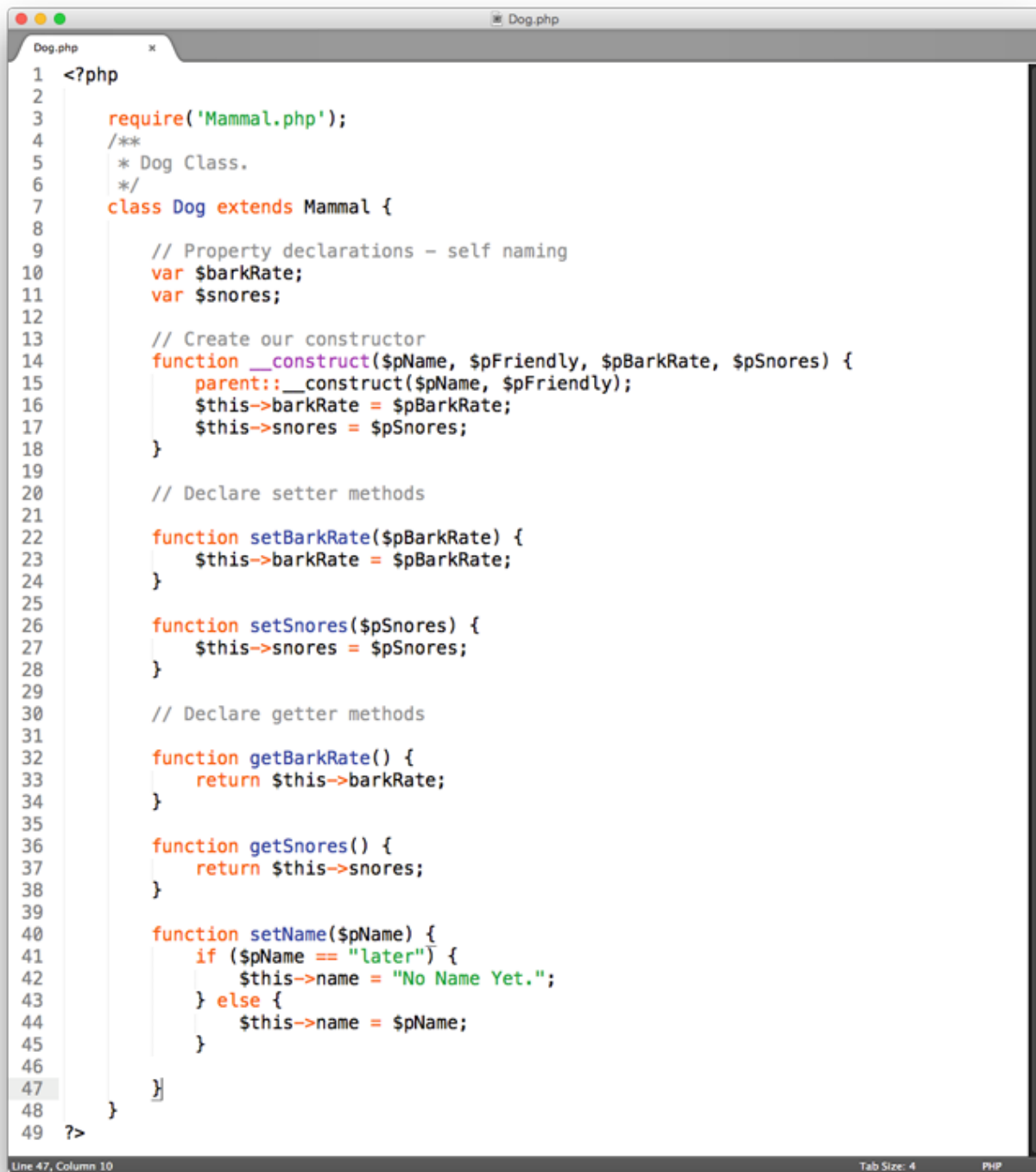
```
1 <?php
2
3     require('Dog.php');
4
5     $dog1 = new Dog("Snoop", TRUE, "slow", FALSE);
6
7     $dog1->setName("later");
8
9     var_dump($dog1);
10 ?>
```

The status bar at the bottom indicates 'Line 8, Column 6', 'Tab Size: 4', and 'PHP'.

This is what we will see in our browser:



This is the behavior we should expect from `setName()`. That method belongs to the parent class of `Dog` and it simply assigns what we pass in as a parameter to the instance variable `$name` for our `Dog` object. Maybe that is what we want for most `Mammals`, including `Cats`, but we want it to work differently for `Dog` objects. We can do that by overriding the `setName()` method inside of the `Dog` class like this:



```
1 <?php
2
3 require('Mammal.php');
4 /**
5  * Dog Class.
6  */
7 class Dog extends Mammal {
8
9     // Property declarations - self naming
10    var $barkRate;
11    var $snores;
12
13    // Create our constructor
14    function __construct($pName, $pFriendly, $pBarkRate, $pSnores) {
15        parent::__construct($pName, $pFriendly);
16        $this->barkRate = $pBarkRate;
17        $this->snores = $pSnores;
18    }
19
20    // Declare setter methods
21
22    function setBarkRate($pBarkRate) {
23        $this->barkRate = $pBarkRate;
24    }
25
26    function setSnores($pSnores) {
27        $this->snores = $pSnores;
28    }
29
30    // Declare getter methods
31
32    function getBarkRate() {
33        return $this->barkRate;
34    }
35
36    function getSnores() {
37        return $this->snores;
38    }
39
40    function setName($pName) {
41        if ($pName == "later") {
42            $this->name = "No Name Yet.";
43        } else {
44            $this->name = $pName;
45        }
46    }
47 }
48
49 ?>
```

Line 47, Column 10      Tab Size: 4      PHP

Now if we run the code listed in the screen shot above, we will see this in the browser:





Yes, there are other ways that we could have accomplished this, like assigning NULL to the \$name instance variable. But this example is intended to show you how we can use the same name of a function in a child class that has already been declared in the parent class in order to override the behavior for the method when used for a child class object.

## Access Modifiers

---

We have been using the var keyword to declare our member variables up to this point. As mentioned earlier, the var keyword is simply an alias for public in PHP 5.3+.

Public is one type of access modifier (also called a visibility modifier), but there are two others as well that we will explain here:

- Public
  - Properties and methods can be accessed outside of the class in which it is declared, from within the class in which it is declared, and from within another class that implements the class in which it is declared.
- Private
  - Properties and methods are limited to the class in which it is declared. They can not be accessed from outside of the class or from classes that inherit the class in which it is declared. (It is best to use this sparingly)
- Protected
  - Properties and methods are accessible in the class in which it is declared and in classes that extend that class. They are not available anywhere else.

What the access modifiers really end up doing is defining the scope of the properties and methods of the object or class.

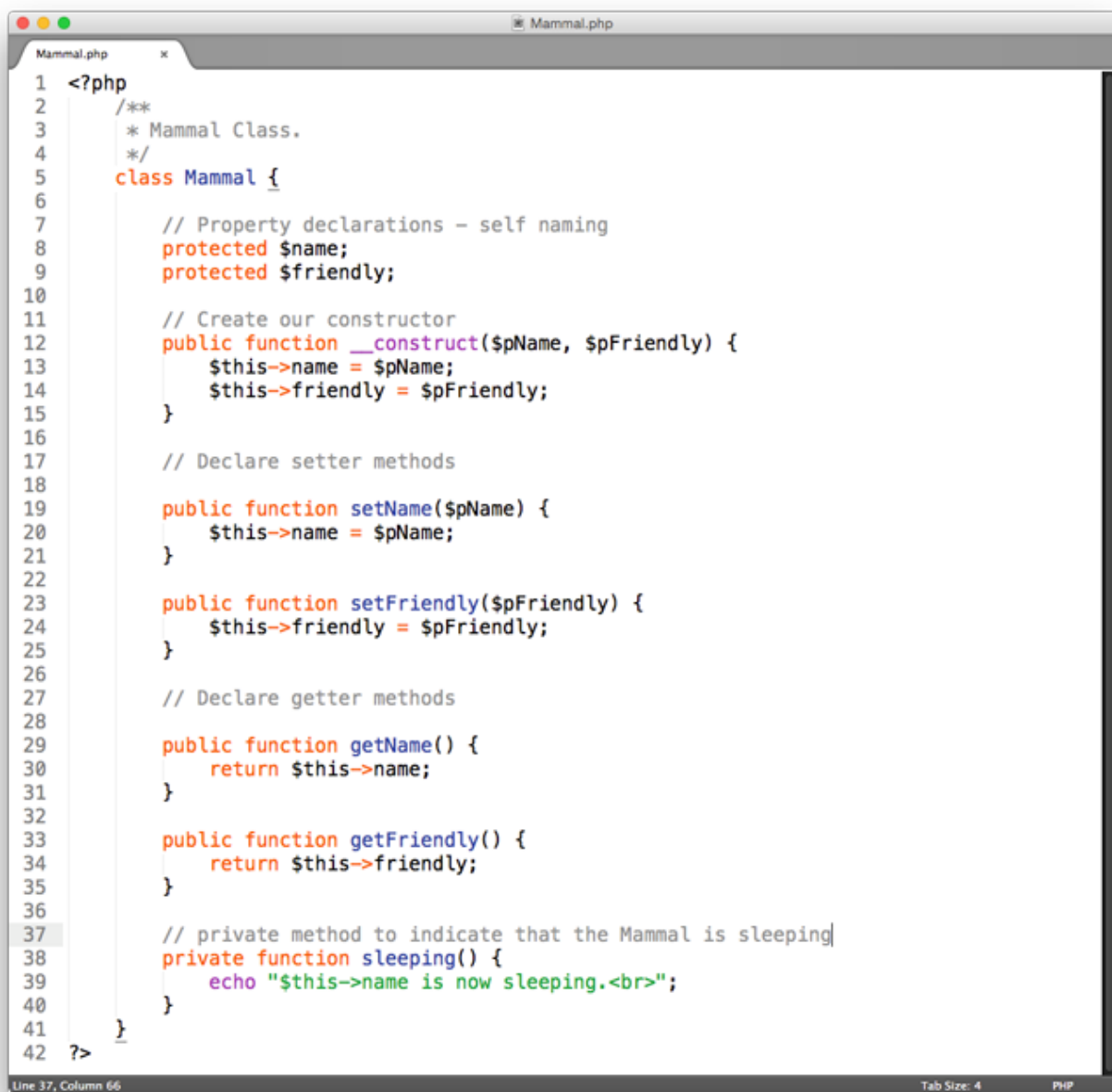
We will now use the keyword public instead of var moving forward when we declare properties and methods.

