Welcome

Welcome to PHP, Object-Orient Programing. We are excited to teach you object-oriented programming and PHP. In this course, we'll start by defining classes with properties and methods and then create instances from those classes. We will learn how to use class inheritance to share, extend, and override object behaviors. We will use visibility modifiers to control access to the code inside a class. We will discover how to define behaviors on a class as a whole, rather than on a specific object.

We will uncover the automatic behaviors of several magic methods and learn to write our own. And finally, we'll get some real world experience by adding OOP code to website project. This is an intermediate course, which will assume that you already know the basics of PHP. If you need to review your skills, PHP Essential Training and PHP with MySQL Essential Training parts one and two will prepare you with everything you need to know for this course. Once you're ready, let's get started learning, PHP, object-oriented programming.

Overview and project setup

What is OOP?

Let's begin by talking about what is OOP. OOP is an abbreviation for object-oriented programming. Object-oriented programming is a type of programming where we define data structures, those are the objects, which can hold both values and functions. PHP is a scripting language or a procedural language. If you've been using PHP already, then you've been setting variables and you've been defining functions. Working with PHP in this way is considered procedural, not object-oriented.

Our code changes are rather linear as we move down the page of code. We set values, we call functions, we work with the results. We're calling procedures as we move down the page. PHP is not considered an object-oriented language. Instead, it's a scripting language with object-oriented features built into it. Features that were added in PHP five. Now, the fact that PHP isn't fundamentally object-oriented isn't a problem, it just means that we have a choice. We can either work with PHP in a procedural way or in an object-oriented way.

So, how do you know if you need object-oriented programming? Well, if scripting and procedures are all you want PHP to do, then you may not benefit from it. But, if you're asking PHP to do more complex work, then OOP may be exactly what you need to manage that complexity. Here's my rule of thumb. For a simple site, object-oriented programming adds unnecessary complexity, but for a complex site, object-oriented programming adds necessary simplicity. And database-driven sites generally have enough complexity that they will benefit from object-oriented programming.

So, what exactly are these objects that we're talking about? Well, in the simplist sense, objects are simply grouping code together by a common theme. So, if you think back to some of the work we've done in the past where we were working with forms that people are submitting, we write functions that relate to those forms. We could take all of those functions and roll them up into a form object and then our code would be grouped together by a common theme and we'd be able to find those functions all in one place. We can also use software objects to create abstractions for concepts in our code as if they were real-world objects.

And this is probably a more common use than just grouping them together by some basic theme. If we have a website that's selling products, then we would probably have a customer object and that would be an abstraction for the real-world customer object. We would probably also have a product object that we're selling the customer; a shopping cart object; a credit card object; and so on. Objects don't have to have real-world analogs either. If we have a blog, then we might have a blog post object and if users are able to comment on the blog post, then we would probably have a user object and a comment object as well.

Remember, object-oriented programming is about creating data structures which hold values and functions. We call those properties and methods. These are the fundamental elements for defining and working with objects. Properties are just variables inside the object. They define the characteristics of a particular object. When we think of real-world objects, we naturally think of their properties. The dress is green; green is a property of the dress. The table is four feet long and two feet wide. The house has three bedrooms and two bathrooms. In code, object properties might be a student object which has a first name and a last name; or a book object which has a title, an author, a year, and the number of pages. These are all just variables that we set to define the properties of a particular object. Methods are just like our regular procedural functions, but they're stored inside the object so that the object knows how to do something. For example, a shopping cart object could have a method called total, which would inspect the properties of all the product objects in the shopping cart, get their prices, and return back the total amount.

This is where objects get a lot of their power because we can add methods to teach them how to perform tasks, to examine properties, and to call other methods. Our shopping cart object might have an add to cart method which performs several tasks: it checks product inventory, then it adds the product to the cart, then it increments the cart quantity, and perhaps even removes an item from the inventory. That's a lot of complexity that's being nicely packaged up into a single, simple function. So objects are very useful, especially for managing complexity.

They make our code organized and easier to maintain. It adds clarity to our codes so that it's clear what we're trying to do and reduces complexity. We can write simple rules to allow complex interactions because functions can call other functions which call other functions and so on. And they also allow us to emphasize the data over the procedure. The objects have certain characteristics and they react a certain way based on those characteristics and objects allow us to have modular, reusable code.

If we create a customer object in one project, it very well may be largely reusable in another project and we'll be able to take advantage of all of the intelligence that we've put into that object in the previous project. Now that we have some idea of what objects are and why they're usable, let's get started using them.

Project setup

In this movie, we'll get our development space set up so that we can learn object oriented programming. The first thing that we'll want to do, is make sure that you know where your web document root is. That's the default directory that your web server will use to serve up documents to the browser. If you've been working in PHP chances are you know where that is right away. That's the place we're going to be locating our files. And of course you'll want to make sure that PHP is running as well. We'll do a quick test of that together.

This directory will be a place where we can put all of the examples of the different concepts as we're learning them. It'll give you a convenient location where you can refer back to those concepts if you need reference and it'll provide a space where you can experiment on your own, if you want.

For web server we will use XAMPP, and we will put all files in oop folder in htdocs, so we can access them with browser and have functional PHP engine.

So the first step is to locate the web document root. I'm going to make a new project inside NetBeansIDE and put files inside to open it. We will open in browser on our localhost folder in htdocs, for example. You can store your files online or on different server than XAMPP but you need to have PHP working. If not above mentioned is the setup we can recommend.

Object basis

Define a class

In this movie we will learn how to define a class in PHP. A class provides the definition of an object. It describes it and it serves as a template for creating new objects. Every class definition begins with the keyword class followed by the class name. Then you have curly braces that surround the class definition.

class Person {

// code for this class goes here

}

class ProductImage {

}

In the first example, the class is called Person, in the second it's ProductImage. There are a few best practices that we should observe when declaring classes.

The first is that you want to capitalize and camelcase the class names. Camelcase means that you use uppercase for the first letter and then the other words would also be capitalized as well. **ProductImage** was an example of that. Both the **P** and the **I** are capitalized. Class names should also use singular nouns. Don't use plurals because we're defining what a single object would look like. It's also a best practice to define each class in a separate file. In our final project, we're going to be doing that for sure. When we're working with it in our course folder, frequently we'll be putting some other code into the class definition file as well just for convenience and to keep it isolated as a single example but it's a best practice to keep class definitions in their own file and then you would group those class files together in a directory, frequently that's a directory that's just simply called classes. We'll learn more about writing the definition for our class in the following movies but before we do, there are two functions for classes that we should know about.

Whenever we declare a class, PHP keeps track of the fact that it's been declared and we can call **get\_declared\_classes()** which will return an array of the classes that PHP knows about. Then we also have another function called **class\_exists($string)** which takes a string as an argument and returns true or false depending on whether a class is in that array of declared classes. Let's try these out in our oop project folder.

Here I am in my project and I'm actually going to just control click on top of the sandbox and choose New File and that'll create a new file in the project. I'm just going to simply call it **class\_example.php** and you'll see that it appears right here inside folder. Once I'm there, then I can put in my class definition. Now, we have to do this inside PHP tags because of course we're working in PHP and we define a class or declare a class by simply having class and then whatever the name of the class is. I'm going to use Student. So again, remember, you want to use singular nouns and if you have something that's sort of a hyphenated or multi-word noun like StudentProfile or StudentSchedule you want to make sure that you use capitals for the second word as well.

I'm just going to use Student for this one and now I have a class definition. Now, it's not very exciting, there's nothing that's in there yet. We're going to learn how to make it more exciting later but that's all it takes to actually define a class and we can check that by getting that array using get\_declared\_classes and that will return an array of the classes and then we can actually look at that. Let's do that with echo "Classes" and let's add that to our array to actually turn the array into a string so we can output it.

We're going to use implode and then I'll use classes, then let's use a br tag at the end just to make it look nice because down here I also want to use that class\_exists. I'm going to do a couple of tests here. Let's do class\_names. We're going to make a new array and I'm going to have Product and Student and let's do lowercase student and then let's do a loop.

Foreach we'll do class\_names as class\_name and then inside our loop let's test each one of those items to see if the class exists, so we'll use class\_exists, we'll pass in the string for the class\_name and if it exists, and then let's do an else for when it doesn't exist. If it does exist, we'll echo back, let's interpolate a string here for class\_name is a declared class and let's put a br tag at the end and then we'll just copy that line, we'll paste it down here is not a declared class.

So, that'll just loop through and tell us whether something is a declared class or not. Right, let's save this file and let's go try it out. Here is how our code should look like:

class Student {  
  
}

$classes = get\_declared\_classes();

echo "Classes: " . implode(', ', $classes) . "<br />";

$class\_names = ['Product', 'Student', 'student'];

foreach ($class\_names as $class\_name) {  
if(class\_exists($class\_name)){  
echo "{$class\_name} is declared class. <br />";  
} else {  
echo "{$class\_name} is not a declared class. <br />";  
}  
}

Let's come over here to browser and open class\_example.php and there we go. Now, here's the list of classes. Notice that there's a lot of classes built into PHP, not just our Student class which is here at the end but all these other classes are there as well. These are built-in classes, these are functionality.

Yours may be different. You may have different libraries installed in your PHP, so don't panic if it's not the same but you'll see that there are a lot of classes there as well as our class and then we have **Product is not a declared class**, **Student is a declared class** and notice this, lowercase **student** is also considered a declared class, so it doesn't matter whether the string is capitalized or not, it's case insensitive when it checks, so it is still considered a declared class. So, now we know how to define a class but we don't know how to actually use a class and that's what we're going to learn how to do in the next part when we create new instances of this class.

Instances

In this movie we'll learn how to use our class definition to create an instance. Let's begin with the definition for this word instance. In general use, an instance is a single occurrence of something. When we're talking about object oriented programming, that single occurrence of something is a single object created from a class definition. So we will be using this word instance a lot. And you'll hear the terms object and instance used interchangeably. I'm certain that I'm going to do it, but instance is technically the more correct term.

So let's see how we create an instance. The first thing, of course, is we have to have a class definition before we can create an instance of it. The way we create the instance of the class is by using the new keyword followed by the class name. This is called instantiation. Or we could use it as a verb and say that we instantiate it. Instantiation is just a fancy word that means to create an instance. Note that the class name here is not a string. Once a class has been declared, we can call it by its name in PHP.

Creating a new object is almost always done while also assigning that object to a variable. Here we're assigning it to the variable **person1**. Then the variable will reference that object going forward, and we can work with it in our code. We can create a second instance of the class and assign it to a different variable, **person2**. Now we have two instances of the person class. We can have as many of these instances as we want. Each one will be a unique object, but all of them will share the same class definition.



Creating instances from a class definition is a lot like one of those memo pads on picture. Like a while you were out pad, you pull off a page every time you take a new message. Each page is different, but they all came from the same template. The structure and the blank spaces are going to be the same. We create new instances from our class definition, and then we fill out each one with different information so that they're unique.

There are two functions that are helpful when working with instances. The first is **get\_class($object)**, and it takes an instance as its argument, and it returns the name of that class. So we pass in an instance, it tells us what the class name is.

The second is **is\_a($object, $string)**. This function takes an object and a string as arguments, and it returns true or false if the object has the same class name as the string.

Let's try creating some instances in our project folder. I'm going to create new file. I'm going to make another file in here called class\_instances.php.

And inside there I'll open up my PHP tags cause obviously we are going to need those, and I'm going to go ahead and just add that same class that we had from example back here, right, class Student. Copy that, let's come back over here, and let's paste it in. Okay, so we've got our student class. Now we can create instances of that class using new and then the name of the class, not a string, the actually name of the class and a semicolon at the end. Now, of course, this will instantiate the class, but then it would just be there and there'd be no way to reference it. Right, it'd just be lost in PHP with no way to talk about it. So we want to assign it to a variable. So I'm going to have **student1**, and that's because I'm going to also copy this, and I'm going to create another instance that is **student2**. So I'm just ripping off pages of my memo pad, my student memo pad. Each one is going to be unique. Now, eventually we'll learn how to fill these out with information, like the student's first name and their last name, for example, and that'll make them unique. For now, we just have a blank class definition. It doesn't have anything more interesting in it than that, but we can try out those new functions that we talked about. This is how our code looks so far:

class Student {  
  
}

$student1 = new Student;  
$student2 = new Student;

**echo get\_class**, let's put in **student1**. That'll return its class. Let's also just put a BR tag at the end so that just drops to a new line, and then let's also go up here to this class example that we were working with before, and I'm going to grab this loop we were working with. Save some time by copying that. Coming back over I'm going to paste it in. I'm still going to use these same class names, Product, Student, student. I'm going to loop through them, but instead of **class\_exists** I'm going to use **is\_a**.

I'm going to pass it an instance. I'm going to use student1 as the instance that I'm going to pass in, and the class name is still that string as we loop through it, right? So it'll check to see, is student1 a Product? Is student1 a Student? Is student1 a student using lowercase? And then we'll echo back the results, which is going to be slightly different. Let's say student1 is a, let's remove this, and I'll just copy that line. Student1 is not a, right? So this will return true or false.

Is it that class name or not? So let's go and let's try this out. Code now looks like this:

class Student {  
  
}

$student1 = new Student;  
$student2 = new Student;

echo get\_class($student1) . "<br />";

$class\_names = ['Product', 'Student', 'student'];

foreach ($class\_names as $class\_name) {

if(is\_a($student1, $class\_name)){

echo "Student 1 is a {$class\_name}. <br />";

} else {

echo "Student 1 is not a {$class\_name}. <br />";

}

}

Let's save this, go over to browser, and open **class\_instances.php**. There we go. You can see the name of the class is student. Right, that's coming from right here from **get\_class($student1)**. So that's how I'm able to know that this is a student. Student1 is not a Product. Student1 is a Student. Student1 is a lowercase student.

