**OOP Fundamentals PHP**

**Overview**

**OOP (Object Oriented Programming)**is something that has revolutionized the way that programming is done today. If used correctly it can save you loads of time. It will also help you to avoid repeating code to solve the same simple problem and make maintaining your code easy and effective. In this chapter, you'll learn how to use and implement OOP.

Understanding object-oriented programming principles is the key to making your code more organized and modular. This chapter is designed to teach object-oriented programming principles to help you better understand how to organize your code in the future. Get ready!

**Objectives**

* Understand what objects and classes are and why they are useful.
* Understand how to create classes and objects in PHP
* Understand more advanced OOP concepts (inheritance, private vs public, methods, etc.)
* Understand the difference between procedural and OOP code.

Here is an overview of the topics that will be covered in this chapter:

1. **Objects and Classes**
2. **Class Structure**
3. **Instantiation**
4. **Properties**
5. **Methods**
6. **Construct**

# Text Editor, Terminal, and You

It's crazy what we can do as developers with just a terminal and a text editor. From a blank page, you create life. You write the rules of the universe you create with your code. Object-oriented programming really helps us visualize the power that we have as developers. With object-oriented programming, you can create different species that interact in the world that you create.

Okay, enough of high-level talk, let's create our world! Our world will consist of humans, but not just normal humans. There are three main clans between the humans: Wizards, Ninjas, and Samurai. We'll get to them in the next chapter. They are all Humans, but they are from different subclasses. We will just focus on creating instances of Humans in this chapter, and in the next chapter we will create the subclasses of Humans.

## Humans

Note: The following code is the finished implementation of our blueprint for creating instances of Humans. We will build this together throughout the tab so don't worry if you don't understand every line of code.

Let's say we want our world to be lively and we want to create a lot of instances of humans. We can start from scratch and manually create each Human, but this clearly isn't going to scale. It would be awesome if we could have some sort of factory where we can provide a blueprint for creating a Human. What attributes and actions are common across all instances of Humans?

One thing is for sure, every instance of a Human can be unique. Everyone's unique. Don't worry, having a blueprint doesn't mean that each instance of a Human is not going to be unique. The blueprint will only provide the basic structure that all Humans share, such as a name. This will allow us to provide a unique name for each Human but at the same time all Humans will have an attribute of name. Don't worry if you don't understand the code yet!

This is the finalized code of the class Human that we are going to write incrementally throughout the following tabs:

*class* Human {

public $health;

public $clan;

public $strength = 3;

public $intelligence = 3;

public $stealth = 3;

public *function* \_\_construct()

{

echo "I am alive";

$this->health = 100;

}

public *function* \_\_get($property)

{

if (property\_exists($this, $property))

{

return $this->property;

}

}

public *function* \_\_set($property, $value)

{

if (property\_exists($this, $property))

{

$this->$property = $value;

}

return $this;

}

public *function* trashTalk()

{

echo "You want a piece of me?";

}

public *function* attack($human)

{

$this->trashTalk();

$luck = rand(0, 100);

if($luck \* $this->intelligence > 1000)

{

if($luck > $human->stealth)

{

$human->health -= $this->strength;

return true;

}

else

{

return false;

}

}

else

{

return false;

}

}

}

This is going to be the blueprint to create instances of our Human class. All instances of Human in our world are going to have these following properties: $health, $type, $strength, $intelligence, and $stealth.

You might notice that the $strength, $intelligence, and $stealth come with a default value of 3. Notice that these are just variables. You can think of these **properties**, also known as **attributes**, as columns in a table. If we have a users table, the columns first\_name and last\_name are the *attributes*. In fact, this is what an Object Relational Mapper does. It treats each row of the table as one instance of the object that is held by the collection or table.

# Class Structure

You can create a **class** in PHP starting with the keyword class and the name of the class followed by opening and closing curly braces ({ }). The convention in PHP is to name your classes using UpperCamelCase. Instead of using all lowercase and separating words with an underscore, we start with a capital letter and capitalize any new word that is added.