In some cases, you will want to make your properties protected and use a public method to access the properties instead of providing direct access to the properties by leaving them public.

Let's go into our Mammal.php file and make a couple of changes:

- Set \$name and \$friendly to “protected”
- Add a private function named sleeping()


It will look like this:



```
1 <?php
2 /**
3  * Mammal Class.
4  */
5 class Mammal {
6
7     // Property declarations - self naming
8     protected $name;
9     protected $friendly;
10
11     // Create our constructor
12     public function __construct($pName, $pFriendly) {
13         $this->name = $pName;
14         $this->friendly = $pFriendly;
15     }
16
17     // Declare setter methods
18
19     public function setName($pName) {
20         $this->name = $pName;
21     }
22
23     public function setFriendly($pFriendly) {
24         $this->friendly = $pFriendly;
25     }
26
27     // Declare getter methods
28
29     public function getName() {
30         return $this->name;
31     }
32
33     public function getFriendly() {
34         return $this->friendly;
35     }
36
37     // private method to indicate that the Mammal is sleeping
38     private function sleeping() {
39         echo "$this->name is now sleeping.<br>";
40     }
41 }
42 ?>
```

Line 37, Column 66 Tab Size: 4 PHP

We will then create a Dog object, inspect it's properties with `var_dump()` and then try to use the `sleeping()` method that we declared in our `Mammal.php` file.



```
1 <?php
2
3     require('Dog.php');
4
5     $dog1 = new Dog("Rhett", TRUE, "fast", TRUE);
6
7     var_dump($dog1);
8
9     $dog1->sleeping();
10
11 ?>
```

Here is what we will see when we view it in the browser:

```
object(Dog)#1 (4) { ["barkRate"]=> string(4) "fast" ["snores"]=> bool(true) ["name":protected]=> string(5) "Rhett" ["friendly":protected]=> bool(true) }
Fatal error: Call to private method Mammal::sleeping() from context " in /home/cs602/public_html/access_modifiers.php on line 8
```

You can see that it is telling us that name and friendly are protected now and that we have an error when trying to call the private method `Mammal::sleeping()` from within the `access_modifiers.php` file because we are trying to call this outside of the `Mammal` class.

### Abstract and Final

There are two modifiers that are similar to access modifiers that we have not discussed yet: `Final` and `Abstract`.

- `Final`
  - This keyword can be used to declare that a function or class cannot be overridden by a subclass.
- `Abstract`
  - This keyword is used when you want to prohibit a class or function from being created directly. It must be extended in order to be utilized.

Keep both of those in mind as we will be using them fairly extensively in future sections.

## Static Properties and Methods

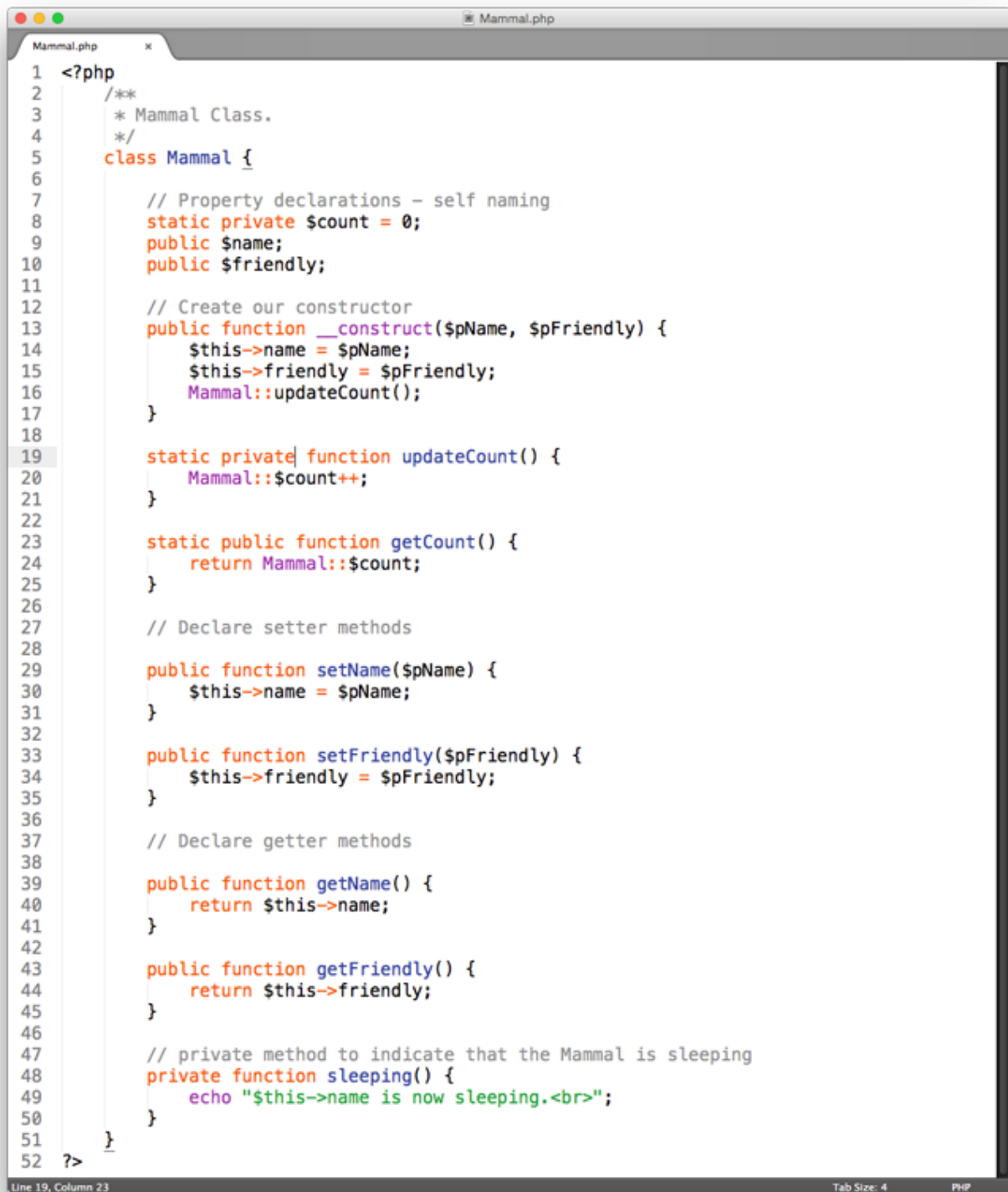
We can define class properties and methods that can be used without instantiating an object first. These are known as static properties and methods. This can be done by putting the static keyword before the visibility modifier (like public).

We access them by using the double colon operator, ::

So the static property belongs to the class, not the object.

A static method is a method that has no need to access any other part of the class. So you can't use \$this inside of a static method.