So it is case insensitive when it checks. But it is a best practice to always try and use the uppercase cause that is the name of the class. So go ahead and do that. Even though it allows the lowercase, you want to try and use the correct one. Notice also that this whole process of defining a class and instantiating a class had no output whatsoever. It was only once we started doing echoing that we actually put something out in the browser. There's no output from defining a class or instantiate an object fromm that class.

Alright, now we know how to define a class and we know how to create instances of the class. Now we're ready to start making our class definitions more interesting by adding in attributes and methods to them.

Class properties

In this part, we're going to learn how to add properties to our class definitions. Properties are simply variables to hold object values. These are also frequently referred to as attributes, class variables, or instance variables, and you'll hear these terms used interchangeably all the time, they mean the same thing. They're even used interchangeably inside the PHP documentation, so don't be confused if you see it called properties one place, and attributes somewhere else. The way that we define properties is with the var keyword, followed by the variable name that we want to use, and our variable name follows all the normal variable naming rules.We also have the ability to set an initial default value for each property. Let's see an example.

class Person {

var $first\_name;

var $last\_name;

var $employed = false;

var $contry = ‘None;

}

Here I have a class Person, and inside there are four properties being defined, first name, last name, employed, and country. You can see that all of them begin with that var keyword, to let it know that it's defining a property, followed by the variable name with a dollar sign in front of it. For the first two, there's just simply a semicolon at the end of the line. For employed and country, there's also a default value that's being set, so it's being initialized to employed equals false, and country equals the string None.

The way that we set values to these attributes, and read them back, is by first creating an instance of the object, and then using arrow notation to reference the property.

$customer = new Person;

$customer->first\_name = ‘Ana’;

$customer->last\_name = ‘Stevenson’;

echo $customer->first\_name;

So here, I have the customer variable at the beginning, followed by the arrow notation, that's the minus sign and then the greater than sign, together they make an arrow, followed by the property name, in this case, first\_name. Notice that the property name does not have a dollar sign in front of it. This frequently trips up beginners. The dollar sign is only in front of the variable itself, customer. The arrow notation takes the place of the dollar sign, we don't need to have it. It knows that we're talking about a property, and it knows that that property is a variable, so there's no reason to put the dollar sign in front of it again. After that, you can see I'm using an equals sign to set a value to that property, so now, this instance of person is going to have a first name property set to Ana. I can also set the last name equal to Stevenson, then I can read those values back also by using that arrow notation.

Here, I'm asking to echo customer, and then the arrow notation, first\_name, and of course it returns back Anna. So, whether we have an equals sign after it or not determines whether we're setting or reading a value, just like with a normal variable. We can also call for the customer country, and we'll get back that default value that we set, the value None, or we can of course set it to a new value, and when we read it back, it will return the new value. Properties is where object-oriented programming gets its power, because now, each of our instances that we create can be different and unique.

We can have a thousand instances of the person object, and every single one of them can have a different first name and a last name, and therefore behave differently because of it.

There are a couple of functions that are useful for working with properties. The first is **get\_class\_vars($string)**, and it's going to return a list of the properties that are defined in this class. We don't have to have an instance to do it, we just call the name of the class using a string. We also have **get\_object\_vars(objects)**, which does the same thing, but as an argument, it takes an instance of the object. They're very similar, they return basically the same information, except that class\_vars is going to have the default values for the class, where as object\_vars will return the current values for that instance, and then last, we have **property\_exists($mixed, $strings)**, which returns true or false, based on whether a property exists on the class or on the object. You'll notice that the first argument there is called mixed, that's because mixed can either be a class name as a string, or an object instance, it'll accept either one, and then the second argument is a string, which is the name of the property we want to check for.

Let's try these in our project folder. We can create new file properties.php, and copy in it content of class\_instances.php and inside the class definition, let's put some properties.

We'll put var, dollar sign, first\_name, and var, space, dollar sign, last\_name, and then let's add one more with the default value, country equals None. You can add more if you want, but I think that's adequate, we have three properties now on Student. So, each one of these instances now has that property, right, it's just like ripping off the page of the memo pad. We've created a new student, we have those three properties available to us, so let's try setting one of them.

Let's say Student1, we'll use our arrow notation, first\_name with no dollar sign in front of it, it's not necessary when we're talking about a property. I don't want to read back the value, I know there's no value there yet, so let's set a value, let's set it equal to Lucy, and then let's do the same thing, Student1, again using the arrow notation, and then last name, and let's make her last name Ricardo, and then I'm going to copy those two, I'm going to come down here below Student2, I'm just going to change the one to be a two, let's change this one to be Ethel, and the last name to be Mertz.

Let’s try this bit of code and see how it looks like:

class Student {

var $first\_name;

var $last\_name;

var $country;

}

$student1 = new Student;

$student1->first\_name = 'Lucy';

$student1->last\_name = 'Ricardo';

$student2 = new Student;

$student2->first\_name = 'Ethel';

$student2->last\_name = 'Mertz';

echo $student1->first\_name . " " . $student1->last\_name . "<br />";

echo $student2->first\_name . " " . $student2->last\_name . "<br />";

$class\_vars = get\_class\_vars('Student');

echo "Class vars:<br />";

echo "<pre>";

print\_r($class\_vars);

echo "</pre>";

$object\_vars = get\_object\_vars($student1);

echo "Object vars:<br />";

echo "<pre>";

print\_r($object\_vars);

echo "</pre>";

if(property\_exists('Student', 'first\_name')){

echo "Property first\_name exists in Student class. <br />";

} else {

echo "Property first\_name does not exists in Student class. <br />";

}

If it does, let's echo, property, first\_name exists in Student class, and then let's just copy that line, and put it here, and change exists to does not exist in Student class. Alright, now we've got all those, let's try them out. Come back to browser, and open **class\_properties.php**.

So, the most important things to remember about properties are that you define them inside your class using var, followed by a variable name, and then when you reference them, either to read a value or set a value, you use the arrow notation without the dollar sign, and last, remember that properties are what make each of our instances unique.

Class methods

In this part, we're going to learn to add methods to our class definitions. A method is just a function inside the class to perform object actions. When we're working with procedural or non-object-oriented programming, we just call them functions. When they're inside a class, they're correctly called methods. Now you're going to see those two terms used interchangeably all the time, even in the PHP documentation, so don't get too hung up on it. A method is just a function inside a class, and in fact, you define them inside the class just like regular functions.Let's take a look how syntax looks like.

class Person{

var $first\_name;

var $last name;

function say\_hello() {

return “Hello World!”;

}

}

Here I have a class definition for Person, and it has a couple of properties for **first\_name** and **last\_name**, and then we have a method. Our method is defined using the same function keyword that we would normally use, followed by the name of a function, parentheses to surround any arguments that might be there, and then curly braces around the function definition. You can see that our function's also returning a value back, just like a normal function would. In every way, our class method is just like a function. What's different is the way that we refer to it.

When we had an instance of customer before, and we wanted to talk about its properties, we use the arrow notation to refer to those. So for the property, first\_name, we had customer and then the minus and the greater than sign to make an arrow, and then the property name.

$customer = new Person;

$customer->first\_name = ‘Ana’;

echo $customer->first\_name;

We do the same thing when we're working with methods. We use the arrow notation to refer to the method name. What's different, though, is notice the parentheses that are after the method name. That's how you can tell the difference. The parentheses are required, even if there's no arguments.If we see the arrow notation, and we see something followed by the parentheses, then we know it's a call to a method. If there are no parentheses, then we know it's a call to a property.

echo $customer->say\_hello();

It's very important, and of course, customer->say\_hello would return back the value, Hello world, and then we would echo that back. There are also some functions that are useful for working with methods. We have get\_class\_methods, which takes either a class name as a string, or an instance of an object, and returns the different methods that are available on it, and then there's method\_exists. Again, you pass in either the class name as a string or an instance as the first argument, and then the second argument would be the method that you want to know if it exists or not. Let's try these out in our project folder.

Let's begin by making a duplicate of our class properties file, and then call it class\_methods.php. Let's leave most of this the same, but now, inside our class definition, after country, let's add a method. So again, it's the function keyword, and then let's just have it say\_hello.

You can write something different if you want, and I'll just return back the value 'Hello world!' in the center column, so that's it, and we call this method. All it's going to do is return back the string, 'Hello world!'. Right, so now let's go down here and let's try calling it. After we've returned back the student's first name, let's add a new line here that's going to echo the result of calling student1, then we'll use the arrow notation and then say\_hello, and remember, we must put those parentheses after it.

That's how it's able to know that this is a method call and not a call to a property. So we'll do say\_hello, and then after it, let's just add in br tags as well. And just so you can see, I think this is clear, but let's do it again for student2. Just so you see that it does the exact same thing. We have that same method available on all instances of this object. And then let's come down here to where we have get\_class\_vars, and let's change it to get\_class\_methods for Student. That's class\_methods, and we don't need to use print\_r anymore. We have something simpler we can use. We can just simply join it together. Let's do Class methods, and join it together using implode, because it's not an associative array anymore. It's just going to be a simple array $class\_methods, and there are no object\_vars anymore, so we can remove that, and instead of property\_exists, now we're going to check to see whether method\_exists.

The method we want to check for is say\_hello, and let's say Method, let's call it say\_hello. Let's put the parentheses after it in this case. Whether it exists in the Student class, let's paste it down here, and say it does not exist in the Student class. All right, let's save it and let's try it out. Code looks like this in the end:

class Student {

var $first\_name;

var $last\_name;

var $country;

function say\_hello(){

return 'Hello world!';

}

}

$student1 = new Student;

$student1->first\_name = 'Lucy';

$student1->last\_name = 'Ricardo';

$student2 = new Student;

$student2->first\_name = 'Ethel';

$student2->last\_name = 'Mertz';

echo $student1->first\_name . " " . $student1->last\_name . "<br />";

echo $student2->first\_name . " " . $student2->last\_name . "<br />";

echo $student1->say\_hello() . "<br />";

echo $student2->say\_hello() . "<br />";

$class\_methods = get\_class\_methods('Student');

echo "Class methods:" . implode(', ', $class\_methods) . "<br />";

if(method\_exists('Student', 'say\_hello')){

echo "Method say\_hello exists in Student class. <br />";

} else {

echo "Method say\_hello doesn't exists in Student class. <br />";

}

We'll come back over here to our browser. Instead of class\_properties, let's try class\_methods. Now there you are. Notice now it's calling Hello world! It's doing it two times 'cause I'm calling it on both instances, and you can see the Class methods that are there are say\_hello. Doesn't have the parentheses after it, right, it's just the name of the different methods that are there. And finally it says yes, it does exist in the Student class.

Refer to an instance

Now that we know how to add methods to our class definitions, we need to spend some time talking about how to refer to instances from inside those methods. First, let me show you what the problem is with it and then I'll show you the solution. Let's say that we have a class definition for "person," and it has two properties for first name and last name and it has a method and we want that method to take the first name and last name and join them together with a space in between. So you might try and do it like this, using simple variable names to refer to "first\_name" and "last\_name".

class Person {

var $first\_name;

var $last\_name;

function full\_name() {

return $first\_name . “ “ . $last\_name;

}

}

So then we go to create an instance of that person and we will assign it to the variable "$p." Then we can use "$p" to refer to this instance from then on. It's very convenient, the variable now points to the instance and we can use it to refer to the different properties as well as to the method. But when we ask for full name, we don't get back the stream we expect. Instead, we get back a notice about undefined variables for "first\_name" and "last\_name."

$p = new Person;

$p ->first\_name;

$p->last\_name;

echo p->full\_name();

What's going on is that "first\_name" and "last\_name" are just simple variable names in PHP, and PHP does not know that it is looking for an object and meet properties of the object. It's treating them like simple variables. What we need is some way for the "full\_name" method to refer to this current instance and then be able to get a property from there, the same way we did with "$p" when we were outside the class. But we can't use "$p," obviously, inside the class. It's just a temporary variable that we've set.

So how do we do it? Well, when we're outside the class, we saw that we could use "$variable" and then the arrow notation. When we're inside the class, PHP gives us this special variable to use called **"$this."** It makes a lot of sense, right? It's **this** instance, so it's a variable saying, "hey, **this** instance, go and get either it's property or it's method." It can be reused for absolutely anything.

function full\_name() {

return $this->$first\_name . “ “ . $this->$last\_name;

It's just a way to refer to this current instance. So, instead you would have something like this. Notice now, that in the "full\_name" method, it's calling "$this->first\_name" and concatenating it with "$this->last\_name." Now when we call the "full\_name" method, it returns the string that we would expect.

Let's try an example. Instead of creating a new file for this, let's actually do this in our class\_methods.php file that we already have, and save it as class\_methods2.php.

Let's just add a new method here and we're going to call it "full\_name" just like our example. Now we don't have to pass any values to it, we are going to use these attributes. We are going to do that by returning back the value of "$this->first\_name" and then let's concatenate that together with "$this->last\_name." And that's it, go down and try and use it. And let's see right here where we were concatenating this together and calling them, let's just change this to be "full\_name," and remember we have to have the parentheses at the end.

Now copy that and I'll replace this one with the same thing. Now let's go and bring that page back up. Here are our class methods, let's hit "return," and you'll see that it works. Notice now, class methods added a new method that's here, "full\_name." And you'll see that it works; we still get both "Lacy Ricardo" and "Ethel Mertz," which is the results of that. Now if you want to try it the other way just to see, just take this out for a second, try it the other way and let's reload the page. You'll see we get a notice, see? "Undefined variable: first\_name." We need to have "$this" in front of it so that it knows that it's talking about this current instance.

function full\_name(){

return $this->first\_name . " " . $this->last\_name;

}

$student1 = new Student;

$student1->first\_name = 'Ana';

$student1->last\_name = 'Stevens';

$student2 = new Student;

$student2->first\_name = 'John';

$student2->last\_name = 'Doe';

echo $student1->full\_name() . "<br />";

echo $student2->full\_name() . "<br />";

Another way to think about it, is just any time we're working with either a property or a method, we are going to use the arrow notation. It doesn't matter if we are inside or outside of the class. The arrow notation is how we know we're working in an object-oriented way.