*class* MyFirstClass

{

}

## Instantiation

Now with this blueprint, we can create **instances**that are of this class. This class is the blueprint for creating empty objects. We will learn how to make more interesting blueprints for objects that we will instantiate in the later tabs, but for now, let's bring an object to life with the following code:

$bobby = new MyFirstClass();

$robby = new MyFirstClass();

We just created two different instances of the class MyFirstClass. It happens when we use the keyword **new**to create a new object of the specified class. You can use var\_dump() to see what is contained inside your object.

**We can create as many instances of the class as we want simultaneously.** We gave the first instance a nickname of $bobby and we gave the second instance a nickname of $robby.

var\_dump($bobby);

var\_dump($robby);

Now let's create more interesting objects...

# Properties (Attributes)

You can easily add a **property** to a class similar to the way you would create variables in PHP. The public keyword, which is an **access modifier,** determines the visibility of the variable and we have the option to set that variable to a default value, although this isn't required.

*class* MyFirstClass

{

public $property1 = "I am the first property! Woohoo";

}

We can now see that the property is visible when we output our object!

$obj = new MyFirstClass();

var\_dump($obj);

You can reference that property from the creation of a new object. Using PHP object notation, you can access a method or property using the arrow (- >) off of the creation of an object. Notice how you can omit the **$** when referring to a property.

echo $obj->property1;

# Methods

Methods are just functions contained inside a class. We will be creating **instance methods**, which means functions that are available to the instances of the specified class.  You declare them the same way you would declare a function.

## Adding a Method To Your Class

We create our method the same way we would create a function, but we give the function an **access modifier**. This will again determine the visibility of our method, which we will talk more about later.

In this example, we use what are called ***getter***and**setter** methods. They are just methods that are responsible for getting and setting the value of a particular property, but it is fairly common to do this inside a class. It is considered bad practice to refer to a property directly from an object instance (**$obj->property1**) as it can lead to problems later. For this reason, we will use getter and setter methods instead.

To reference a particular property or method within the class for the **current object instance**, PHP has a special keyword it uses called  **$this**. **$this is referring to the current object instance so you can access properties and methods per object instance.**

*class* MyFirstClass

{

public $property1 = "I am the first property! Woohoo";

public *function* set\_property1($property1)

{

$this->property1 = $property1;

}

public *function* get\_property1()

{

return $this->property1;

}

}

$obj = new MyFirstClass();

echo $obj->get\_property1();

$obj->set\_property1('New value for property1');

echo $obj->get\_property1();

## Creating Multiple Objects

The power of OOP doesn't become apparent until you are able to create multiple instances of the same class.

*class* MyFirstClass

{

public $property1 = "I am the first property! Woohoo";

public *function* set\_property1($property1)

{

$this->property1 = $property1;

}

public *function* get\_property1()

{

return $this->property1;

}

}

$obj1 = new MyFirstClass();

$obj2 = new MyFirstClass();

echo $obj1->get\_property1();

echo $obj2->get\_property1();

$obj1->set\_property1('New value for property1 for the first instance');

$obj2->set\_property1('New value for property1 for the second instance');

echo $obj1->get\_property1();

echo $obj2->get\_property1();

# Construct

PHP has a number of reserved method names that start with two underscores, known as **magic methods**, that will be called in certain circumstances within a class. The most common one is the \_\_construct() method. The \_\_construct() is a special function within a class that gets called for every new instance of a class. We didn't have to define the \_\_construct() method in our previous implementations of our class. We only have to implement this method if we want to do some kind of custom set-up before the instantiation of an object.

Let's say we wanted the instances of our class to echo out a special message every time it comes to life. We can do this by modifying our previous class implementation to this:

*class* MyFirstClass

{

public $property1 = "I am the first property! Woohoo";

public *function* \_\_construct()

{

echo "I get called for each instance of this class!";

}

public *function* set\_property1($property1)

{

$this->property1 = $property1;

}

public *function* get\_property1()

{

return $this->property1;

}

}

$obj1 = new MyFirstClass(); // runs the \_\_construct function immediately

$obj2 = new MyFirstClass(); // runs the \_\_construct function immediately

echo $obj1->get\_property1();

echo $obj2->get\_property1();

$obj1->set\_property1('New value for property1 for the first instance');

$obj2->set\_property1('New value for property1 for the second instance');

echo $obj1->get\_property1();

echo $obj2->get\_property1();

We can also pass in variables to the constructor that we pass into the creation of the object.