Let's modify our Mammal class:



```

1  <?php
2      /**
3       * Mammal Class.
4       */
5      class Mammal {
6
7          // Property declarations - self naming
8          static private $count = 0;
9          public $name;
10         public $friendly;
11
12         // Create our constructor
13         public function __construct($pName, $pFriendly) {
14             $this->name = $pName;
15             $this->friendly = $pFriendly;
16             Mammal::updateCount();
17         }
18
19         static private function updateCount() {
20             Mammal::$count++;
21         }
22
23         static public function getCount() {
24             return Mammal::$count;
25         }
26
27         // Declare setter methods
28
29         public function setName($pName) {
30             $this->name = $pName;
31         }
32
33         public function setFriendly($pFriendly) {
34             $this->friendly = $pFriendly;
35         }
36
37         // Declare getter methods
38
39         public function getName() {
40             return $this->name;
41         }
42
43         public function getFriendly() {
44             return $this->friendly;
45         }
46
47         // private method to indicate that the Mammal is sleeping
48         private function sleeping() {
49             echo "$this->name is now sleeping.<br>";
50         }
51     }
52  ?>

```

Line 19, Column 23      Tab Size: 4      PHP

We've added a few things here, first is a static property:

```
static public $count = 0;
```

And two static method declarations:

```
static protected function updateCount() {
    Mammal::$count++;
}
```

```

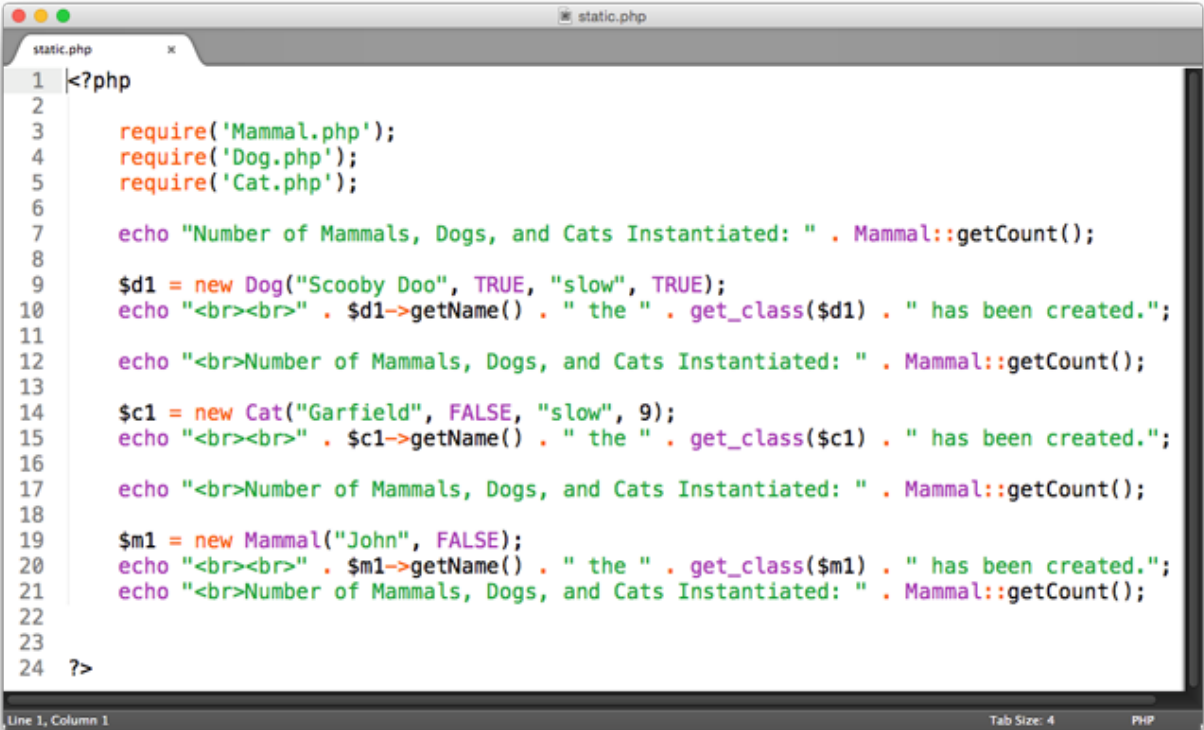
    }

    static public function getCount() {
    return Mammal::$count;
    }

```

You may have also noticed that we changed the access modifiers for \$count and updateCount() to private.

Take a look at this:



```

1 <?php
2
3     require('Mammal.php');
4     require('Dog.php');
5     require('Cat.php');
6
7     echo "Number of Mammals, Dogs, and Cats Instantiated: " . Mammal::getCount();
8
9     $d1 = new Dog("Scooby Doo", TRUE, "slow", TRUE);
10    echo "<br><br>" . $d1->getName() . " the " . get_class($d1) . " has been created.";
11
12    echo "<br>Number of Mammals, Dogs, and Cats Instantiated: " . Mammal::getCount();
13
14    $c1 = new Cat("Garfield", FALSE, "slow", 9);
15    echo "<br><br>" . $c1->getName() . " the " . get_class($c1) . " has been created.";
16
17    echo "<br>Number of Mammals, Dogs, and Cats Instantiated: " . Mammal::getCount();
18
19    $m1 = new Mammal("John", FALSE);
20    echo "<br><br>" . $m1->getName() . " the " . get_class($m1) . " has been created.";
21    echo "<br>Number of Mammals, Dogs, and Cats Instantiated: " . Mammal::getCount();
22
23
24 ?>

```

Note: Just in case you have been following along and recreating the example code, you should know that we removed the require(Mammal.php); lines from both Dog.php and Cat.php.

In the static.php file listed above, we call Mammal::getCount(); before we create any object. We get 0 as a result because no objects have been created yet. Every time we create an object, we call getCount() again and can see that it is keeping track of all objects created that are Mammals or child objects of Mammal (Dogs and Cats).

## Interfaces

Interfaces allow us to organize our code in a more consistent manner. They specify what methods a class must implement, but don't say how this needs to be done. Interfaces don't actually even contain any code that will run. They

can define method names and arguments, but none of the contents for the method itself. If a class implements an interface, it must implement all of the methods that are defined inside of that interface.

So interfaces are similar to abstract classes, with the key different being that you can add all of the code you need to your methods in an abstract class.

The real point of an interface is to give you some flexibility of requiring your class to implement multiple interfaces (as long as there are no conflicting method names), but not be forced into multiple inheritance.

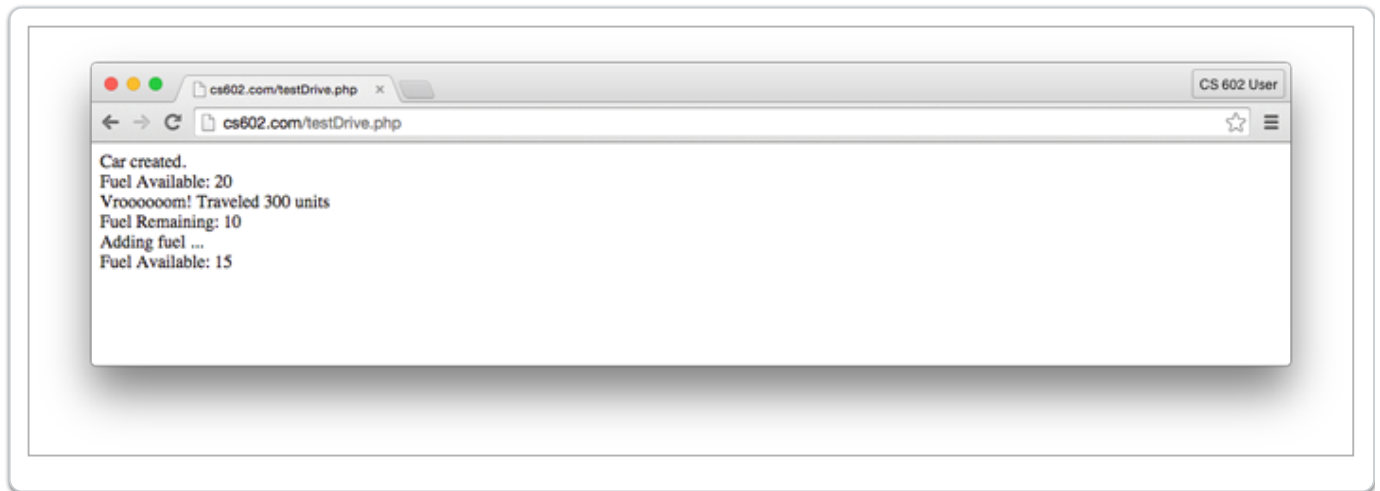
Here is an example:



```
testDrive.php
1 <?php
2
3 // Vehicle.php
4
5 interface Vehicle {
6     public function drive($distance);
7
8     public function addFuel($amount);
9
10    public function readFuel();
11 }
12
13 // Car.php
14
15 class Car implements Vehicle {
16     public $fuelAmount;
17
18     function __construct() {
19         $this->fuelAmount = 20;
20         echo "Car created.<br>";
21     }
22
23     public function drive($distance) {
24         echo "Vroooooom! Traveled " . $distance . " units";
25         $this->fuelAmount -= ($distance/30);
26     }
27
28     public function addFuel($amount) {
29         $this->fuelAmount += $amount;
30     }
31
32     public function readFuel() {
33         return $this->fuelAmount;
34     }
35 }
36
37
38 // testDrive.php
39
40 $myCar = new Car();
41 echo "Fuel Available: " . $myCar->readFuel() . "<br>";
42 echo $myCar->drive(300) . "<br>";
43 echo "Fuel Remaining: " . $myCar->readFuel() . "<br>";
44 echo "Adding fuel ... " . $myCar->addFuel(5) . "<br>";
45 echo "Fuel Available: " . $myCar->readFuel() . "<br>";
46
47
48 ?>
```

It's not necessary, but it may be helpful to pretend that the code listed above is split into three separate files as described in the comments above.