Challenge: Properties and Methods!

It's time for our first challenge. In this challenge, we want to put all the things we've worked on in this chapter to the test. Defining classes, adding properties, and adding methods. To do that, I want you to create a new file inside your project folder, which is going to be called **challenge01.php**. Inside that file, you'll define a new class for a bicycle. We're going to be creating instances of bicycles. And the **properties** that each bicycle will have are **brand, model, year, description, and weight\_kg.**

That's a place for storing the weight in kilograms. Then, in addition to that, we're going to have some **methods**. We're going to have **name**, **weight\_lbs**, which is an abbreviation for pounds, and **set\_weight\_lbs**. So let me describe what all those methods should do.

The first one is just going to be a simple **name** for this bicycle. What I want you to do is take brand, model, and year, and put them together in a string of your choosing to output a simple name for this bicycle. So for each instance, you should be able to find out, tell me about this bicycle, what's its name. And the name is going to be a combination of brand, model, and year.

Then, for **weight\_lbs**, what I'd like you to do is convert from kilograms into pounds. And the conversion that you can use for that is that one kilogram equals 2.2046226218 pounds. So that means that if you have a certain number of kilograms and you want to find out the number of pounds, you're going to multiply in order to find that value.

And then last, **set\_weight\_lbs** is going to be a way that we can pass in a weight, in pounds, and have it converted into kilograms and stored in weight underscore kg.

So you'll be able to set the weight, either by kilograms, or by pounds, depending on which way you do it. That last one is a little bit tricky. After you've done that, you should instantiate a few different objects. You should practice setting and reading their properties, and you should try calling all of the methods that you've created to make sure they behave the way you expect. Play around with it, get used to it. This is your chance to experiment with objects and get familiar with them.

Solution: Properties and Methods

I hope that you were able to complete our first challenge assignment. If not, follow along with me as I show you the solution that I came up with. So the first thing that I did was I created a new file inside our OOP directory called challenge1.php.

My code looks like this, but yours can be similar, as long as you followed our previous classes.

class Bicycle {

var $brand;

var $model;

var $year;

var $description;

var $weight\_kg;

function name(){

return $this->brand . ", ". $this->model . ", " . $this->year;

}

function weight\_lbs(){

return floatval($this->weight\_kg) \* 2.2046226218;

}

function set\_weight\_lbs($value){

$this->weight\_kg = floatval($value) / 2.2046226218;

}

}

$trek = new Bicycle;

$trek->brand = "Trek";

$trek->model = "Emonda";

$trek->year = "2007";

$trek->weight\_kg = 1.0;

$cd = new Bicycle;

$cd->brand = "Capriolo";

$cd->model = "Synapse";

$cd->year = "2016";

$cd->weight\_kg = 1.0;

echo $trek->name() . "<br />";

echo $cd->name() . "<br />";

echo $trek->weight\_kg . "<br />";

echo $trek->weight\_lbs() . "<br />";

$trek->set\_weight\_lbs(2);

echo $trek->weight\_kg . "<br />";

echo $trek->weight\_lbs() . "<br />";

If you didn’t make code like this it is time for you to review what we learned and get back to this challenge again.

Let’s move to another chapter!

Class inheritance

What is inheritance

In this chapter, we're going to be learning about inheritance. And we're going to start out by getting an overview of what inheritance is, and why we need it. Inheritance is when a new class takes on the properties and the methods of an existing class. Why would we want to do that? Well, using class inheritance helps us to organize our code, and by inheriting code from one class to another, we prevent repetition. That in turn simplifies the maintenance of the code, and it also prevents inconsistencies and bugs.

Let me give you an example so that you can see why it's important. Let's say that we have couple of different classes, like in this example:

class ProductBrochure {

var $file\_name;

function download\_path(){

return "/downloads/" . $this->file\_name;

}

}

class WarantyDocument {

var $file\_name;

function download\_path(){

return "/downloads/" . $this->file\_name;

}

}class CleaningInstructions {

var $file\_name;

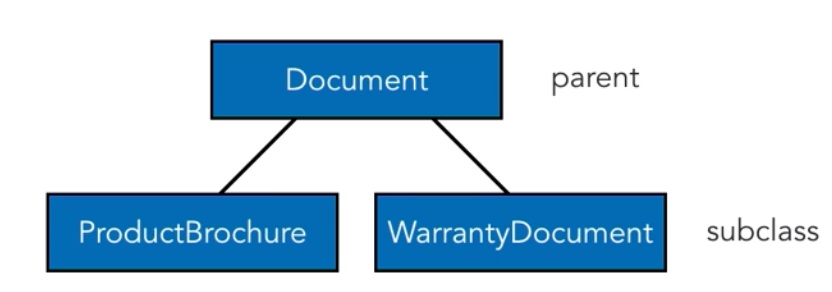
}

Let's say we have one called ProductBrochure. You can see it has a property for file\_name and a method for download\_path. Then we have another class, WarrantyDocument. It's very similar. It's also going to be working with a file, some kind of a PDF, perhaps. And it's going to also have a file\_name property, and it's going to have another method called download\_path. I'm repeating myself here. Now, if we want to make a change to the download\_path method, we've got to remember to make that change in two different places.

If we don't, we'll have an inconsistency, and a bug. So, you can see how repetition makes our code harder to maintain. Notice that we also got a third class there called CleaningInstructions. This would also be a document. But notice that it's not defined the same way. We have an inconsistency. We have filename, with no underscore there, and we haven't defined the method for download\_path. If all of these were able to inherit the same code, and able to inherit the same properties and attributes, they could all behave the same way.

The code would be organized, consistent, and would have less bugs. When objects are similar, and share many of the same properties and methods, then class inheritance can keep their code organized. Behaviors of a class are shared with all of its subclasses. We are using behaviors here as a shorthand for both properties and methods. Those are the behaviors of the class. And when we add new behaviors to the parent class, then all the subclasses also gain those behaviors. Now, the subclasses aren't stuck with them; they don't have to use them.

They can ignore them, or they can override those parent behaviors with different behaviors. Or, they can add new behaviors that the parent doesn't have. They're just inheriting it, but after that, they can go on to define themselves in additional ways or override their inherited behavior. So, let's imagine that instead of having just a ProductBrochure and a WarrantyDocument, that both of those inherited from another class called Document. Document would be called the parent class, and ProductBrochure and WarrantyDocument would be the subclasses.



Parent can also be called superclass, but in PHP, we're going to stick with the term parent because we're going to actually use that name later in our code. Both ProductBrochure and WarrantyDocument will inherit the same properties and methods from Document. We won't make a mistake and implement them differently. If we want to make a change to one of those behaviors, we don't have to track it down in several different places. We can just go into document and make the change one time. And both of the subclasses will get that change then. But each subclass can also override those inherited behaviors if they want to do things differently.

Now that we understand the benefits and the concepts behind inheritance, let's see how we implement it in PHP.

Define a subclass

Now that we have an overview of how class inheritance works and why we want to use it, let's learn to actually implement it in PHP by defining a subclass. The way that you define a subclass in PHP is by first defining the parent class, just like you normally would, and then you define the subclass in the same way, except that you add another keyword after the class name, which is Extends, and then the name of the parent class that you want to inherit from. That's all it takes to make a subclass.

class Parent {

}

class Subclass extends Parent {

}

The parent must be declared first, or else PHP would complain when it gets to the part where it says extends Parent. It would say, I don't know what parent class you're talking about. So we must always declare the parent class first, and then we can easily make a subclass from it by using the keyword Extends. To use the example from the previous movie, you can see here we have a parent class called Document, and it has a property and a method in it, and then we have three classes that are subclasses. ProductBrochure, WarrantyDocument, and CleaningInstructions.

class Document {

var $file\_name;

function download\_path(){

return "/downloads/" . $this->file\_name;

}

}

class ProductBrochure extends Document {

}

class WarantyDocument extends Document {

}

class CleaningInstructions extends Document {

}

All of those extend the document, so they use that keyword extends, and then the name of that parent class, Document. All three of those classes now have that same property and that same method in them.

Here's another example using Vehicle as a parent class. It has a subclass of Car, and a subclass of Motorcycle. The fourth class that's being declared is CompactCar, and you'll notice that it doesn't extend Vehicle, it extends Car.

class Vehicle {

var $wheels;

var $doors;

function gasolyne\_type(){

return 'Unleaded';

}

}

class Car extends Vehicle {

}

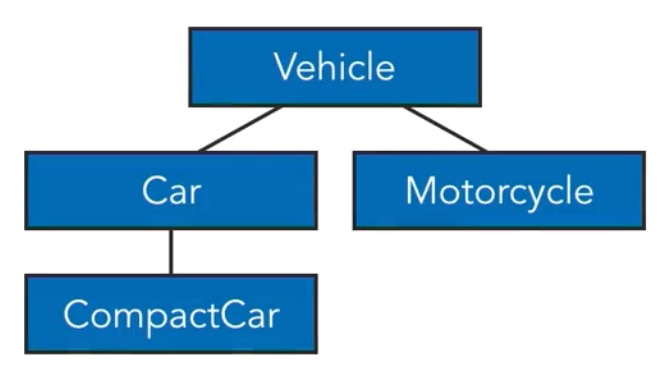
class Motorcycle extends Vehicle {

}

class CompactCar extends Car {

}

So it gets all the properties and methods in Vehicle, but also any additional or overridden behaviors in the Car class. You can visualize it like this. It's perfectly fine for subclasses to also have additional subclasses.



All of the behaviors would just trickle down to the subclasses. PHP has several built-in functions which are useful when working with inheritance. **get\_parent\_class()**, which can either take the name of the class or an instance of the class as an argument. **is\_subclass\_of()**, which also can take the name of a class or an instance as its first argument, and then the parent class. You want to find out if something is a subclass of whatever that parent class might be, or we can ask for all **class\_parents**. We can pass in either the name of the class or we can pass in an instance and it'll tell us all of the parent. So in the case of the example we just saw, CompactCar would have parents that included both Vehicle and Car.

Let's try defining subclasses in our project folder. I've already created a PHP file called class\_inheritance.php . So I've got class of User, which as $first\_name, $last\_name and $username. I've got a full\_name method, which is familiar to us by now. It's just taking those different properties and concatenating them together. Then notice I've got a subclass here called Customer. Customer extends User. So now Customer is a subclass of User.

Let's try using these. So let's first just create a new user, user equals new User. Now I have an instance of the User class, $u->first\_name, make it Jerry. And $u->last\_name, will be equal to Seinfeld. $u->username is going to be equal to jseinfeld. All right, so now we've got one user.

Let's just duplicate that user. I'm going to copy those lines, and this one I'm going to make the variable that I use is going to be $c, and we're going to create George Costanza, and gconstanza. Okay, you can use any names you want, oops, it looks like I made a mistake here, I actually forgot username, there we go. Now I've typed it correctly. All right, so now I've created two instances of the user class, that's nothing new. We know how to work with those.

Now, though, I'm going to change this, so that instead it's an instance of the Customer class, right, it's a different class. Now, we're going to learn how to make changes to that Customer class so that it actually behaves different than User. For right now, it behaves exactly the same. It has the same properties and it has the same methods. And we can see that if we drop down here, and we'll just ask for $u and full\_name, let's echo that. Full\_name, that's a method, and put a br tag at the end, and then I'll just copy that.

Let's do it again, this time with the $c variable. So we're going to ask each of those to just tell us what their full name is, and we'll see that the subclass inherited that full\_name method. Before we do that, let's also just add another one here for get\_parent\_class, and we'll do that for $u, and again we'll put that br tag at the end, and then let's do the came thing but for $c, see what it's parent class is. And then let's use that if(is\_subclass\_of, and we can use either a name or an instance.

class User {

var $first\_name;

var $last\_name;

var $username;

function full\_name(){

return $this->first\_name. " " . $this->last\_name;

}

}

class Customer extends User {

}

$u = new User;

$u->first\_name = 'Jerry';

$u->last\_name = 'Seinfeld';

$u->username = 'jseinfeld';

$c = new Customer;

$c->first\_name = 'George';

$c->last\_name = 'Constanza';

$c->username = 'gcostanza';

echo $u->full\_name() . '<br />';

echo $c->full\_name() . '<br />';

echo get\_parent\_class($u) . '<br />';

echo get\_parent\_class($c) . '<br />';

It's more common to use an instance because we're really probably curious about a particular instance. We probably know about the structure of our classes and subclasses, and don't need to ask PHP to tell us. So let's see if it's a subclass of user, then we'll return back a line that says echo Instance is a subclass of User, and last of all let's try that class parents. $parents = class\_parents, and we'll ask the $c instance.

And then let's echo that back. It's going to return an array. Implode, there we go, $parents, and we'll just grab this at the end and paste that in there.