*class* MyFirstClass

{

public $property1 = "I am the first property! Woohoo";

public *function* \_\_construct($instance)

{

echo "I am getting called for object: ".$instance;

}

public *function* set\_property1($property1)

{

$this->property1 = $property1;

}

public *function* get\_property1()

{

return $this->property1;

}

}

$obj1 = new MyFirstClass

copy

('instance one'); // param will be passed into the \_\_construct

$obj2 = new MyFirstClass('instance two'); // param will be passed into the \_\_construct

echo $obj1->get\_property1();

echo $obj2->get\_property1();

$obj1->set\_property1('New value for property1 for the first instance');

$obj2->set\_property1('New value for property1 for the second instance');

echo $obj1->get\_property1();

echo $obj2->get\_property1();

# class Human

Now that we learned how to instantiate new objects from a class, how to add properties, and how to add methods, we can create instances of **Humans** in our world!

Let's start by first adding the properties that instances of the class Human will have and then start adding on some methods. How boring would it be if the instances of our Human can't do anything?

**Properties** are variables that an instance of an object keeps track of. In our blueprint, we declare what properties or attributes every instance of this class is going to have. Think back to what you know about MySQL. When we pull out a user from our database, we have columns of different key-pair values. A user has a name, an email address, a password, etc. The same for our instances! Check out the example below:

*class* Human {

public $health;

public $clan;

public $strength = 3;

public $intelligence = 3;

public $stealth = 3;

public *function* \_\_construct() {

echo "I am alive";

$this->health = 100;

}

}

In our world, a Human class is going to have these following attributes: $health, $clan, $strength, $intelligence, and $stealth. These are the variables that a particular instance of this class is going to keep track of.

We have also declared a function within the scope of the class declaration. This is what's called a **method**, a function that an instance of this class can call. In fact, this method is a special method that all objects have. This is the magic method that will automatically get run every time you create a new instance of this class! Therefore, this is generally the area where you want to do some set up before you launch the object into the world. In this class, every time we create a new instance of this class, it is going to echo "I am alive" and then will set an initial value of 100 to the $health property. This method will get run when we instantiate new objects with the following code:

$julius = new Human();

$clarkson = new Human();

Now, both instances of the class Human have the value of 100 in its $health property. This does not mean that instances always have the same values in its properties. We can modify $julius's health by creating a getter and setter methods for that property and then running the following code:

*class* Human

{

public $health;

public $clan;

public $strength = 3;

public $intelligence = 3;

public $stealth = 3;

public *function* \_\_construct() {

echo "I am alive";

$this->health = 100;

}

public *function* get\_health()

{

return $this->health;

}

public *function* set\_health($health)

{

$this->health = $health;

}

}

$julius->set\_health(80);

Now, $julius has a $health of 80 while $clarkson still has 100.

In the next chapter, we are going to have the instances of Human or instances of a class that subclasses Human to battle each other. We are going to provide two more functionalities to instances of Humans. We are going to implement an attack() instance method and also the trashTalk() instance method. We can achieve the following by modifying our code like the following:

*class* Human {

public $health;

public $clan;

public $strength = 3;

public $intelligence = 3;

public $stealth = 3;

public *function* \_\_construct()

{

echo "I am alive";

$this->health = 100;

}

public *function* \_\_get($property)

{

if (property\_exists($this, $property))

{

return $this->property;

}

}

public *function* \_\_set($property, $value)

{

if (property\_exists($this, $property))

{

$this->$property = $value;

}

return $this;

}

public *function* trashTalk()

{

echo "You want a piece of me?";

}

public *function* attack($human)

{

$this->trashTalk();

$luck = rand(0, 100);

if($luck \* $this->$intelligence > 1000)

{

if($luck > $human->stealth)

{

$human->health -= $this->strength;

return true;

}

else

{

return false;

}

}

else

{

return false;

}

}

}

Notice the two new magic methods that we added \_\_get() and \_\_set(). These magic methods are provided to us by PHP so that we don't have to implement our own**getters**and **setters**for each property. Now we can get any property by '$obj->property\_name' and we can set any property '$obj->property\_name = value.'