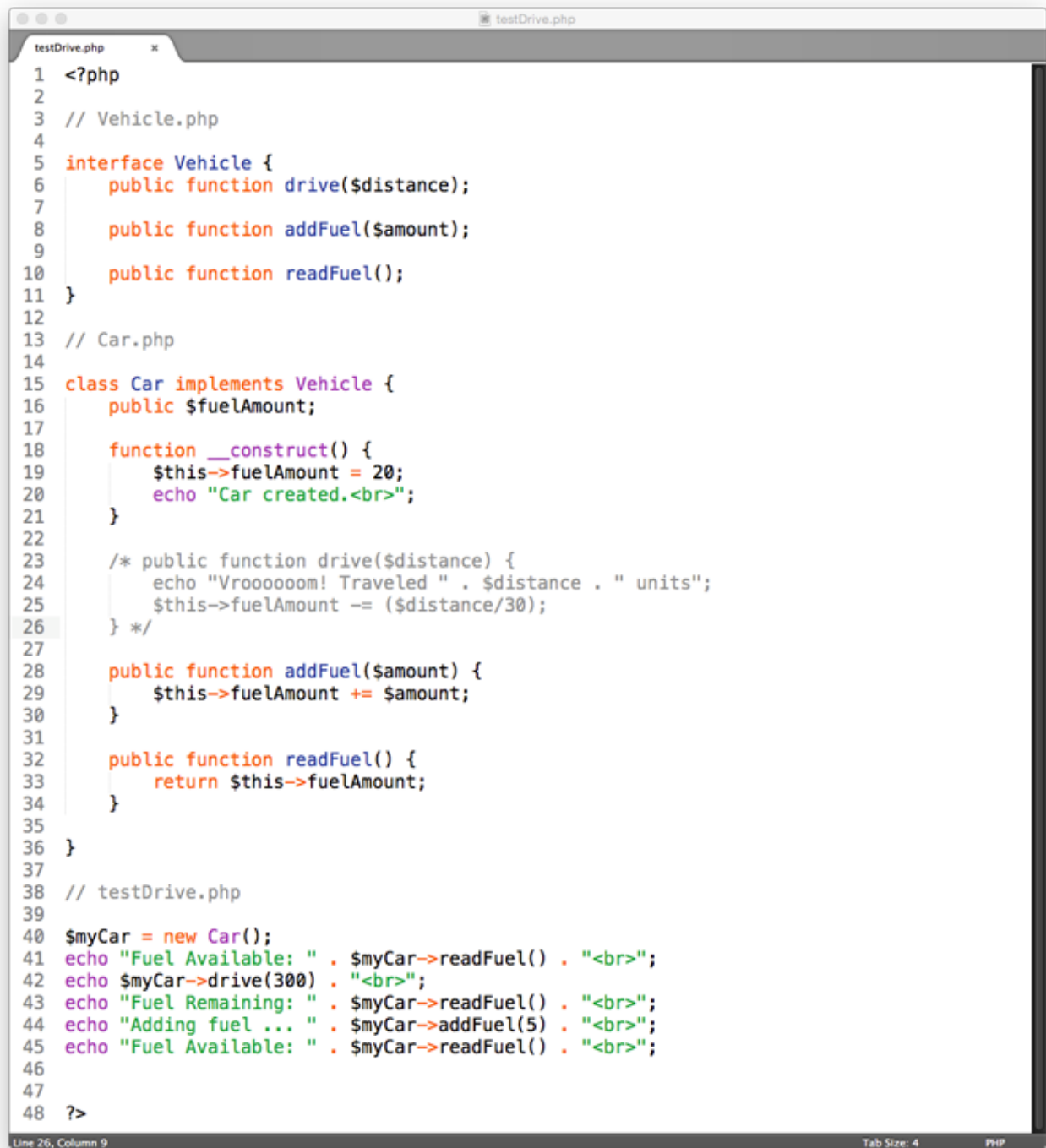
When we run this, this is what we should see in the browser:



Everything looks good because we abided by the “contract” we agreed to when the Car class implemented the Vehicle interface.

Let's look at what happens if we break the contract and don't include all of the methods called for in our Vehicle interface:

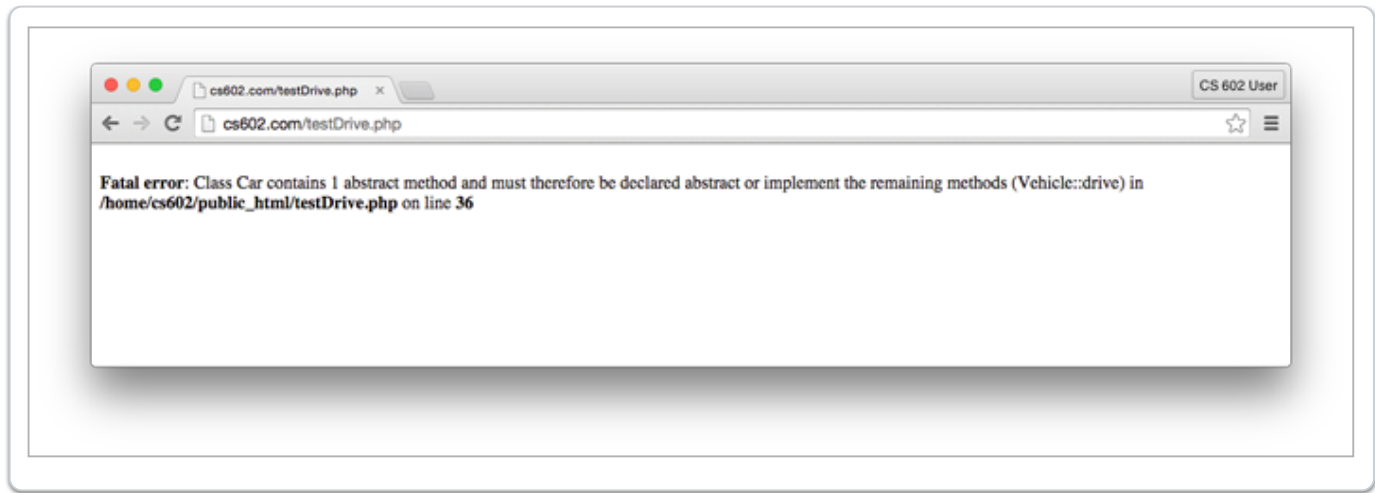




```
1 <?php
2
3 // Vehicle.php
4
5 interface Vehicle {
6     public function drive($distance);
7
8     public function addFuel($amount);
9
10    public function readFuel();
11 }
12
13 // Car.php
14
15 class Car implements Vehicle {
16     public $fuelAmount;
17
18     function __construct() {
19         $this->fuelAmount = 20;
20         echo "Car created.<br>";
21     }
22
23     /* public function drive($distance) {
24         echo "Vroooooom! Traveled " . $distance . " units";
25         $this->fuelAmount -= ($distance/30);
26     } */
27
28     public function addFuel($amount) {
29         $this->fuelAmount += $amount;
30     }
31
32     public function readFuel() {
33         return $this->fuelAmount;
34     }
35 }
36
37 // testDrive.php
38
39 $myCar = new Car();
40 echo "Fuel Available: " . $myCar->readFuel() . "<br>";
41 echo $myCar->drive(300) . "<br>";
42 echo "Fuel Remaining: " . $myCar->readFuel() . "<br>";
43 echo "Adding fuel ... " . $myCar->addFuel(5) . "<br>";
44 echo "Fuel Available: " . $myCar->readFuel() . "<br>";
45
46
47
48 ?>
```

Line 26, Column 9 Tab Size: 4 PHP

You can see that we commented out the drive method in our Car class. Here is what happens when we refresh the page in our browser:



As expected, it complains that we haven't implemented Vehicle's drive() method in our Car class.