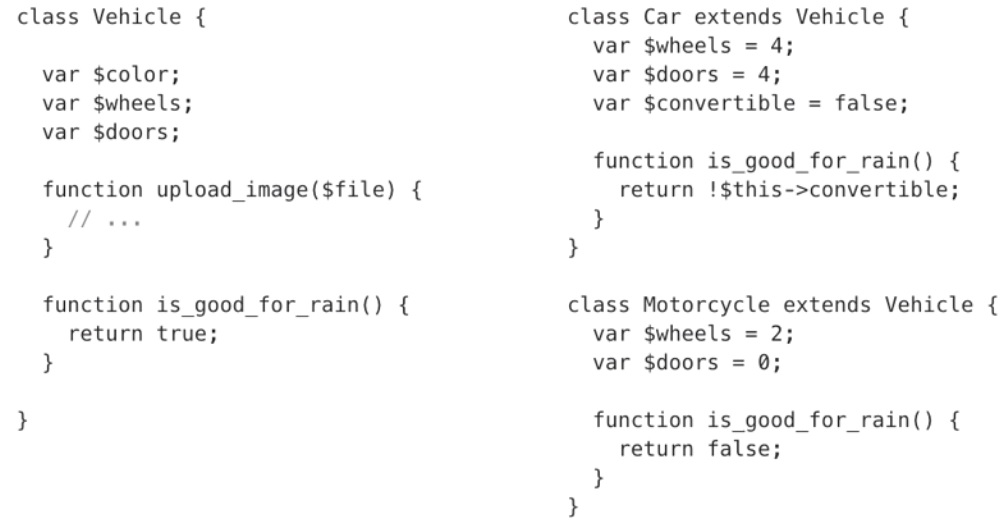
Let's go over to our browser, and the page that we want is class\_inheritance. There we go, you can see they both have that full name method available to them.

You can see that when we ask for the parent of the user class, it didn't give us anything. It came back with a blank line. When we ask for the parent of the customer class, they came back and told us it was User. And yes, instance is a subclass of User, that returned true. And, when we asked for the parents of the Customer class, it returned back just User. Right, there's only one. If there had been more than one, we had several layers, then it would have bene a list of those with a comma between them.

So that's all there is to being able to define a subclass. Now, the reason to do this, though, of course is because it gets more interesting when we can start to customize the Customer class, so that it can make use of the parts of User that it wants, but it also can extend or override that behavior as well. And that's what we'll talk about in the next parts.

Extend and override

We know how to define a subclass. But the real power of subclasses is their ability to extend and override behaviors that they inherit from the parent class. We're going to explore that idea in this part. Here you can see I have three example classes, vehicle, car, and motorcycle, and you'll notice that car and motorcycle, over on the right, are subclasses of the vehicle class over on the left. That means that they inherit the properties and the methods that are defined in the vehicle class. It's as if that code that was inside the dealer class was the first section of code inside each of those class definitions.



So, things like the color property are available in all three of these classes. The wheels property is also available in all three of these classes. But you'll notice that the car class and the motorcycle class actually redefine those values. When we redefine a property or a method inside the subclass, it has the effect of overriding the value that was inherited. So, in this case, even though they all three still have a wheels property, it is a different behavior. In the case of vehicle, wheels has no default value.

In the case of car, it has four wheels. In the case of motorcycle, it has two wheels. So if we create an instance of the vehicle class, we get one behavior. If we create an instance from the car class, we get a different behavior. Notice also that the car class has another property, called convertible, which the other two classes don't have. The idea here is whether or not the car's roof can be opened up and folded back. When I was designing this, I decided that that was a property that was applicable to the car class, but not to vehicles generally, and certainly not to motorcycles.

This is an example how we can not only inherit values from a parent class, but we can extend the behavior by adding more code into the subclass, and that's part of why we use that word extends as a key word when we're creating subclasses. We're extending the behaviors by either overriding them or adding new things. And the same is true for methods as well. **upload\_ image** is a method that's inherited, and therefore at all three of the classes. The **is\_good\_for\_rain** method is also available in all three of the classes, but it's behavior is very different in each case.

For vehicles, by default, it's just going to return true. For a motorcycle, by default, it's going to return false. And for a car, it's behavior is going to depend on whether the property convertible has been set to true or false. So you start to get the idea of how we can inherit, override or extend behaviors inside our subclasses. Deciding how to organize your code and deciding which behaviors get inherited, overridden and extended is going to require judgment calls on your part.

For example, we could have decided that, by default, all vehicles should have four wheels, and then we wouldn't need to override it in the car class. Or we could have decided that convertible was a property that should be available in the vehicle class as well, so that it was inherited by both car and motorcycle. Or we could have architected things completely differently and created another subclass called convertible car, which inherited from the car class. Knowing how to arrange code takes experience and sometimes just good old trial and error.

The one bit of guidance we can give you is to remember that the purpose of inheritance is to keep your code well-organized and to keep you from repeating yourself, so keep that in mind. If you find that your code is less organized, then you may want to rethink your approach.

Let's try an example of extending and overriding in our OOP project folder. We can save existing file and save it as class\_inheritance2-php. Let's continue working with our user and customer classes that we were working with before, and let's add in some extensions here, let's add in var city for our customer, var state, or, if you prefer, province, and var country.

The idea being that our customer is different than our user, because our customer might have, for example, a billing address. A way that we want to contact them. Or maybe we just want to know what their location is. So that's what we're going to have here, we're going to ask what their location is, it's going to return this city, concatenated together with a comma and a space, and then a period, and I'm just going to copy that. Save me some typing, do it a couple more times, because it's going to be city followed by the state, and then country.

class User {

var $first\_name;

var $last\_name;

var $username;

var $is\_admin = false;

function full\_name(){

return $this->first\_name. " " . $this->last\_name;

}

}

class Customer extends User {

var $city;

var $state; //or province

var $country;

function location() {

return $this->city .", ". $this->state . ", ". $this->country;

}

}

class AdminUser extends User{

var $is\_admin = true;

function full\_name\_admin(){

return $this->first\_name . " " . $this->last\_name. ", " . $this->is\_admin;

}

}

$u = new User;

$u->first\_name = 'Jerry';

$u->last\_name = 'Seinfeld';

$u->username = 'jseinfeld';

$c = new Customer;

$c->first\_name = 'George';

$c->last\_name = 'Constanza';

$c->username = 'gcostanza';

$c->city = 'New York';

$c->state = 'New York';

$c->country = 'USA';

$a = new AdminUser;

$a->first\_name = 'Admin';

$a->last\_name = 'Adminsky';

echo $u->full\_name() . '<br />';

echo $c->full\_name() . '<br />';

echo $c->location() . '<br />';

echo $a->full\_name\_admin() . '<br />';

echo get\_parent\_class($u) . '<br />';

echo get\_parent\_class($c) . '<br />';

if(is\_subclass\_of($c, 'User')) {

echo "Instance is subclass of User. <br />";

}

$parents = class\_parents($c);

echo implode(', ', $parents). '<br />';

So it'll just return city, state and country with commas in between them whenever I ask for location. So let's come down here and for our customer, let's add in those values. City is going to be equal to New York. Again, I'll just save myself a little typing by cutting those, and we'll call this state, this one's going to be country. New York, New York, that's fine. I just have to change this to United States. Okay, so now that I have that, I can drop down here, and right below the full name, let's add a line, and instead of the full name, we want to ask for the customer's location.

Alright, let's try it. Let's go over here. There we are, New York, New York, United States. Now, the important thing to realize here is that we cannot take this line and change this to a u or user. Our user up here is a user instance, not a customer instance, and the user class does not have any concept of city, state, country or location. If you want to try it, just to see what happens, we can come over here and reload the page, and you'll see call to an undefined method, right? There is no method called location inside the user class, so let's just comment that line out.

Say no method error. So that's an example of how we're able to extend the class. What about overriding the class? Let's say we're going to have our class here, let's have another var is\_admin, and by default, is\_admin is going to be false. And then let's just take that line and let's come down here, let's create a new class, it's going to be called AdminUser, extends user, it'll be exactly like our user in every other way, except that is\_admin is going to be set to true.

Now, customer, is it going to be an admin or not? No, it's not. It's going to have this is\_admin property, because it's going to inherit it with the default value of false. AdminUser is going to modify that value when it's inherited here. Now, we could also do the same thing with methods, let's just try real quick, we'll take full\_name, we'll come down here to our AdminUser, and whenever we have an AdminUser, we're also going to append to the end that their name is parentheses Admin. Let's put a space there, there we go.

So any AdminUser's full name will report the fact that they are an admin. Hopefully, you're beginning to understand the utility of being able to create parent classes and subclasses.

Challendge Inheritance

It's time for another challenge assignment. For this challenge, I want you to try creating PHP Classes which use Inheritance. I'm not going to give you a set of classes for you to work with. I want you to choose your own category. Pick a topic that interests you. Maybe it's a hobby of yours, or it's something that you frequently work with. Or, something you're very familiar with. You want to try and find a category that has subcategories which have similarities to the parent category and have shared attributes. After all, that's when Inheritance works best.

And, that's part of why I don't want to give you a list of things to work with. It's because I want you to do some thinking about when Inheritance would actually be useful. Once you do that, take the time to map out the Inheritance on paper. Before you start writing code, get out a piece of paper, draw some boxes, list out the properties and methods, and make sure that you have a good feeling for what the Inheritance is going to look like. It'll also help you to identify if you've picked the bad category and need to rethink it again. Once you've worked that out, then you can start to code. You want to create a new file in the project folder called challenge\_02.php, and in there you'll define your PHP Classes.

Once you have your classes set up, as parents and subclasses, then you'll want to inherit, extend, and override various properties and methods. Get creative. Once you do that, then create instances so you can test your assumptions, and make sure that you understand what properties and methods are available in each one of these instances. Try creating instances of the parent class, as well as instances of the subclasses. If you're really stuck for ideas, and you can't think of a good category, let me give you a couple of ones that are frequently used for examples.

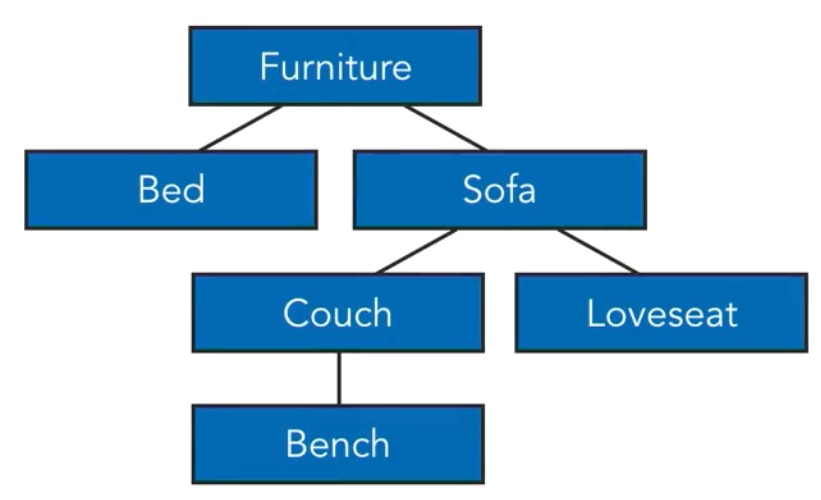
Animals, so for example, you would have categories like mammals and reptiles. Or, you could work with just specific animals, maybe horses zebras and donkeys. Those are all similar. They all have characteristics that are in common. Or clothing. Maybe it's shirts, and you have Polo shirts and dress shirts. Or, it's dresses. And, dresses have shift dresses, A-line dresses, and wrap dresses. There's foods. Meats, fruits, vegetables. You could attributes like, whether something has seeds.

Or, whether it can be eaten raw. And, furniture. Furniture is another good category because there are lots of similarities between lots of different types of furniture. You can have couches and love seats. You can have chairs. Those are all very similar, but there are differences, as well. Beds, cabinets, desks, again they have similarities, but they're not all the same. For example, all furniture has common attributes for width, height, and depth, but they might have unique characteristics for things like the number of drawer it has or the cushions and the upholstery.

And then, last of all, musical instruments. So, for example, woodwinds, brass, percussion, keyboards, all of those are possibilities. And again, there's lots of inherited attributes where things are similar to a parent category, but not exactly the same. Again, spend some time mapping it out on paper, and then try writing your code.

Solution

Hopefully you were able to complete the challenge assignment on inheritance. Remember there is no absolute right or wrong answer to this. The goal of the challenge was to get a feel for how inheritance works. As a demonstration of the kinds of results that you should have had, I'm going to show you the solution that I came up with using furniture as a category. Once I decided to use the furniture category, I tried to think of subclasses that had some properties in common and some differences.



There are common properties to all furniture. All furniture has a width, a depth, and a height and therefore we can also calculate the area of the amount of floor space that it takes up and the volume. So all of my subclasses are going to inherit those behaviors. I decided to choose two main subclasses, which are bed and sofa. A bed is a piece of furniture that you sleep on, so I decided there'd be an attribute called is\_sleeper that would determine if we could sleep on it and a sofa is something that you sit on, so it's going to have property called is\_seating.

Now of course you can sit on a bed and you can sleep on a sofa, but I'm talking about its intended purpose. Now a sofa can also be a sleeper sofa, so it's an attribute that we can change. By default it's not going to be a sleeper, but we change that attribute so that it becomes a sleeper sofa. Below sofa I decided to expand with some more subclasses and I chose couch, loveseat, and bench. The terms couch and sofa are often used interchangeably, but technically a couch is a sofa that doesn't have any arms.

You may not have known that. So it's going to inherit sofa and have all of it's behaviors, but by default, instead of having two arms, it's going to have zero. And if a couch has zero arms, then a bench is essentially the same thing as couch, but without a back on it. A loveseat is another kind of sofa, it's just narrower, which means that it has less seating. A sofa typically seats three people, a loveseat typically seats two. Once I had an idea of the hierarchy that I wanted to use and I had a sense of the attributes that would be shared between the different classes, then I set out to write the code for it.

The first thing that we did was we created a new file inside the project folder called challenge\_02.php and we set out to write my furniture class. So we gave it a width, a depth, and a height property and I gave it methods for area and volume. Then I also added properties for is\_seating and is\_sleeper which default to false. By default, furniture's going to have those set to false. And that's because other kinds of furniture I might want to add in the future might be a cabinet or a desk or something like that. So by default they're going to be false.

<?php

class Furniture {

var $width;

var $depth;

var $height;

var $is\_seating = false;

var $is\_sleeper = false;

var $is\_upholsetered = false;

function area(){

return floatval($this->width) \* floatval($this->depth);

}

function volume(){

return $this->area() \* floatval($this->height);

}

}

class Bed extends Furniture {

var $is\_sleeper = true;

}

class Sofa extends Furniture {

var $is\_seating = true;

var $is\_upholstered = true;

var $seats = 3;

var $has\_seatcushions = true;

var $has\_backcushions = true;

var $arms = 2;

var $depth\_opened;

function area\_opened(){

if($this->is\_sleeper) { return $this->area();}

return floatval($this->width) \* floatval($this->depth\_opened);

}

}

class Couch extends Sofa {

var $arms = 0;

}

class Loveseat extends Sofa {

var $seat = 2;

}

class Bench extends Couch {

var $has\_backcushions = false;

}

echo "<hr />";

$sofa = new Sofa;

$sofa->width = 4;

$sofa->depth = 2;

$sofa->height = 3;

echo 'Area: ' . $sofa->area() . "<br />";

echo 'Volume: ' . $sofa->volume() . "<br />";

echo 'Area opened: '. $sofa->area\_opened() . "<br />";

$sofa->is\_sleeper = true;

$sofa->depth\_opened = 7;

echo 'Area opened: '. $sofa->area\_opened() . "<br />";

echo "<hr />";

$bench = new Bench;

$bench->width = 3;

$bench->depth = 1.5;

$bench->height = 1;

echo 'Area: ' . $bench->area() . "<br />";

echo 'Seating? ' . ($bench->is\_seating ? 'true' : 'false'). "<br />";

echo 'Sleeper? ' . ($bench->is\_sleeper ? 'true' : 'false'). "<br />";

echo 'Backcushions? ' . ($bench->has\_backcushions ? 'true' : 'false'). "<br />";

And I also added another one for is\_upholstered which is also set to false. All right let's fold up furniture. We can just click right and fold it up. And then let's take a look at the bed class. Bed is going to be a subclass of furniture. It's going to inherit all of the properties and methods we just defined, but it's going to override is\_sleeper to be is true. I decided to just leave a bed at that. A bed is just a piece of furniture you can sleep on and I didn't go any further. On sofa, I set is\_seating equal to true by default and is\_upholstered equal to true by default.

I overrode each one of those. I also added some new properties. I said that it has a seats property that defaults to three. Has seat cushions is equal to true, has back cushions is equal to true, and has arms is equal to two, all by default. Now of course, different kinds of sofas can be different. Some of them may have seat cushions, may have back cushions, and so on. These are attributes that we can change. Now I could have certainly put those up into the furniture class and inherited them, but I decided they were unique enough to this seating type of furniture that I would include it here.

I could have also had another subclass in between called seating furniture and put them there, but that was a bit of overkill for what I was trying to do here, so I just introduced them as an extension of the furniture class inside my sofa class. I also added another one for depth\_opened. The idea is, if this turns out to be a sleeper sofa, then it's depth is different. It's width is the same but its depth changes. So I wanted to be able to define that in that case. Now you don't see is\_sleeper here, but remember that's something we're inheriting. Is\_sleeper is false by default, but if it were set to true, then we would want to take into account the depth\_opened.

So I also created a new method here called area\_opened. It's going to check and see if it's a sleeper or not. If it's not a sleeper, then we'll just return the regular area, but if it is a sleeper, then we want to calculate the area using depth\_opened instead of the regular depth. Let's fold that up and then let's take a look at the last three, couch, loveseat, and bench. A couch is just a sofa that defaults to having zero arms. Now I could have one arm or two arms, but by default we're going to say it has zero. A loveseat is just a sofa that has two seats instead of three.

And a bench is just a couch that has no back cushions. What I want you to notice here, is that I'm creating a new class, a new type of object that I can work with, but it's inheriting a lot of behaviors from those parent classes. So even thought its definition is quite simple, there's lots and lots of complexity there. And if in the future, I come back and I add more methods or properties to furniture, well they're automatically going to appear in all of these different subclasses. That's one of the advantages to using inheritance in object-oriented programming.

Right it doesn't show anything 'cause it's false so we don't see it there. But is\_seating, is\_upholstered, right, all those things are being inherited even though our class definition is really quite small. Okay, so now that we've seen the different ways inheritance works, let's try it out. I'm going to create a new sofa. I'm going to give it a width, a depth, and a height and then I can return its area and its volume. I'm also going to return it's area\_opened. Now this is not a sleeper sofa. Right, I haven't told it, by default it's not a sleeper.

So area\_opened should be the same as the area. And then I'm going to change it into a sleeper sofa. I'm going to say, you know what, let's make it a sleeper sofa and let's give it depth\_opened and let's calculate that value again, see what we get back. Let's go over here and let's take a look at that and you'll see we get the area, eight, the volume is 24. Area\_opened is eight when it's not a sleeper sofa. Once I convert it into a sleeper sofa, the area\_opened changes into being 28. And then down here for our bench, create a new bench, right, a new instance of the bench class.

So it's going to inherit all the behaviors of the parent classes as well as as the extensions and overrides that we've done in the bench class. So I have width depth and height. I can get its area and then I can check, is it seating, is it sleeping, does it have back cushions. And you can see here, here's the area. Seating is true, sleeping is false, back cushions is false. So again, there is no absolute right answer to what you should have come up with here. But you should now have a feel for how inheritance works.

Exapmle2: Relestate/Nekretnine

class Nekretnina {

var $zemlja;

var $grad;

var $deo\_grada;

var $kvadratura;

var $uknjizeno = true;

var $provizija = true;

var $dodatni\_opis;

function opis\_nekretnine() {

return $this->zemlja . ", ". $this->grad . ", " . $this->deo\_grada . ", " . $this->kvadratura . "m2, uknjizeno: " . $this->uknjizeno . ", provizija: " . $this->provizija . ". ";

}

function dodatno(){

return $this->dodatni\_opis;

}

}

class Stan extends Nekretnina {

var $broj\_soba;

var $podrum = true;

var $vrsta\_grejanja;

var $terasa;

var $dodatni\_opis;

function opis\_stana(){

return "Broj soba: " . $this->broj\_soba . " , Podrum: " . $this->podrum . ", Grejanje: " . $this->vrsta\_grejanja . ", Terasa: " . $this->terasa . ". ";

}

}

class Kuca extends Nekretnina {

var $broj\_spratova;

var $okucnica = true;

}

class Zemljiste extends Nekretnina {

var $gradjevinska\_dozvola = false;

var $klasa\_zemljista;

}

class Garsonjera extends Stan {

var $odvojena\_kuhinja = true;

}

class Studio extends Garsonjera {

var $zvucna\_izolacija = true;

}

class PoljoprivrednoZemljiste extends Zemljiste {

var $gradjevinska\_dozvola = false;

}

class GradjevinskiPlac extends Zemljiste {

var $gradjevinska\_dozvola = true;

}

function inspect\_class($class\_name) {

$output = '';

$output .= $class\_name;

$parent\_class = get\_parent\_class($class\_name);

if($parent\_class != ''){

$output .= " extends {$parent\_class}";

}

$output .= "\n";

$class\_vars = get\_class\_vars($class\_name);

ksort($class\_vars);

$output .= "properties: \n";

foreach ($class\_vars as $k => $v){

$output .= "- {$k}: {$v}\n";

}

$class\_methods = get\_class\_methods($class\_name);

sort($class\_methods);

$output .= "methods: \n";

foreach ($class\_methods as $k) {

$output .= "- {$k}\n";

}

return $output;

}

$class\_names = ['Nekretnina', 'Stan', 'Kuca', 'Zemljiste', 'Garsonjera', 'Studio', 'PoljoprivrednoZemljiste', 'GradjevinskiPlac'];

foreach ($class\_names as $class\_name){

echo nl2br(inspect\_class($class\_name));

echo '<br />';

}

echo "<hr />";

$stan = new Stan;

$stan->broj\_soba = 2;

$stan->kvadratura = 56;

$stan->zemlja = 'Srbija';

$stan->grad = 'Beograd';

$stan->deo\_grada = 'Banovo Brdo';

$stan->uknjizeno = true;

$stan->vrsta\_grejanja = 'centralno';

$stan->terasa = true;

$stan->dodatni\_opis = 'Renoviran, luks, na odlicnoj lokaciji...';

echo 'Stan: ';

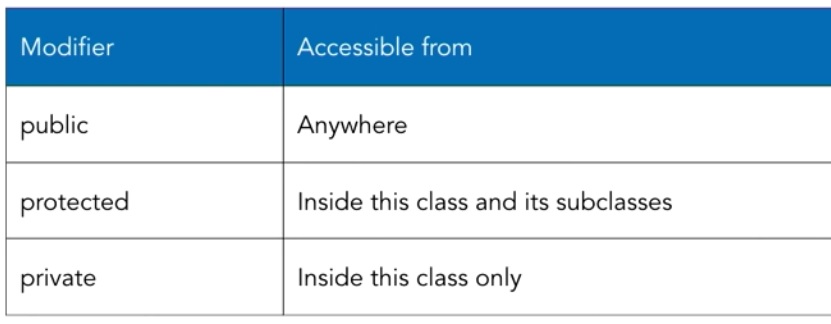
echo $stan->opis\_nekretnine() . $stan->opis\_stana() . $stan->dodatno();

echo "<hr />";

Object access control

Visibility modifiers

In this chapter we will learn about PHPs visibility modifiers. Visibility modifiers are used to control access to objects, properties, and methods. There are three visibility modifiers. Public, protected, and private.



Any property or method listed as being public is going to be accessible from anywhere. Anywhere in our code that we want to reference those, we're able to. It doesn't matter whether we're inside the class definition, whether we're inside the code that belongs to a sub class, or whether we've created an instance of the object and we're working with it there.

Private on the other hand is the opposite extreme. If we have a property or method labeled as being private, it's accessible from only inside the class code. That means actually inside the curly braces that define this class. It's not accessible in any sub classes, and it's not available if we were to create an instance of the object and try to reference the property or method that way. Protected works just like private, but it also allows sub classes to be able to access the property or method.Let me show you how we declare these modifiers.

class Example {

public $a;

protected $b;

private $c;

public function hello\_world(){

return "Hello everyone!";

}

protected function hello\_family(){

return "Hello family!";

}

private function hello\_me(){

return "Hello me!";

}

}

Here we have a class called example. And you'll notice that when we declare the three properties, $a, $b, and $c, we don't use the keyword var at all. Instead, we declare it with this visibility modifier. Public, protected, or private. Now if we use var, it's the same thing as being public by default. But it's a better practice to actually use the word public than to use var. We've been using var so far only because we didn't know about visibility modifiers yet. When we're working with methods, it's a little bit different. Instead of replacing the keyword, we're going to prepend the modifier in front of the word function. So now instead of just function, it's public function, protected function, and private function. If we didn't declare it, the default would be for it to be public. Let's look at a more concrete example.

class User {

private $is\_admin = false;

private function has\_admin\_access (){

return $this->is\_admin === true;

}

public function access\_level() {

return $this->has\_admin\_access() ? 'Admin' : 'Standard';

}

}

$user = new User;

echo $user->is\_admin; //Error: Cannot access private property

$user->is\_admin = true; //Error: Cannot access private property

echo $user->has\_admin\_access(); //Error: Call to private method

echo $user->access\_level(); // 'Standard'

Here we have a class called user, and we have a private property called is\_admin() which defaults to false. So this user is not an admin. That's the kind of thing we wouldn't want people to be able to change easily, right? We wouldn't even necessarily want other parts of our code to be able to change that easily.

So we can restrict access to it by making it private. That means that if we were to create a new instance of the user, and we were then to ask that user to return the value is admin, it would tell us sorry, cannot access the private property. We wouldn't get a result back. And we can't set a value to it, either. Now that doesn't mean that we can't access it at all. Because you'll notice that the has\_admin\_access() method that we've written there, it does reference it. It's available for use inside the class code.

That's what private means. We can't access it outside the class, but inside the class it is available to us. Now the same thing is true for our methods. If we were to try and use an instance to call that has admin access method, it wouldn't let us. Because we have declared it as being a private method. You'll see that we have a second method there called access level. That is declared as being a public method. So we can call that one. And it in turn can call a private method, which in turn calls a private property.

There are a few best practices that we want to follow. You want to always use visibility modifiers. Don't depend on the defaults. We were using the defaults before only because we didn't know about them. It's really important to always declare them, even if they're all public. Go ahead and declare them as being public. But you should consider what visibility level you need when you're coding. And by default, you want to try and make properties and methods public only when necessary. If it needs to be accessible from outside the class, then make it public.

Or if it needs to be available to its sub classes, make it protected. But if it's only going to be used for internal purposes, then mark it as being private. It's also a good idea to try and group properties or methods with similar visibility together. So all of your private properties are all in one chunk, it's nice and easy for you to find them and understand that hey, these are the private ones. If you've got them intermixed, it can be a little hard to spot sometimes.

Let's try an example in our project folder. Let's start by going to our class\_methods.php file we created earlier. Let’s call it class\_visibility.php . Alright so, we already saw this before, right? We were able to just work with different, different methods here, call different things. That's all there. Let's now modify this. Instead of using var, let's use public for all these. That's a best practice. So right off the bat, if you spot var, change it to public. And if you spot a function like this, put public in front of the function. Okay, these are all now public, and that's what they were before by default. But now it's explicit. And that's a best practice. So let's try adding a couple of others. We will add methods like in code bellow.

class Student {

public $first\_name;

public $last\_name;

public $country;

protected $registration\_id;

private $ituition = 0.00;

public function full\_name(){

return $this->first\_name. " " . $this->last\_name; ;

}

public function hello\_world(){

return 'Hello world!';

}

protected function hello\_family(){

return 'Hello family!';

}

private function hello\_me(){

return 'Hello me!';

}

}

$student1 = new Student;

$student1->first\_name = 'Lucy';

$student1->last\_name = 'Ricardo';

echo $student1->full\_name() ."<br />";

echo $student1->hello\_world() . "<br />";

echo $student1->hello\_family() . "<br />";

echo $student1->hello\_me() . "<br />";

Alright so, right now let's just try these and see what we come up with.