## Polymorphism

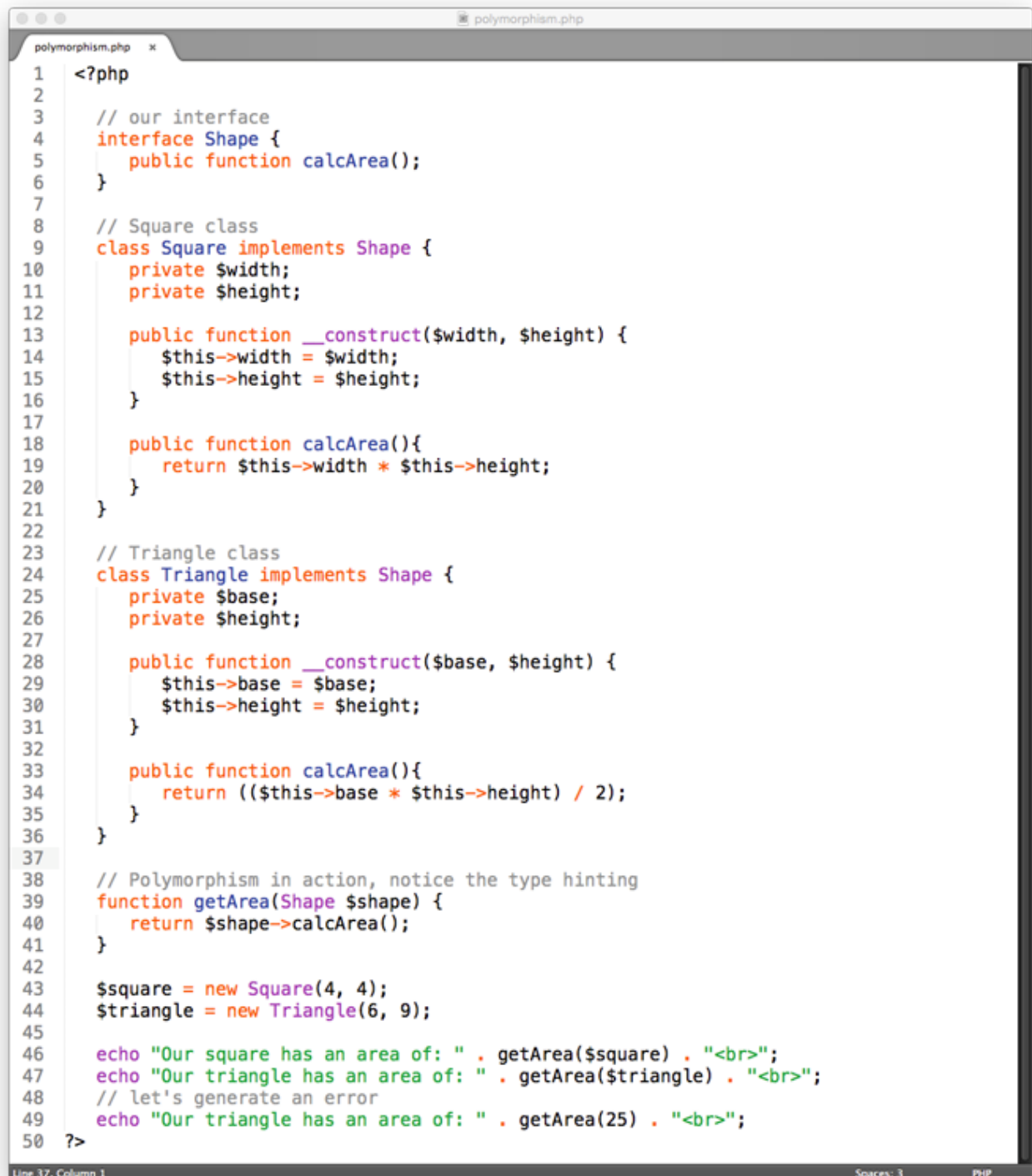
---

The term polymorphism has Greek language roots.

- Poly = many
- Morph = change or form

So you can think of polymorphism as meaning “taking many forms”. In OOP, polymorphism allows for classes to have different functionality while sharing a common interface.

Let's demonstrate this with an example:



```

1  <?php
2
3  // our interface
4  interface Shape {
5      public function calcArea();
6  }
7
8  // Square class
9  class Square implements Shape {
10     private $width;
11     private $height;
12
13     public function __construct($width, $height) {
14         $this->width = $width;
15         $this->height = $height;
16     }
17
18     public function calcArea(){
19         return $this->width * $this->height;
20     }
21 }
22
23 // Triangle class
24 class Triangle implements Shape {
25     private $base;
26     private $height;
27
28     public function __construct($base, $height) {
29         $this->base = $base;
30         $this->height = $height;
31     }
32
33     public function calcArea(){
34         return (($this->base * $this->height) / 2);
35     }
36 }
37
38 // Polymorphism in action, notice the type hinting
39 function getArea(Shape $shape) {
40     return $shape->calcArea();
41 }
42
43 $square = new Square(4, 4);
44 $triangle = new Triangle(6, 9);
45
46 echo "Our square has an area of: " . getArea($square) . "<br>";
47 echo "Our triangle has an area of: " . getArea($triangle) . "<br>";
48 // let's generate an error
49 echo "Our triangle has an area of: " . getArea(25) . "<br>";
50 ?>

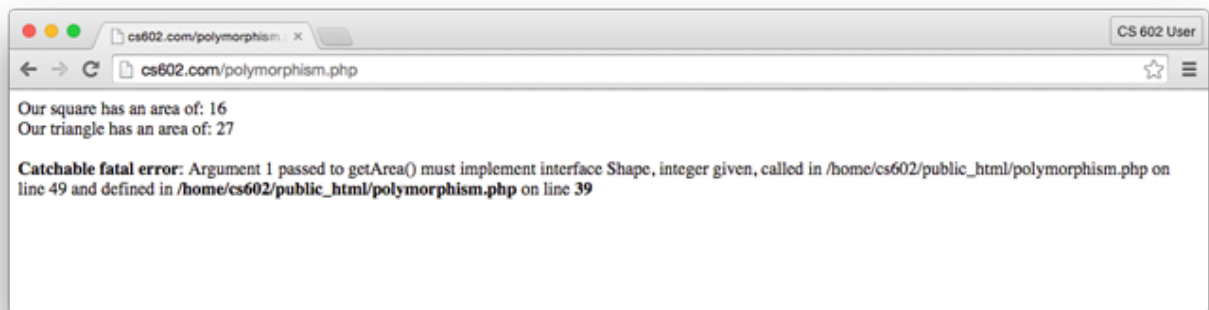
```

Line 37, Column 1      Spaces: 3      PHP

So we can use the `getArea()` method to calculate the area of both a square and a triangle. We could create a class for Circle as well like we did with Square and Triangle and use `getArea()` in the same way to calculate the area for a circle.

A couple of things to notice:

On line 38, we have introduced Type Hinting. With type hinting, we can force parameters to be objects, interfaces, or arrays. Failing to satisfy the type hint will result in an error. On line 49, we provide an integer value to `getArea()` and know this will cause an error due to the type hinting enforcement. Let's see what happens when we load this in our browser:



As you can see on the last try of `getArea(25)`, PHP squawks at us and says it wanted a `Shape` object but it was given an integer.

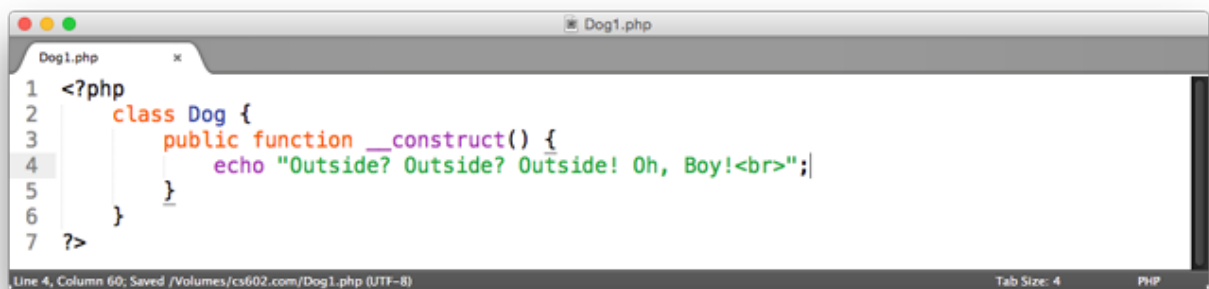
## Namespaces

---

We can't define two classes with the same name in PHP, if we try, we will get an error. We could prefix our class names in order to make them unique, but there is a better way to handle this situation.

Let's take a look at an example where we have three files that each have a `Dog` class declared within them:

Dog1.php



Dog2.php

A screenshot of a code editor window titled 'Dog2.php'. The code defines a class 'Dog' with a public function '\_\_construct()' that echoes 'I'm a dog! Woof, Woof! <br>'. The code is as follows:

```
1 <?php
2     class Dog {
3         public function __construct() {
4             echo "I'm a dog! Woof, Woof! <br>";
5         }
6     }
7 ?>
```

The status bar at the bottom indicates 'Line 7, Column 3', 'Tab Size: 4', and 'PHP'.

Dog3.php

A screenshot of a code editor window titled 'Dog3.php'. The code defines a class 'Dog' with a public function '\_\_construct()' that echoes 'Hey! This looks like the way to the VET! <br>'. The code is as follows:

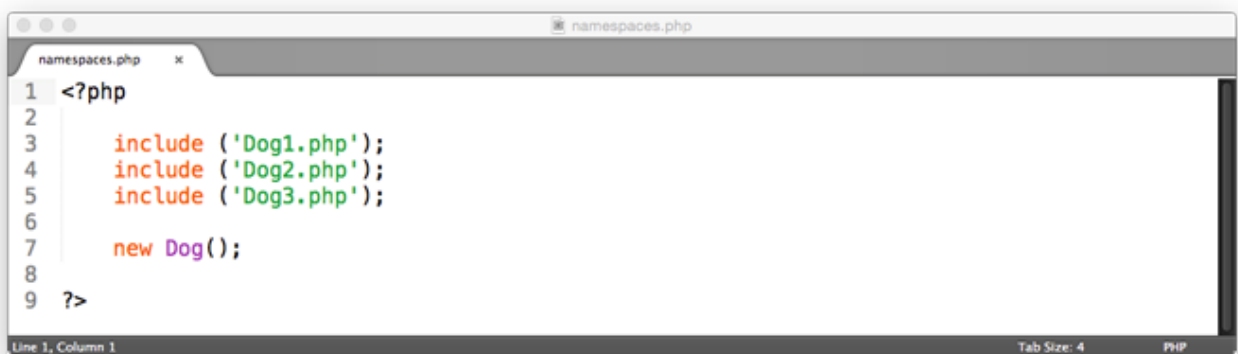
```
1 <?php
2     class Dog {
3         public function __construct() {
4             echo "Hey! This looks like the way to the VET! <br>";
5         }
6     }
7 ?>
```

The status bar at the bottom indicates 'Line 7, Column 3', 'Tab Size: 4', and 'PHP'.

We can see that each file contains a class declaration for Dog, but each one implements it in a slightly different way.

Now let's create another file and include all of the Dog[x].php files and try to create a new Dog object:

namespaces.php

A screenshot of a code editor window titled 'namespaces.php'. The code includes 'Dog1.php', 'Dog2.php', and 'Dog3.php', and then creates a new Dog object. The code is as follows:

```
1 <?php
2
3     include ('Dog1.php');
4     include ('Dog2.php');
5     include ('Dog3.php');
6
7     new Dog();
8
9 ?>
```

The status bar at the bottom indicates 'Line 1, Column 1', 'Tab Size: 4', and 'PHP'.