Error! Call to protected methods student hello family. Alright, we can't call it. Because it's protected. Let's just comment that out. Let's try another one here. Nope. Call to private method student hello me. Right, didn't like that line, either. We are not able to do either one of those. The only one we can do is this hello world.Right? Cause that's the one that's listed as being public.

Now this registration ID and tuition, can do the same thing here. Try it out. Let's just do echo student one. Let's do registration ID. And let's copy that and let's echo the student's tuition.

echo $student1->registration\_id;

echo $student1->tuition;

We'll get what we expect, which is an error. Saying it can't access registration ID because it's protected. And let's try another one here. We'll reload it, can't access the private property tuition. Right? Neither one of those are accessible from outside. But they are accessible from inside.

Just as an example. Let's just create a sub class here. Class part time student. And part time student is going to extend student. Alright now it's a sub class. So it should have the ability to call that hello family method.

class PartTimeStudent extends Student{

public function hello\_parent(){

return $this->hello\_family();

}

$student2 = new PartTimeStudent;

echo $student2->hello\_parent();

So let's make a public method here that we can call from outside to test it. Function hello\_parent(), we'll call it. Cause it's going to be talking to its parent, and it's going to return back to value that it gets from $this->hello\_family();

So hopefully you get the idea of visibility. If we have something that's public, it's accessible from both inside the class, the sub classes, and outside. When we're working with instances. If we have something that's private, it's only inside the class code.

If it's protected, it's accessible inside the classes code as well as its sub classes.

Beware of Overloading

In this part, we'll talk about a feature of PHP called overloading, and we're going learn why overloading can make it confusing when you want to control visibility. First, let me demonstrate the issue. In the last movie, we were working with this file called class\_visibility.php. We'll keep working with that. We have a class definition for Student, and it has a property of tuition, and that has been set to have a visibility of private. And we know that private visibility means that the property is only accessible from inside the class' code.

It's not accessible in any subclass, and it's not accessible when we're working with an instance and calling attributes on that instance. We actually proved that to ourselves because down here, we had PartTimeStudent as a subclass. We created a new instance of PartTimeStudent, and assigned it to student2, and then right here, we had a line where we tested it out. I'm just going to copy this line, come down here and we'll try it again. I'm going to also just put a <br /> tag at the end of it.

echo $student2->tuition . “<br />”;

So here we have echo $student2->tuition. Now this is a private property in the parent class, so it should not be inherited, it should not be available. Let's save it. Let's go back to the browser and just prove that to ourselves. When we reload the page and sure enough, Undefined property. It's not defined. PartTimeStudent does not know anything about tuition. So that's what we would expect. That's how visibility's supposed to work, but watch this. I'm going to add a line here, and let's have $student1-> tuition = 1000, all right? Now, this also should not work, right? It has no idea what this tuition property is.

$student2->tuition =1000;

echo $student2->tuition . “<br />”;

We are trying to set 1000 to something it just told me it doesn't know about. Let's save the file. Let's come back over to browser and let's reload it. Look at that. The error went away, and it worked. We were able to set the value equal to 1000, and we echoed it back to the browser. What's going on here? The reason this happens is because PHP has a feature called **overloading**. Now, if you've worked in other programming languages, overloading has different meanings in other programming languages.

For whatever reason, that's what they decided to call it in PHP. Not a very good name probably, but if you want to look it up in the documentation, that's how you're going to find it. A better name might be something like dynamic properties. What it is is it's how PHP handles calls to properties and methods which do not exist or are not visible.

So here's an example of overloading using a bare object class with no inheritants. We have a class called Product and there's absolutely nothing inside of it, no properties at all.

$p = new Product;

echo $p->name;

//Notice: Undefined property: Product::$name

$p->name = 'Guitar';

echo $p->name;

So we can create a new instance of that Product, and ask it to return the value of that property, name. It comes back and tells me Undefined property, but if I set a value to it, and then return it, then it returns that value. That's because overloading has some default behaviors. The default when we get a value for undefined property is to return a notice to us. The default behavior when we set a value of undefined property is to define the property and set the value, and not return any notice whatsoever.

And of course, what we just saw when you set a value and it gets defined, then when we turn right around and get that value, we don't get that notice either. This can be confusing because private properties may seem to be visible in subclasses, but they're not. We're actually defining a new property in the subclass. We're not seeing that private property that was in the parent class. And because we're adding it in the subclass, it doesn't really work the way we would expect either. For example, let's go up the back here in the parent class Student, and let's add a new method.

public function tuition\_fmt(){

return '$' . $this->tuition;

}

It's going to be a public method, and let's call it tuition\_fmt(). That is, that it's formatted. Thus the fmt is for format, and we're going to return back just a simple dollar sign, and then we're going to call tuition. Now this can call tuition because it's in the same class. So it has the ability to call private $tuition, and that's going to be initialized to mean zero dollars. Just so we can see the difference, let's just set this equal to another value.

Let's say the tuition by default is going to be 500 dollars. Let's come down here, and let's call our new method.

**echo $student2->tuition\_fmt() . "<br />";**

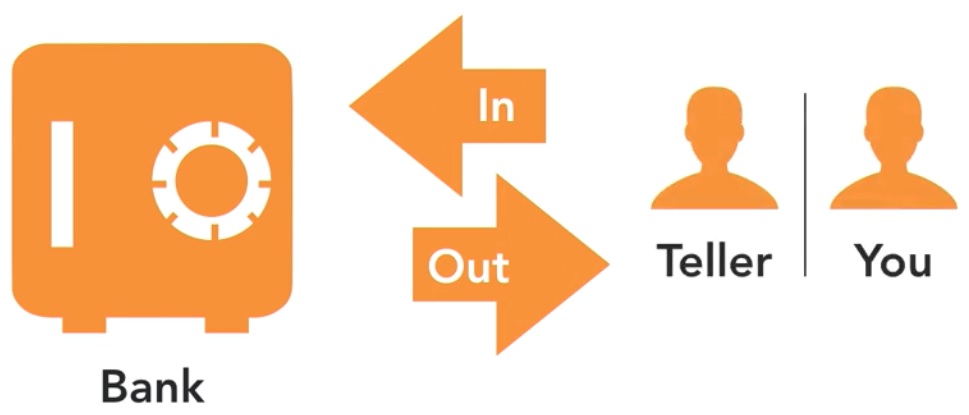
We are going to copy this, going to paste it in, and going to call it tuition\_fmt, and then a set of parentheses because we're making a method call. Okay, so we set the value, we looked at it. Now let's take a look at it formatted with the dollar sign in front of it. Let's come back over, reload the page. Look at that.

The value's equal to 500. Not equal to the value we just set. It just told me in the line right above it that tuition was set to 1000. But because it's adding a new one dynamically, it's different than the one that was there before. So the reference that's being found here is different than the dynamic one that's being added. So overall, it's just all around confusing. Overloading causes that problem, so you need to be aware of it. If you accidentally misspell something, if you accidentally refer to something that's private, that you shouldn't have access to, you're going to get the same behavior.

Setter and getter method

In this aprt we're going to talk about another concept in PHP called setter and getter methods. This folds right in with what we've been talking about visibility. Let's start by using a metaphor. Let's say that we have a bank account and we want to put money into our bank account. So we have the ability to do that. We can put money in the bank and we can take money back out of the bank. Now, in reality, that's not the way it works. We can't just walk into the bank vault at our local branch and put money in and take money out.

Instead, what we have is a teller. The teller's there, you walk up to the desk, you tell the teller you'd like to deposit money, the teller brings up all your account information, they authorize you, and then they take care of putting the money into your bank account. And if you want to get money out, again you talk to the teller, the teller authorizes you, makes sure that you have enough money in your account, that there's no problems, make sure the bank has enough money to pay you, and then they take that money out, they give you your money, and you go your way.



The teller acts as an intermediary for both putting things into the bank and bringing them back out again. That's how setter and getter methods work. The idea is that you would set the property visibility to be private so it's not accessible directly. That would be like our bank account. We can't access it directly; it's private. But, then we can define a method which does have access to that private property and it can set the property's value for us and we can define another method that will get the property's value bank out. See those words, set and get? That's where we get the names, setter and getter methods. It's a very common design pattern.

Here's an example using a class called product. We have made it's name private. So now, when we create a new instance, we are not able to set a value or read a value back from the name property. However, notice that there's also two public functions, one called set\_name() and one called get\_name(). And guess what they do? They set those values for us.

class Product {

private $name;

public function set\_name($value){

return $this->name;

}

public function get\_name(){

return $this->name;

}

}

$p = new Product;

$p->set\_name('Birdhouse');

$p->get\_name();

The first one just simply sets the value's name, the second one just returns the value back to me. So setter and getter methods allow access to what would otherwise be private properties. This is useful because it allows us to regulate access the same way that the teller at the bank is able to regulate our access to the bank account. It's also useful if we have situations where we only want to be able to read properties back or only write those properties. Setter and getter methods are also useful if we want to pre-process values.

That is, we want to take some information, we want to do some processing on it before we put it into our property, or we want to pull the value out of our property and do some processing on it before we return it back to the user. What you want to avoid however, are what are referred to as naive setter and naive getter methods. The example that we just looked at with product and name is an example of a naive setter and getter. This is just simply less efficient and slower than giving us access to the property as a public property. We have zero gain to our code, there's zero benefit for security or anything else like that, it's just simply slower. **So, don't do this version. This is the simplified version and it's what you should not do!**

Let's look a a better example. Let's say that our product class has another private property called price. I'm going to have two public functions, set\_price and get\_price that can interact with that private property.

class Product {

private $price;

public function set\_price($value){

$no\_format = preg\_replace('/[\$,]', '' , $value);

$float = floatval($no\_format);

if($float <= 0) {

trigger\_error('Price must be greater then zero,', E\_USER\_NOTICE);

return;

}

$this->price = $float;

}

public function get\_price(){

return '$' . number\_format($this->price, 2);

}

}

$p = new Product;

$p->set\_price(121,345);

$p->get\_price();

echo $p->get\_price();

The first one, set\_price, is going to take the value and it's not going to assume that it's an integer or a float value, instead it's going to strip out any dollar signs or commas that might be in the value, in case we were given something that was a string, and then it's going to turn it into a floating point number, and it's going to check and see whether or not it's less than or equal to zero, and if it is, it's going to come back and return an error saying this product or price can't be set to a negative value.

Then, once all of that pre-processing is done, it willll actually set the value of the property. And then, when we get the price back, I'm not simply just going to return the price, I'm going to put that dollar sign in front of it, I'm going to convert it to a string and make sure it has two decimal places.

The way you decide to utilize setter and getter methods inside your own projects is going to depend on what you need to accomplish in each project. It's important just to understand the concept. The fundamental idea is that the property remains private but we still have access to it through public methods.

Challenge: Access Control

It's time for another challenge assignment. This time the challenge is to make sure that you understand access control and how to use visibility modifiers inside your objects. There are four parts to this challenge. The first is that we want to add visibility modifiers to our Bicycle class. That's the class that we created in Challenge 01. So the very first thing you want to do, is create a new file called challenge\_03. Then you want to go to that challenge\_01 file and copy the class definition for Bicycle into it.

So that we can start with that. It's a starting point. Then, you're going to set visibility for all of the existing properties and methods. You'll need to consider which of the attributes and methods need to be public and can any of them be made private? And if they do need to be made private, do subclasses still need access to them? That'll be part one of the challenge. The second part of the challenge is to create Unicycle as a subclass of Bicycle. And once you've done that, you want to add the property, $wheels, with default values for each of those classes.

A Bicycle has two wheels and a Unicycle only has one. Then, set the visibility such that $wheels is only available to the code that defines Bicycle and Unicycle. Then you'll define the new method called wheel\_details(), which is going to return a sentence describing the wheels on either the Bicycle or the Unicycle instance that we're working with. So, for example, if we have a Bicycle instance, it would return "It has 2 wheels." If we have a Unicycle instance, it would return "It has 1 wheel." After you've done that, we're going to go and modify the $weight\_kg property that we already had before.

We're going to change $weight\_kg to be a private property and then I want you to define a setter method for it. So we want to create a method that allows you to set a value of that private property and then we're going to create a getter method that'll read that value back. And, not only will it read it back, it will also append "kg" after the number so that we can see that it's a weight in kilograms. We'd already written another method called weight\_lbs(), and that returns the weight in pounds. We're going to just modify that slightly so that it also adds "lbs" after it.

And then it will be clear from the results if we're looking at kilograms or pounds. And, last of all, ask yourself, what bug now exists related to this $weight\_kg property? The last part of the assignment. The third thing that we did created a bug. So, figure out what the bug is and then decide how you would fix it. Once you complete this challenge assignment, meet me in the next movie where I will show the solution that I came up with.