And now let's see what happens when we load this in our browser:

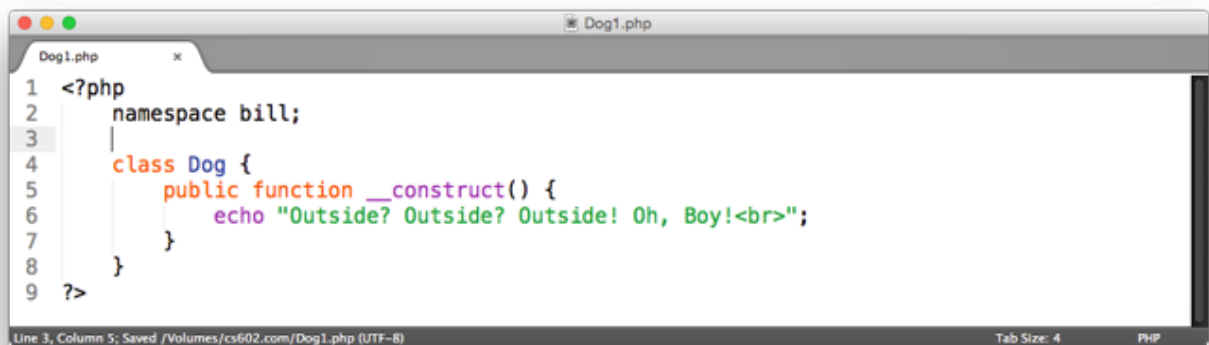


As you may have expected, we get an error. The error is informing us that we tried to redeclare the Dog class again through our second include statement. We never even get to our third include statement because we've already encountered an error.

Luckily we can fix this with namespaces. PHP received namespace support in version 5.3. Namespaces allow us to group classes together into virtual directories. This allows us to avoid overlapping class names in our programs. It is important to note that only classes, interfaces, functions, and constants are affected by namespaces. We can have sub namespaces as well and nest them as many times as we need to based on our use case.

Let's make a few changes to the files we shared above to include namespaces:

Dog1.php



Dog2.php



```
1 <?php
2 namespace john;
3
4 class Dog {
5     public function __construct() {
6         echo "I'm a dog! Woof, Woof! <br>";
7     }
8 }
9 ?>
```

Line 3, Column 5; Saved /Volumes/cs602.com/Dog2.php (UTF-8) Tab Size: 4 PHP

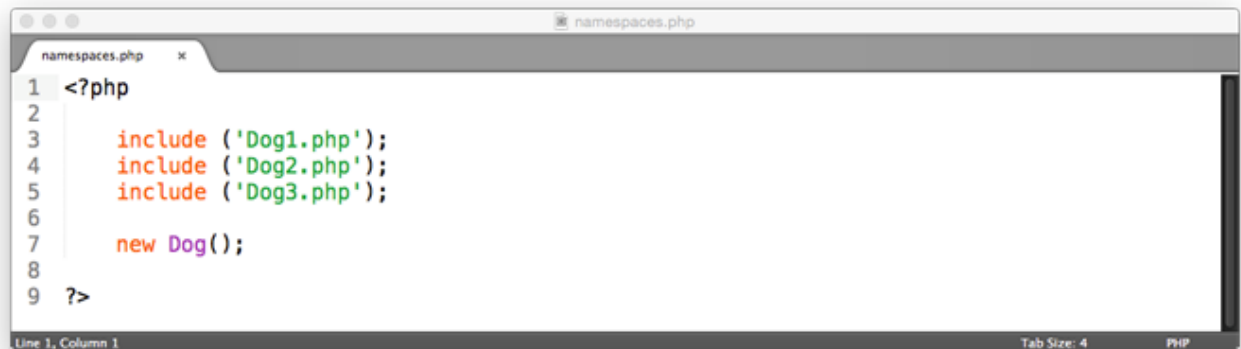
Dog3.php



```
1 <?php
2 namespace jane;
3
4 class Dog {
5     public function __construct() {
6         echo "Hey! This looks like the way to the VET! <br>";
7     }
8 }
9 ?>
```

Line 1, Column 6 Tab Size: 4 PHP

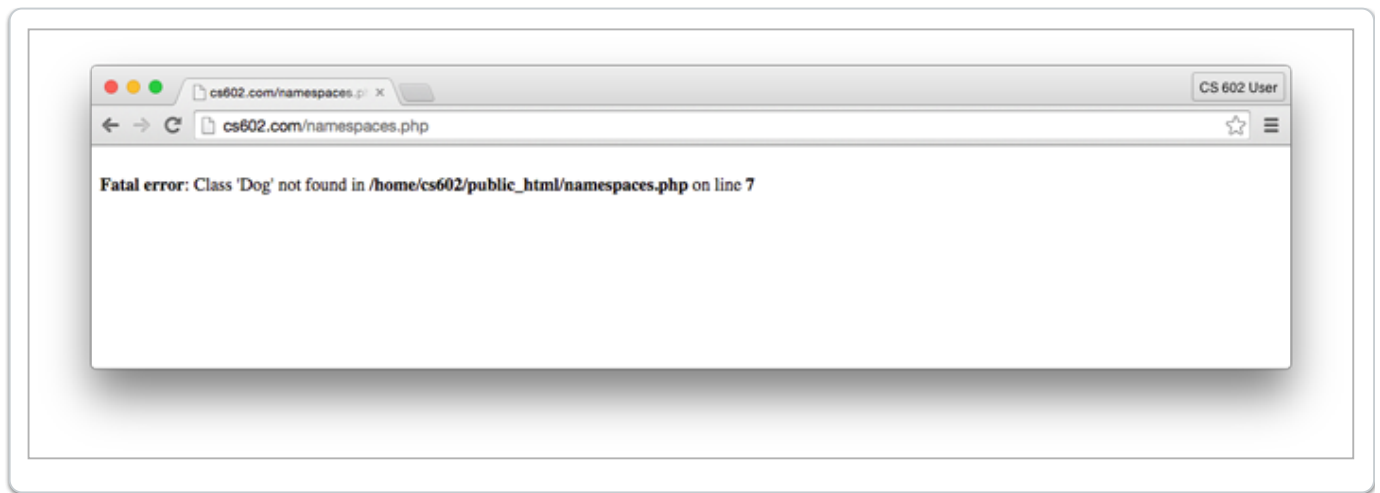
namespaces.php



```
1 <?php
2
3 include ('Dog1.php');
4 include ('Dog2.php');
5 include ('Dog3.php');
6
7 new Dog();
8
9 ?>
```

Line 1, Column 1 Tab Size: 4 PHP

So we have added namespaces to the three files with our Dog class definitions. Let's see what happens when we load namespaces.php in our browser this time:



Our error is different than last time. This time it's telling us that it can't find the class Dog, even though we have three different versions included! It actually can't find the Dog class because PHP has a GLOBAL namespace and that is where it is trying to locate our Dog class(es). We have to let PHP know in what namespace(s) it can find our Dog classes. Let's make a change to namespaces.php:



OK, great. We are making some progress now. You can see that it was able to find the Dog class in the bill namespace. Let's go ahead and create new Dog objects from the john and jane namespaces as well:



Now you can see that we have instantiated an object from each one of our Dog classes that are found in different namespaces and the constructors have echoed the statements found within each of the classes.

So here are some things to keep in mind about namespaces:

- Namespaces must be our first line of code within our PHP files
- We can create sub namespaces, just like we can have subfolders in a folder on our computer's filesystem.
  - namespace TopNS\SubNS
- There are three ways that we can refer to our namespaces:
  - Unqualified - Used to refer to a file in the current namespace
    - \$foo = new Foo();
  - Qualified - Used to refer to a sub namespace in the current namespace
    - \$foo = new SubNS\Foo();
  - Fully qualified - Used to refer to any file in any namespace (like an absolute path)
    - \$foo = new TopNS\SubNS\SubSubNS\Foo();
- It might be cumbersome to continually type out long namespace names, in this case you can do this:



- use `TopNS\SubNS\SubSubNS;`
  - We can even alias it as well: use `TopNS\SubNS\SubSubNS` as `myNS;`
    - Now you can use `myNS` like a variable that points to the true path of the namespace
- If we are currently inside of a custom namespace and need to access a class in the global namespace, just prefix the name of the global class with a backslash ( \ )

## Autoloading

---

Let's create a new PHP file and create a new Cat object:

```
.
```

Here is what we get in the browser:

```
.
```

Oops! It looks like we forgot to include our `Cat.php` file. Let's go ahead and do that now:

```
.
```

And let's try this again in the browser:

```
.
```

Rats! We included `Cat.php`, but we don't have `Mammal.php` included in the `Cat.php` file or our current file, let's fix this:

```
.
```

And let's cross our fingers and hope this works now. It should, right?

```
.
```

Woo-hoo! It does work now. It can be a bit tedious including all of the class files that we need for our projects. When our projects get larger, we might have to include dozens or more of these include or require statements in order to ensure that we have all of our class declaration files available to us. Luckily, there is a better way: autoloading.

Let's take a look at an example:

```
.
```

And let's see how this turns out in our browser:

Wow! That's pretty cool. So what is going on here?

You can see that we created a function and we chose to name it `my_autoloader()`. The combination of this function and the `spl_autoload_register()` function allows us to find all of the class files in the current directory and include them automatically as they are needed when we try to instantiate a new object. That might be a bit confusing, let's explain this another way:

When PHP sees that we are trying to instantiate a class, it passes the name of the class as a string to `spl_autoload_register()`. This allows us to pick up the class name as a variable and use it as an include for the appropriate class file. Because we named our PHP class files with the same name as the class names themselves, all we have to do is append `.php` to the name of the class and let PHP include that file. If our class files were in another directory, we would just need to prepend the path to the string we have created on line 3.

## PHP Data Objects (PDO)

---

In the last module, we explained how to access a MySQL database with PHP using the MySQLi extension. We chose to use MySQLi with procedural statements because we had not yet introduced the concepts of object oriented programming. We shared that there were two other ways access a MySQL database with PHP:

- MySQLi - object oriented
- PHP Data Objects (PDO)

We are now going to show how we can use PDO to interact with our databases. You can learn more about MySQLi with OOP methods by referring to your textbook.

PDO stands for PHP Data Objects. It is a database access layer that provides a uniform method of access to multiple types of database management systems. With PDO, you could create your application using one database type and easily switch to a different database type later on down the road. PDO currently supports the following databases:

- CUBRID
- MS SQL Server
- Firebird
- IBM
- Informix
- MySQL
- Oracle
- ODBC and DB2
- PostgreSQL
- SQLite
- 4D

Let's look at one example: Connecting to a MySQL database with PDO and inserting a record into a table. You can learn how to retrieve, update, delete, etc. records with PDO by referring to the code examples in your textbook.

Line 4: We define the data source as MySQL and indicate it resides on the local machine. We also specify the name of the database we want to connect to.

Lines 5-6: Credentials for accessing the database

Line 7: Options we can send along in our connection string. In this case, we are setting up how we want to handle errors.

We then have code wrapped into a try/catch block. This is a way to process any errors that we encounter. If an error occurs, it is known to be "thrown". In this case, we can try to catch it. Whatever is contained within try is there to facilitate catching these potential errors. The catch block is used to actually deal with the error.

In this case we are trying to create a new PDO object in order to connect to our database. We then create a prepared statement for security purposes and then try to execute a query that inserts the records into the database. Again, we are trying to do a few different things and there are a numerous ways that we can experience errors during this process. So all of that code is wrapped in the try block in order for it to watch for errors with any of the actions we are attempting within it. If an error does occur, only then will we execute the code found in our catch block.

## Design Pattern: MVC

---

Let's build a small web application. For our example, we are going to create a program that allows a user to enter a stock symbol and receive the price of the stock along with some other information. We will check the stock symbol that is requested against our data source to see if it can be found and return the results to the user.

For a program of the size that we are going to create, we could put the presentation code, the business logic code, and the database code into one single PHP file. However, this program may scale significantly in the future and we know that we'll have to maintain and improve upon this code in time. We are going to assume that the program will eventually evolve into a complex project. Let's also assume that we will be working in a team environment where individuals have specific tasks such as user interface designer, programmer, and database expert.

The best way to approach a complex problem is to to break it down into small, manageable parts. This allows us to solve several smaller problems rather than trying to tackle one large one, which can often be intimidating. When we start breaking things into smaller pieces, it is a good idea to follow the separation of concerns principle (SoC). SoC is a design principle for separating our program into distinct sections that address a specific concern. We can then assign our team members to handle a specific concern that aligns with their area of responsibility and expertise. We are going to use an architectural design pattern known as MVC to accomplish this within our project.

Note: A design pattern is a general reusable solution to a commonly occurring problem in software design.

## MVC

MVC stands for Model View Controller. It is a very popular design pattern used in web applications. Each part of the MVC pattern has a specific role and purpose:

- **Model**
  - The model represents the data on which the application operates. It stores information and responds to requests for this information. Normally, this is done with a database, but we will be using a more primitive data store for our example.
- **View**
  - The view renders the data from the model (via the controller) into a format that is suitable for user interaction. It essentially manages the display of information and provides the user interface.
- **Controller**
  - The controller facilitates the interaction between the view and the model. It responds to events, typically via user interactions. The controller is also responsible for the business logic processing.

An interesting point about MVC is that it can bring complexity to small projects, but it can reduce complexity in large projects. The example we share below could certainly do without MVC in its current state, but because we know that the intent is to grow this project in the future, starting off with MVC is a wise choice.

**Note:** *The example we are showing is a very simple implementation of MVC. This is done intentionally for instructional purposes. We will be sharing an overview of more complex and powerful MVC implementations in the next section.*

We will start off by showing the structure for our application. You will see that each MVC component is contained within its own folder:



Now let's examine the source code for each file:



The index.php file is very straightforward. All it does is include the Controller.php file, instantiate a new controller object, and call the main function on the new Controller object.



Our Controller.php file is where things really start to happen. It includes the Model.php file and then creates a new Model object. The Controller object has a main method that does a few key tasks (business logic):

- It receives the user input as `$_POST['stock']`. Because this is raw user input, it runs a few of PHP's built in functions to sanitize and clean up the information for security purposes.
  - `strip_tags()` - removes HTML and PHP tags from the input string.
  - `trim()` - removes any leading and trailing whitespace in the input string
  - `strtoupper()` - converts the input string into all caps. This allows the user to enter the stock symbol in any case without having to worry about entering it in as all caps to ensure it is an exact match to the stocks we

have in our Model.

- We then check to see if the user actually submitted information in the form and check to see if it is not blank or empty after cleaning it up. If there is data to process, we submit it to the Model's `getStock()` method and include the appropriate view page for displaying the result. If the form has not been submitted or if the data is `NULL`, we simply present a view of the form again.

.

The `Model.php` file includes the `Stock.php` file. We then have a function called `getStockList()` that returns our stock information that is stored in an array. This would normally contain the database code for connecting to the database and querying it for a match against the stock that was requested by the user. We also have a `getStock()` method that returns the matched stock if it is found in our array of stocks.

.

`Stock.php` is a simple class that allows us to create a stock object that contains the symbol, price, and company name.

.

`viewstock.php` is one of our view files. This file is responsible for formatting the results page based on the information that the Model returns to our Controller. The Controller passed this along to the `viewstock.php` for output processing. If it finds that `$stock` contains a value (which is determined by the fact that `$stock` is not `NULL`), it presents the appropriate information on the user's requested stock. If the stock was not found/matched, it displays a message to the user informing them to try again and gives them a hint as to what stocks might be available. We have a button at the bottom of the page that will take us back to the form so that we can try our search again.

.

`form.php` is very simple. It asks the user to input a stock symbol and then submits the data to itself using `$_SERVER['PHP_SELF']`. This may sound confusing, but instead of submitting this to a specific form processing script, it submits it back to itself because it has its own form processing logic built in to it. Remember that `form.php` was included in the Controller, so we are actually submitting the form to the Controller, which in turn passes it along to the Model.

Let's see how this all works out:

The screenshot below simply shows what happens when we visit `index.php`. We see a form ready for us to enter and submit our stock symbol.

.

We enter the stock symbol 'GE' and click on submit.



We are informed that the symbol was not found, but we are given a hint as to what symbols we can try. So we click on the Go Back button to try again.



We are then presented with the form and enter 'MSFT' for our choice of symbol.



We receive the confirmation that it found our stock symbol: 'MSFT' and displays the stock price and company name.



If we click on the Go Back button again, it also takes us back to the index.php page if we want to enter another stock symbol.



And that's it. We've created a small web application, while not perfect, it does a pretty good job at implementing a basic MVC approach.

## Think for a Moment

So let's consider a few things:

What would we need to do if we wanted to swap our the array of stocks in favor of an actual database?

The correct answer is: edit the Model.php file. We would just need to change the code to reference a database query rather than containing an array and it's associated logic.

If we would have designed this as a single file, or even multiple files without an MVC approach, we would be changing database access code, business logic code, and our view code. This could result in introducing errors and would also take more time to locate all of the information that needed to be changed in order to accommodate this new approach to storing and retrieving information. Additionally, all three of our team members would be working on the same file, with some code that they all may not be very familiar with. With the MVC approach, only the person responsible for the Model (the database expert), would need to be involved in the revision.

## MVC Frameworks for PHP

---

Instead of spending your time creating your own MVC implementation, you should consider using an existing PHP MVC framework. A framework is a set of files that is designed to help develop programs while alleviating much of the overhead associated with the process of building an application. You can find many PHP MVC frameworks licensed under an open source model. There are numerous benefits to using a framework:

- They make a programmer's life easier by taking care of common and mundane tasks.
- The code is proven to work and has already been debugged.
- There is a large community of developers who use specific frameworks and many are willing to help other developers.
- Helps to organize your code.
- Many provided utilities and libraries.
- Security (this can be good in most cases, but also presents some negative side effects).
- Faster development times.

There are also some disadvantages to using a framework:

- You will learn to use the framework, but might lose out on the experience of writing your own code.
- Learning a framework takes time.
- Security - you are using common and popular code.
- It may not be appropriate for small projects.