Solution

<?php

class Bicycle {

public $brand;

public $model;

public $year;

public $description = 'Used bicycle';

private $weight\_kg = 0.0;

protected $wheels = 2;

public function name() {

return $this->brand . " " . $this->model . " (" . $this->year . ")";

}

public function wheel\_details() {

$wheel\_string = $this->wheels == 1 ? "1 wheel" : "{$this->wheels} wheels";

return "It has " . $wheel\_string . ".";

}

public function weight\_kg() {

return $this->weight\_kg . ' kg';

}

public function set\_weight\_kg($value) {

$this->weight\_kg = floatval($value);

}

public function weight\_lbs() {

$weight\_lbs = floatval($this->weight\_kg) \* 2.2046226218;

return $weight\_lbs . ' lbs';

}

public function set\_weight\_lbs($value) {

$this->weight\_kg = floatval($value) / 2.2046226218;

}

}

class Unicycle extends Bicycle {

// visibility must match property being overridden

protected $wheels = 1;

}

$trek = new Bicycle;

$trek->brand = 'Trek';

$trek->model = 'Emonda';

$trek->year = '2017';

$uni = new Unicycle;

echo "Bicycle: " . $trek->wheel\_details() . "<br />";

echo "Unicycle: " . $uni->wheel\_details() . "<br />";

echo "<hr />";

echo "Set weight using kg<br />";

$trek->set\_weight\_kg(1);

echo $trek->weight\_kg() . "<br />";

echo $trek->weight\_lbs() . "<br />";

echo "<hr />";

echo "Set weight using lbs<br />";

$trek->set\_weight\_lbs(2);

echo $trek->weight\_kg() . "<br />";

echo $trek->weight\_lbs() . "<br />";

echo "<hr />";

// Will this work?

// echo "Set weight for Unicycle<br />";

// $uni->set\_weight\_kg(1);

// echo $uni->weight\_kg() . "<br />";

// echo $uni->weight\_lbs() . "<br />";

Static Properties and Methods

In this chapter, we will learn about static properties and methods. Static properties and methods are for behaviors that are related to the class generally, or the class as a whole. These are behaviors that are not tied to any particular instance of the class. For this reason, they're often referred to as class properties or class methods. We're going to have access to them by calling them directly on the class without having to have an instance of the class first. In order to define them, we're going to use the keyword static. So for example, if we have a student class, we can have a static property for grades by putting the static keyword in front of it.

class Student {

static $grades = ['Freshman', 'Softmore', 'Junior', 'Senior'];

static function motto() {

echo 'To learn PHP OOOP!';

}

}

In the same place where we put our visibility modifiers in the last chapter, and then we can have a static method called motto by putting static in front of the word function. The idea is the grades property is holding data that is related to the class as a whole. It's not specific to any one instance of the class. Presumably if I had an instance of student, that student would be one of these grades, freshman, sophomore, junior, senior, but it makes sense that we would have an idea of all grades that are possible stored in the class as well. That's knowledge that the class has about the possible choices, even though an instance is only going to pick one of them. Even though it's a little silly of an example, the motto is going to be something that belongs to the class so that all students have the same motto, it's not a motto that's unique to each and every student.

Once we've defined our static properties and methods, we're going to need to use a different syntax in order to reference them. We can't use the arrow notation we've been using when working with instances. Instead we're going to use the class name, followed by two colons, and then the name of the property or the method.

Student::$grades // Property

Student::moto() // Method

Notice that we have a dollar sign in front of the property name. That's very important. That's different from what we were working with arrow notation. When we had arrow notation, we omitted the dollar sign, but when we're working with static properties, we must include it. In previous chapters, our method code was able to make use of a special pseudo variable called this. Our static methods cannot use it, and that makes sense because this refers to this instance, and we don't have an instance here.

We haven't created an instance. We're talking to the class directly, so it can't refer to a specific instance using the $this variable. Instead, if a static method wants to refer to the current class, it can either do it by using the class name and two colons, or, even better, it can use the special keyword self, so self, colon, colon, and then the property or the method. It's considered a best practice to use the class name when you're outside the class and to use self when you're writing code inside the class.

Part of the reason that's important is that we can also combine the static modifier with our visibility modifiers, and if we make a property or a method public, it's going to be available both to student, colon, colon, and to self, colon, colon, but if we make it private, then we can call it from inside the class using self, but we will not be able to call it from outside the class using student. Here's an example that demonstrates that. Notice that I'm using my visibility modifiers with my static modifier.

The order that they go in doesn't matter. I like to put my visibility modifiers first. You can see that grades has been marked as being public, so therefore we're able to reference it outside the class definition and ask for the grades at index position zero and it returns back freshman. Total students, though, has been marked as being private, so when we try and make reference to that from outside the class using student, colon, colon, it comes back with an error saying we cannot access this private property. However, I can have a public static method called count, which can reference it by using that self keyword, and then I can call that public static method in order to get back the value.

So far, all of our code has been talking directly to the class without instantiating any objects. If we did instantiate an object of this class, it's important to realize that we cannot access the static properties from that instance. The static methods are accessible from an instance, but it's a quirk of PHP and you shouldn't do it. It's considered a bad practice. PHP 5 will do it, but will also return a warning. PHP 7 will do it, but returns a deprecation warning, which essentially says you can do this, but support for it may go away in the future, so beware.

Let's try an example. Inside our folder, in the exercise files, we've included a new file called static\_modifiers.php, where we essentially just define the student class that we were looking at in the slides. We gave it a public static property for grades, and we gave it a private static property for total students. You can see that we've got a public static method for motto() and a public static method for count(). Let's just try some of those out real quick.

class Student {

public static $grades = ['Freshman', 'Softmore', 'Junior', 'Senior'];

private static $total\_students = 0;

public static function moto(){

return 'To learn PHP OOP';

}

public static function count(){

return self::$total\_students;

}

public static function add\_students(){

self::$total\_students++;

}

}

echo Student::$grades[0] . "<br />";

echo Student::moto() . "<br />";

// echo Student::$total\_students . <br />; //Error

echo Student::count() . "<br />";

Student::add\_students();

echo Student::count() . "<br />";

Let's start by just trying to reference those. So let's do echo, we're going to call this student, we're going to use colon, colon, and then a dollar sign. Very important. We're going to ask it for the grades that index position zero. We'll put a BR tag after it just so that it goes to a new line, and let's also just copy that, but instead of the grades property, let's call the motto() method. Okay, so let's save that, go back over to the browser, and let's try it.

Going to be static\_modifiers.php. There we go, freshman, and to learn PHP OOP. So there you go. We get back both those values. We were able to use double colon notation to reference both of those.

Now let's try another one here where we try and go directly to dollar sign total students. Let's see what that gives us back. Let's reload our page. We get back an error. It's a private property and we can't access it, alright, we knew that, let's mark here, error comment, and I'll just comment that line out. Just grab this line, copy it.

Instead we asked for count(). It can access it. Even though it's a private property, we're using self and it's able to get there, so let's ask for that value. It comes back and says it's zero. Now, there's also another method here called add\_student(). Maybe every time we create a new student, we are going to increment this value of total students, and then we can keep a running count of the students, so it's going to call **self::total students++** we'll increment it, so now we can call that. It's public. We can certainly call it.

And then let's ask for the count again. Save it, come back over and reload the page, and there we go, zero and one. So that's how static or class property and methods work. Static properties and methods are an important tool in your PHP object oriented programming.

Inherited static behaviors

In this part, we're going to talk about what happens when you mix object inheritance together with static properties and methods. The short version is that static properties and methods are inherited. So if we define static properties and methods on a class, then we define a subclass, that subclass is going to have the same static properties and methods. It's going to follow the same visibility modifier rules that we've been working with so far. So, if something is defined as being public or protected, then it's available in the subclass, but of course if we define this being private, then it's not inherited and not available.

It's only in the parent class. However, there's one thing that you need to watch out for, which is that inherited static properties are shared variable. That means that changes to the parent value change the subclass values, and changes to a subclass value is going to change the parent value. Let me demonstrate. Let's say I have a class called Student, and it has a public static property for grades, and then I have a subclass of student called PartTimeStudent. PartTimeStudent is going to still have access to that property, and we can see that in the first example below the class definitions, where we ask it for the value that's at position zero in the array, and it returns Freshman to us.

But here's the tricky part. What happens when I take that subclass PartTimeStudent, and I tell it to append a new value onto the end of the array, so we add ‘Alumni’. Notice in the last example, when we ask it to output a list of all the grades and to put them together with commas, we are asking it on the parent class, Student, not the subclass that we added it on, and we get back an array that includes Alumni at the end. Changes that we made to the subclass also happen on the parent class, because it's a shared variable.

This is important to understand because I think it trips up every single PHP developer at some point. For example, here's another common use case. Let's say that we have a class of Students, and we're keeping track of the total number of Student objects that we've created, and we're doing that in a static variable called total\_students. I've just made it a public static variable so I can quickly access it. Now I have subclass called PartTimeStudent. So let's say that my code is going along, and it's creating new Student instances. And every time it does, I'm incrementing the value of total students.

So we call Student:: total\_students++, and it adds one to that value. We'll do that a couple times. Then suddenly we created a PartTimeStudent:: total\_students++ and we increment its value as well. We might think that we are adding that to a unique variable that we are going to end up with two different accounts, account of Students and account of PartTimeStudent, but we are not, because it's a shared variable. So the result at the end is going to be four for both of them. Let's try adding this to our project folder.

Let's go into static\_modifiers.php, and copy whole code to new file static\_inheritance.php. Let's add a little comment here that just says Static properties and methods are inherited, and let's create a subclass, class PartTimeStudent extends Student. And we won't put anything inside of there, but we know now that's going to inherit everything that was inside student.

<?php

class Student {

public static $grades = ['Freshman', 'Softmore', 'Junior', 'Senior'];

private static $total\_students = 0;

public static function moto(){

return 'To learn PHP OOP';

}

public static function count(){

return self::$total\_students;

}

public static function add\_students(){

self::$total\_students++;

}

}

echo Student::$grades[0] . "<br />";

echo Student::moto() . "<br />";

// echo Student::$total\_students . <br />; //Error

echo Student::count() . "<br />";

Student::add\_students();

echo Student::count() . "<br />";

// Static iproperties and methods inheritance

class PartTimeStudent extends Student {

}

echo PartTimeStudent::$grades[0] . "<br />";

echo PartTimeStudent::moto() . "<br />";

//Changes are shared too!

PartTimeStudent::$grades[] = 'Alumni';

echo implode(', ', Student::$grades) . "<br />";

Student::add\_students();

Student::add\_students();

Student::add\_students();

PartTimeStudent::add\_students();

echo Student::count() . "<br />";

echo PartTimeStudent::count() . "<br />";

That count is the same for both the Student and the PartTimeStudent, and the reason is because this private static variable is being shared between the two of them.

Class constants

In this part, we're going to learn about class constants. You've probably been using constants before in PHP. They're used for values which don't change. Well, class constants are just for class values which don't change. The way that we define them in our class is we use the keyword **const**, followed by the name in **all capital letters**, just like we do with normal constants. The value that we set a constant to can be something simple like a string, or a number, or an array, or it can contain mathematical expressions.

If you're using a new version of PHP that is something later than 7.1, then you can also put a visibility modifier in front of that const keyword in order to make something public, protected, or private. Prior to 7.1 and by default, it's going to be set to public. So constants are available publicly outside the class, as well as inside the class. The way that you reference those constants is using **ClassName::** followed by the name of the constant or **self::** followed by the name of the constant.

So they are extremely similar to static properties. The difference is that a static property has values that can be changed, while constant values cannot. Here's an example of a class or clock, and you can see that it has a constant defined for DAY\_IN\_SECONDS, all capitals. There's no dollar sign in front of it when we have a constant. We have a variable, we put the dollar sign there. Having it all caps with no dollar sign tells us this is a constant. And I'm using a visibility modifier in front of it to make it public, and then I'm setting it equal to a mathematical expression.

class Clock {

const DAY\_IN\_SECONDS = 60 \* 60 \* 24 ;

public function tomorrow(){

return time() + self::DAY\_IN\_SECONDS;

}

}

echo Clock::DAY\_IN\_SECONDS . "<br />";

$clock = new Clock;

echo $clock->tomorrow() . "<br />";

What is a day in seconds? Well, there's 60 seconds in a minute, 60 minutes in an hour, and 24 hours in a day. So this tells me the number of day in seconds. You'll see below the class definition that I'm able to reference the constant by calling Clock::DAY\_IN\_SECONDS, and it returns the value back because it's public. I also can refer to it using self, and you see that I do that inside the method for tomorrow. This is not a static method. It's just a normal method called tomorrow, and it's going to return back the current time plus whatever day is in seconds, and that'll give me the time tomorrow.

So at the bottom, you can see that we have created a new instance of the clock, and then we call its tomorrow() method, and it returns back the time tomorrow. Let's put the same example in our project folder to try it. So let's add a new file to our sandbox. Let's call new file **class\_constants.php**, and we'll put up our php tags, and let's try that example that we just had where we had a class called clock.

And inside, we're going to have a public constant called DAY\_IN\_SECONDS. That's going to be equal to 60 times 60 times 24. Okay, now let's come down here and let's try this out. Let's have echo Clock:: and then I'm just going to copy this, paste it right there, and then at the end, let's put our br tags, so it will be ready for a new line after that. Let's just try that much out.

After that we will create method and return Linux time for tomorrow.

public function tomorrow(){

return time() + self::DAY\_IN\_SECONDS;

}

So let’s create an instance (object) of class Clock and refer to method tomorrow().

echo Clock::DAY\_IN\_SECONDS . "<br />";

$clock = new Clock;

echo $clock->tomorrow() . "<br />";

This is how we can use Class constants.

Refer to the parent class

In this part we want to look at how subclasses are able to refer to behaviors which are in their parent class. So far when working with our static properties and methods we've seen a couple different ways that we can refer to them. The first is that we can use the class name followed by the scope resolution operator, and then either the property or the method name. We also saw that if we're working inside the class definition, we have another option, which is the keyword self. So we can swap out the class name for this keyword self.

But when we want to refer to the parent class we also have two options. The first is that we can use the actual class name, just like we did in that first example, or we have a keyword that we can swap in, and this is considered a best practice and a better way to do it, just like self is. If we're inside the definition of the class it's better to refer to the parent class as parent, and not a refer to whatever class is listed right after the keyword extends. That's the class that we're talking about.