Here are some of the more popular MVC frameworks for PHP with brief descriptions from their respective web sites:

- [CodeIgniter](#)
  - CodeIgniter is a powerful PHP framework with a very small footprint, built for developers who need a simple and elegant toolkit to create full-featured web applications.
- [CakePHP](#)
  - CakePHP makes building web applications simpler, faster and require less code. CakePHP is a modern PHP 5.4+ framework with a flexible Database access layer and a powerful scaffolding system that makes building both small and complex systems a breeze.
- [Laravel](#)
  - The PHP Framework For Web Artisans. Elegant applications delivered at warp speed. Expressive, beautiful syntax. Tailored for your team. Modern toolkit. Pinch of magic.
- [Symfony](#)
  - Symfony is a set of PHP Components, a Web Application framework, a Philosophy, and a Community — all working together in harmony.
- [Zend Framework](#)
  - Zend Framework 2 is an open source framework for developing web applications and services using PHP 5.3+. Zend Framework 2 uses 100% object-oriented code and utilizes most of the new features of PHP 5.3, namely namespaces, late static binding, lambda functions and closures.
- [Yii](#)
  - Yii is a high-performance PHP framework best for developing Web 2.0 applications. Yii comes with rich features: MVC, DAO/ActiveRecord, I18N/L10N, caching, authentication and role-based access control, scaffolding, testing, etc. It can reduce your development time significantly.

Due to the time it takes to learn the intricacies of a framework, we won't be covering them in this course but we wanted to introduce the availability of these options for your future projects.

## Summary

---

We have covered a great deal of advanced PHP in this lecture. Hopefully you now have an understanding of OOP and some of the advanced OOP features. You should now be able to use PDO to connect your PHP scripts to various database types and understand how to use the MVC framework to better organize your larger web development projects.

## Bibliography

---

Abeyasinghe, S. (2009). MVC and Software Teams. In *PHP Team Development* (pp. 29-43). Birmingham, UK: Packt Pub.

British Columbia Institute of Technology. (2015). *CodeIgniter Home Page*. Retrieved July 27, 2015, from <http://www.codeigniter.com/>

Cake Software Foundation, Inc. (2015). *CakePHP Home Page*. Retrieved July 27, 2015, from <http://cakephp.org/>

Fabien Potencier. (2015). *Symfony Home Page*. Retrieved July 27, 2015, from <https://symfony.com/>

Murach, J., & Harris, R. (2014). Professional PHP for working with MySQL. In *Murach's PHP and MySQL* (2nd ed., pp. 614-631). Fresno, CA: Mike Murach and Associates.

Taylor Otwell. (2015). *Laravel Home Page*. Retrieved July 27, 2015, from <http://laravel.com/>

The PHP Group. (2015). *PHP Manual*. Retrieved June 4, 2015, from <http://php.net/manual/en/index.php>

Yii Software LLC. (2015). *yiiframework Home Page*. Retrieved July 27, 2015, from <http://www.yiiframework.com/>

Zend Technologies Ltd. (2015). *Zend Framework 2 Home Page*. Retrieved July 27, 2015, from <http://framework.zend.com/>

## Caching, Registry, Routing

## Learning Objectives

---



By reading the lectures and textbook, participating in the discussions, and completing the assignments, you will be able to do the following:

- Cache MySQL database query results with PHP.
- Implement the Registry design pattern in PHP.
- Implement URL Routing in PHP.

## Introduction

---

We are going to cover some advanced features of PHP in this lecture. You will see some primitive examples of Caching, URL Routing, and a Registry object. These examples have been carefully designed to strip away as much complexity as possible while still allowing you to understand the intended functionality. You should be able to take these examples and add additional functionality in order to suit your needs or to simply understand how certain tasks are accomplished in more complex situations.

## Caching

---

A cache is a temporary area where frequently accessed data can be stored for rapid access. The benefits of using a cache can include reducing the number of database queries in order to improve performance, reduce the number of requests made to external services, reduce filesystem access, reduce processing time, etc. The overall idea is that database driven websites can naturally have a lot of different processes involved that take a bit of time and system resources. We can provide static content that reduces the amount of time and resources needed to provide the requested information to our users. This improves the responsiveness and speed of the page while reducing the load on the server.

There are various approaches to caching and we will cover a simple one in this section. There are certainly more complex and efficient methods available, and many PHP frameworks will perform caching functions for you. PHP also has the Alternative PHP Cache (APC) library that handles many different tasks related to caching and optimizing your PHP code. Learn more about [Alternative PHP Cache](#).

We will be utilizing Output Buffering and some filesystem access commands to capture our HTML and store it in a variable (and then in a file). We will then send it to the browser in once piece after our script has finished processing rather than in individual pieces as the script is running (by default).

The code samples listed below are commented to describe what is being done in each area of the file.

Let's take a look:



In our first code example shown above, we create a file that will contain our cached data and also set an expiration time for how long this version of the cached data will be used. In this case, we want the cached data to last for only 30 seconds, after which time we will create a new cache file after we receive another request for the information. If someone visits the page within 30 seconds of an initial request that created the cached file, they will receive cached information rather than having that information pulled from the database again. This can help reduce load on our MySQL server and speed up overall performance. This is very important when we receive a large number of requests or the requests involve a large number of records resulting in increased processing time. If the user receives cached data, the rest of the page logic is not processed.

```
.
```

Above: Our database connection file that is included when we are not serving cached data.

```
.
```

Above: Our SQL query to request the information from the database along with the logic and formatting to present it on the screen. This is also only called when we are not serving cached data.

```
.
```

Above: Viewing the page initially, page data is loaded from the database.

```
.
```

Above: Viewing the page within 30 seconds of the initial viewing. You can see that the data shown is cached.

```
.
```

Above: Viewing the page again after the cache file has expired. Here we have data that has been pulled from the database.

```
.
```

Above: If we view the page again within 30 seconds, we can see that we are getting a new set of cached data based on the updated time stamp.

## Registry Pattern

---

A Registry is a container for objects to be reused and accessible throughout your project in order to save resources. We can also avoid using global variables using this method.

The example we will be using to explain the registry pattern will actually be modeled against an actual bridal registry. A bridal registry is a type of gift registry that consists of a wish list of items that a recipient can compile and share with friends who can locate their items on the registry and then make a purchase. Our PHP object registry is going to be used to store and locate the objects contained within it all across our program.

```
.
```

Above: Our Product class implementation. It is a simplified version that only contains properties and a constructor. Setters and getters were excluded intentionally as they will not be used in this example.

```
.
```

Above: We create our Registry class. It has static properties and functions so it does not need to be instantiated. We intentionally make the constructor (and \_\_clone) inaccessible to prevent them from being called. This is similar to a Singleton (a design pattern to prevent the creation of a second instance of an object)

```
.
```

Above: We create three new Product objects and add them to our Registry.

```
.
```

Above: Our index file includes the previous three files and displays the objects using var\_dump() and the Registry's get function.

An example of a real use of the Registry pattern could be the storing of a database connection object that could be reused across your program. We encourage you to do further research on the Registry Pattern. You'll find some good uses for it and some duly noted criticisms as well.

## URL Routing

---

Before we discuss URL routing, it will be helpful for you to have an understanding of what a rewrite engine does.

### Rewrite Engine

*"A rewrite engine is software located in a Web application framework running on a Web server that modifies a web URL's appearance. Many framework users have come to refer to this feature as a "Router". This modification is called URL rewriting. Rewritten URLs (sometimes known as short, pretty or fancy URLs, search engine friendly – SEF URLs, or slugs) are used to provide shorter and more relevant-looking links to web pages. The technique adds a layer of abstraction between the files used to generate a web page and the URL that is presented to the outside world."* –

Source: [Rewrite Engine](#).

So this allows us to take a URL such as:

**`http://example.com/wiki/index.php?title=Page_title`**

And rewrite it as:

**`http://example.com/wiki/Page_title`**

The later example is cleaner and friendlier to search engines.

We can then take the cleaned up version of the URL and break it up into individual components. These components will be used to determine what script (likely to be a Controller and an action) to call when that URL is visited. This process is known as Routing.

In order to allow this to happen, you need to enable your web servers rewrite module. Additionally, a .htaccess file must be placed in the root directory of your website with the following information inside of it:

```
RewriteEngine on
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ index.php/$1 [QSA,L]
```

## Routing Code Example

```
-
```

Above: We create our Router class. Everything here is pretty straight forward. Our init() method does most of the work. It checks an array key against a portion of the URI and returns the appropriate content (the array value).

```
-
```

Above: This is our array of predefined segments represented as array keys that are compared against a portion of the URI segments. If there is a match, we will display the array element's value.

```
-
```

Above: Our index.php file. It includes the files we are dependent upon and instantiates a new Router object and calls it's init() function.

```
-
```

Above: Viewing the index.php page.

```
-
```

Above: Appending "cs602" at the end of the URL and we are greeted with the appropriate message.

Above: Appending an unmatched string at the end of the URL and we receive a message that it could not be found (there was no match in our array of routes).

The example provided above is very primitive and is intended to be used solely for instructional purposes. It is not very robust, but it should be simple enough to understand. In reality, you will most likely use a framework to do your custom URL routing.

## Summary

---

In this module, we covered some additional advanced features of PHP and some creative ways to add functionality to our projects. These examples should help you understand what is happening when you implement MVC frameworks that provide similar, yet more complex features. This will help you to understand how the concepts work in case you need to make customizations to the framework components.

## Bibliography

---

Wilhelm, A. (2015). *The Registry Pattern and PHP*. Retrieved July 27, 2015, from <https://avedo.net/101/the-registry-pattern-and-php>

Wikimedia Foundation, Inc. (2014). *Rewrite engine*. Retrieved July 27, 2015, from [https://en.wikipedia.org/wiki/Rewrite\\_engine](https://en.wikipedia.org/wiki/Rewrite_engine)

**Boston University Metropolitan College**