So it's going to ask the parent for its property or method. Now this is only going to be for static properties and methods, not for instances. Now actually the same rules apply as with self, whereas it does work for methods, but you'll get a depracation warning saying, "Please don't do this because this feature "may go away in the future." So it's a best practice not to do it even though technically you can! And it makes sense that we wouldn't do that because self and parent are replacements for the class name!

So, you would use parent followed by either the property or the method name. But the thing is, it's not actually needed for the static properties. You remember we talked about that those static properties are already shared with all the subclasses, it's a common variable that they're both using. So, we don't need to use parent followed by a property name, because self followed by the property name refers to the same variable. Where it's useful is for calling static methods, especially after we've overwritten them in the subclass.

Let's look at the two most common cases where you might do this, because they're going to be very instructive. The first example shows how you extend the functionality of the parent static method. Here we have a class called chef and it has a public static function called make\_dinner() in it. And it's just going to echo back the phrase, "cook food". Now, we have a subclass called AmateurChef and it's going to override make\_dinner(). So the amateur chef's going to make dinner in a different way than the chef did. But notice that it still calls parent make dinner inside of it.

class Chef {

public static function make\_dinner(){

echo 'Cook food!';

}

}

class AmateurChef extends Chef{

public static function make\_dinner() {

echo 'Read recipe';

parent::make\_dinner();

echo 'clean mess';

}

}

Essentially what we are doing is that we are saying: "Hey, I want to do the same thing as my parent did " , but I have some business that I want to take care of, "either right before it, right after it, or both." We still want that parent's method, we don't want to have to rewrite it. We just want to wrap additional code around it, and this does that. So then if we call chef make dinner, it responds with, "cook food". But if we ask the amateur chef to make dinner, you'll see that we get back, "read recipe, cook food, clean up mess". This isn't a particularly real world example, but I hope that it does make clear this idea that you're able to do actions before and after what the parent method would do.

And where this really pays benefits, is if we decided to add another step to chef make dinner, the amateur chef would get that change too. The second common usage is to override the parent method but then to fall back to the parent if something goes wrong, or a condition is not met.

So here for example: We have a class called Image, it has a public static property for resizing enabled, which is currently set to true by default, and we have a public static method for geometry which returns just the string 800 by 600.

class Image {

public static $resizing\_enabled = true;

public static function geometry(){

echo '800x600';

}

}

class ProfileImages extends Image{

public static function geometry() {

if(self::$resizing\_enabled){

echo '100x100';

}else{

parent::geometry();

}

}

}

Now we have a subclass called ProfileImage which is going to override geometry, and it's going to attempt to provide its own answer. Its own answer would be 100x100 but notice that it only returns that answer if resizing is enabled. If not, it's going to fall back to what the parent had. So it basically says, "Try to do this yourself " , but if for some reason you can't do it, something goes wrong, or a condition is not met, "let's fall back to whatever our parent said " and see if the parent can handle it instead." Take a moment and also notice that when we are referring to resizing enabled, we are not calling parent resizing enabled, even though that's defined up in the parent class. We don't have to because that's a shared property, so it's already inherited into the profile image class. So when we try it out we have our image geometry, of course we get back 800 by 600. If we ask for the profile image geometry we would get 100 by 100 as a default, because resizing is enabled. But if we then turn resizing off, notice here that we are doing that in the parent class. It doesn't matter, it would work the same way if we did it with profile image. Then when we call profile image geometry, it's going to try and handle it itself by calling its own overwritten geometry method, but since resizing isn't enabled it's going to fall back to the parent.

So there are many reasons that you might call the parent but these are the two most common usages. Either to extend a current method with something either before it, or after it, or both; or to fall back to it if something goes wrong, or a condition is not met.

Let's try these examples in our project folder just so we can see them. Let’s write our code inside the file refer\_parent.php. So you can see code like this, same as we had in examples.

<?php

class Chef {

public static function make\_dinner(){

echo 'Cook food! . <br />';

}

}

class AmateurChef extends Chef{

public static function make\_dinner() {

echo 'Read recipe . <br />';

parent::make\_dinner();

echo 'Clean up mess. <br />';

}

}

echo 'Chef: <br />';

Chef::make\_dinner();

echo "<br />";

echo 'Amateur Chef: <br />';

AmateurChef::make\_dinner();

echo "<hr />";

class Image {

public static $resizing\_enabled = true;

public static function geometry(){

echo '800x600';

}

}

class ProfileImages extends Image{

public static function geometry() {

if(self::$resizing\_enabled){

echo '100x100';

}else{

parent::geometry();

}

}

}

echo Image::geometry() . "<br />";

echo ProfileImages::geometry() . "<br />";

echo Image::$resizing\_enabled = false;

echo ProfileImages::geometry() . "<br />";

Late Static bindings

In this part we're going to talk about a concept called late static bindings. And the best way to understand what they are is to first understand the problem that they solve. In challenge03.php, we defined a class for bicycle, and a subclass of unicycle, we did that before we knew about static properties and methods. Now we know about them, it might occur to us that the wheels property and the wheel details method are good candidates to make static. They aren't specific to any one instance, but they're behaviors of the bicycle and unicycle classes generally.

So we set about to do that, we add the **static** keyword in front of the wheels property. And we add the **static** keyword before the definition of the wheel details method. Then, inside the wheel details method, we also need to change the first line. Instead of using the dollar sign this with arrow notation, we want to use **self::**. That's going to refer to that property statically. Then, once we have that defined, we go about calling the static method on the bicycle class. We have the Bicycle::$wheel\_details(). It returns back to us, it has two wheels. That's great, it works exactly like we expected.

So now we try it on our unicycle class. And it returns back, it has two wheels. Wait a minute, what's going on here? We inherited the behaviors from bicycle but we also overrode the default value of wheels. Why is it not picking up our change? Why is it still telling us that there's two wheels instead of one? The reason why is because PHP by default uses something called static bindings.

With static bindings, static references are resolved using the class in which the method was defined. Self is defined inside the bicycle class. So it it references bicycle. The reason PHP does this, is it allows PHP to pre-process the class definitions, and have them ready to go so that it's much faster. But the down side is that it prevents inheriting those static references. Because no matter what we do, self has already been predetermined to refer to the bicycle class. Even though unicycle inherits it, PHP sees the word self and it thinks bicycle. The solution to this is something called late static bindings. If the other ones are just static bindings, late static bindings resolve later. That is that they're resolved using the class that's called at runtime. It waits to decide what self means until we actually get to that code and call it. This was added in PHP 5.3, so it's available in all versions of PHP after that.

In order to use it, all we have to do is replace that keyword **self**, with the keyword **static**. Other than that, it's the exact same concept as static references. It just allows us to inherit those references. So when we define the bicycle and unicycle classes, we still define the properties and methods as being static, just like what we did before. The one difference is you'll notice that now,we have changed the keyword self that comes right before that scope resolution operator, and we have made it into the keyword **static**.

It happens to be the same keyword that we used before, it's not the same thing. In this context we're saying, use the late static bindings, get this value at runtime, don't free-compute it. Therefore, when we ask for bicycle wheel details, it calculates static as being bicycle. Because it's in the bicycle class when I call it. But when I call the unicycle class, at runtime it goes through the wheels details method, and it sees that it's been inherited, and so static refers to unicycle, and it returns then back that value of one, and it says it has one wheel.

There's also a function for late static bindings that we should know about. You recall that earlier we learnt about a built-in PHP function called get\_class, when you pass an object as an argument, it'll tell us what class that object is. There's another way we can use get\_class, which is to put no object in at all. And when it's inside our code, we call get\_class, it'll tell us what the current class is. Now as you might guess, that uses static bindings to do that. So if that gets inherited inside a static method, it's going to say that it's the class that it was originally defined in.

There's another method called get\_called class, it won't take any arguments, but when it's inside a method, it'll return the late static binding class. The class at runtime. Let's do a demonstration of late static bindings in our project folder. Let’s create a file called late\_static\_bindings.php and

class Sofa {

protected static $identity = 'Sofa class';

public static function identity\_test(){

echo 'self: ' . self::$identity . "<br />";

echo 'static: ' . static::$identity . "<br />";

echo 'get\_class: ' . get\_class() . "<br />";

echo 'get\_called\_class: ' . get\_called\_class() . "<br />";

}

}

class Loveseat extends Sofa {

protected static $identity = 'Loveseeat class';

}

Sofa::identity\_test();

echo "<hr />";

Loveseat::identity\_test();

We have got a protected static property which we have called identity, and it's just going to tell us that it's the sofa class. And then I'm overriding that in the subclass, so identity here is equal to loveseat class. I'm doing this so I can call the identity test method. And it will return back some information. We'll see the results of what happens if we ask for that value, that property, using self. And what happens when we ask for that property using static. And then we can compare, we can see what the differences between them. And then I'm also going to call get class and get called class, so we can see the difference that each of those returns.

We can see that we should use late static binding if we want to address direcrly to property in subclass.

Spend some time playing with it and defining some new classes of your own 'til you get a feel for it. In next part we can see example of applying this kind of approach.

Challenge: Static references

It's time for another challenge assignment. This challenge relates to everything we've done in this chapter on static properties and methods. In order to do this challenge, we're going to be reusing the Bicycle and the Unicycle classes that we've created in the last challenge, so the very first thing you want to do is create a new file called challenge\_04.php, then you'll go and open up challenge\_03.php and copy those classes for Bicycle and Unicycle into the new file, then you'll be ready for the challenge.

There are a couple of different parts to this challenge. The first is that I want you to add a static property called $instance\_count. The idea is that it'll be used to store a count of the number of instances that have been created. Once we have that static property, we can create a static method called create(). The idea behind create() is that create() is going to create and return a new instance to us, so it'll do two things. It's going to add one to that $instance\_count variable and it's going to create and return a new instance of the class.

Now, make sure it's the correct class. If it's called on Bicycle, it should create and return a Bicycle instance. If create() is called on Unicycle, it should create a Unicycle instance. It's a little bit tricky, so spend some time thinking about that. And then, of course, it's a best practice to make sure we use visibility modifiers on both $instance\_count and create(). The second part of the challenge relates to constants. I want you to define a constant for storing an array of categories. The idea is that these are categories that a bicycle could belong to.

So, for example, Road, Mountain, Hybrid, Cruiser, City, and BMX. If you're using PHP 7.1 or later, you'll also want to set a visibility monitor on that constant and you can set it to public so that we can see it when we're working with a instance. If you're using PHP before 7.1, you don't set the modifier, it'll automatically be public. Then, add a $category property for instances so they can store their category. That way, one instance of a bicycle can be a mountain bike, while another instance of a bicycle can be a hybrid bike.

For the third part of the challenge, I want you to make the property $wheels into a static property. We've already defined it, but now it needs to become a static property and also make wheel\_details() into a static method. Once you've done that, then make sure that both Bicycle and Unicycle return the expected results for wheel\_details(). Unicycle's the one to really watch out for, that's a little tricky because it's a subclass. And then, last of all, we have an extra credit assignment for you. If you feel like you have a good handle on this, try to create a method in Unicycle which extends a method in Bicycle by executing code either before or after the Bicycle's method, and then, create another method in Unicycle which overrides a method in Bicycle, but will fall back to the original method if a condition is not met.

What the behavior is in the parent or the subclass is really up to you, get creative. The point is to try and implement these two design patterns. If you need help, you can go back to the movie on referring to the parent class to get some ideas. Spend a few minutes working on this challenge and make sure that you understand static properties and static methods. In the next movie, I'll show you the solution that I came up with.

Solution: Static references

<?php

class Bicycle {

public static $instance\_count = 0;

public $brand;

public $model;

public $year;

public $category;

public $description = 'Used bicycle';

protected $weight\_kg = 0.0;

protected static $wheels = 2;

const CATEGORIES = ['Road', 'Mountain', 'Hybrid', 'Cruiser', 'City', 'BMX'];

public static function create() {

$class\_name = get\_called\_class(); // must retrieve string first

$obj = new $class\_name; // "new" expects a class or a string

// $obj = new static // self & static work here too!

self::$instance\_count++;

return $obj;

}

public function name() {

return $this->brand . " " . $this->model . " (" . $this->year . ")";

}

public static function wheel\_details() {

$wheel\_string = static::$wheels == 1 ? "1 wheel" : static::$wheels . " wheels";

return "It has " . $wheel\_string . ".";

}

public function weight\_kg() {

return $this->weight\_kg . ' kg';

}

public function set\_weight\_kg($value) {

$this->weight\_kg = floatval($value);

}

public function weight\_lbs() {

$weight\_lbs = floatval($this->weight\_kg) \* 2.2046226218;

return $weight\_lbs . ' lbs';

}

public function set\_weight\_lbs($value) {

$this->weight\_kg = floatval($value) / 2.2046226218;

}

}

class Unicycle extends Bicycle {

// visibility must match property being overridden

protected static $wheels = 1;

public function bug\_test() {

return $this->weight\_kg;

}

}

$trek = new Bicycle;

$trek->brand = 'Trek';

$trek->model = 'Emonda';

$trek->year = '2017';

echo 'Bicycle count: ' . Bicycle::$instance\_count . '<br />';

echo 'Unicycle count: ' . Unicycle::$instance\_count . '<br />';

$bike = Bicycle::create();

$uni = Unicycle::create();

echo 'Bicycle count: ' . Bicycle::$instance\_count . '<br />';

echo 'Unicycle count: ' . Unicycle::$instance\_count . '<br />';

echo "<hr />";

echo 'Categories: ' . implode(', ', Bicycle::CATEGORIES) . '<br />';

$trek->category = Bicycle::CATEGORIES[0];

echo 'Category: ' . $trek->category . '<br />';

echo "<hr />";

echo "Bicycle: " . Bicycle::wheel\_details() . "<br />";

echo "Unicycle: " . Unicycle::wheel\_details() . "<br />";