It's up to you whether you want to utilize these magic methods or write **getters** and **setters** for your property. There are pros and cons of both. Some people claim \_\_get() and \_\_set() are magic and should be avoided. Some people love how it saves more lines of code and makes it more readable.

This is how the attack method works. An instance of the class Human can execute the attack method and pass in an argument of a human object. It is important that we pass in an instance of a human object because we expect the argument that is passed in to have properties such as $intelligence and $strength. First, the instance of the object is going to execute its trashTalk() method. Then we are going to get a random number from 0 to 100 and store it in a variable called $luck. If $luck multiplied by the instance's intelligence is greater than 1000 AND the luck variable has a greater value than the $human's stealth, we are going to subtract the health of the opposing object by the amount of the current object's strength.

**Assignment: Bike**

Create a new class called **Bike** with the following properties/attributes:

* price
* max\_speed
* miles

Create 3 instances of this bike.

Now add a constructor method to the class and require the user to specify the price and max\_speed of each instance. In the constructor also specify in the code so that the initial miles is set to be 0 whenever a new instance is created.

Add the following functions for this class:

* **displayInfo()** - have this method display the bike's price, maximum speed, and the total miles driven.
* **drive()**- have it display "Driving" on the screen and increase the total miles driven by 10.
* **reverse() -** have it display "Reversing" on the screen and decrease the total miles driven by 5.

Have the first instance drive three times, reverse once, and have it *displayInfo().*

Have the second instance drive twice, reverse twice, and have it *displayInfo().*

Have the third instance reverse three times and *displayInfo().*

What would you do to prevent the instance from having negative miles?

## Assignment: Car

Create a class called**Car**. In the constructor, allow the user to specify the following attributes: price, speed, fuel, mileage. If the price is greater than 10,000, set the tax to be 15%. Otherwise, set the tax to be 12%.

Create five different instances of the class Car. In the class have a method called Display\_all() that returns all the information about the car as a string. In your constructor, call this Display\_all() method to display information about the car when a new car is created.

A sample output would be like this:

Price: 2000

Speed: 35mph

Fuel: Full

Mileage: 15mpg

Tax: 0.12

Price: 2000

Speed: 5mph

Fuel: Not Full

Mileage: 105mpg

Tax: 0.12

Price: 2000

Speed: 15mph

Fuel: Kind of Full

Mileage: 95mpg

Tax: 0.12

Price: 2000

Speed: 25mph

Fuel: Full

Mileage: 25mpg

Tax: 0.12

Price: 2000

Speed: 45mph

Fuel: Empty

Mileage: 25mpg

Tax: 0.12

Price: 20000000

Speed: 35mph

Fuel: Empty

Mileage: 15mpg

Tax: 0.15

## Assignment: Chaining Methods

So far, you have learned how to create objects and use methods within that object. Looking back at the Bike example, you probably had codes that looked like this:

$bike1->drive();

$bike1->drive();

$bike1->drive();

$bike1->reverse();

$bike1->displayInfo();

What if you want to chain methods and have your codes produce the same output but with the following code?

$bike1->drive()->drive()->drive()->reverse()->displayInfo();

This method of chaining you probably have already seen when you did jQuery and when you chained different methods. You could also chain methods in PHP by having the method return its own instance!

For example if $bike1->drive() returned its own instance ($bike1) then you could chain another method to $bike1->drive(), doing something like $bike1->drive()->reverse() or $bike1->drive()->displayInfo(); If the reverse method or the displayInfo() method also returned its own instance then you could keep chaining!

The way you have PHP return its own instance is to do something like this:

*class* Bike{

*function* drive(){

...

return $this;

}

}

**"return $this"** returns its own instance allowing you to chain methods after calling that method. Now that you learned this, see if you can go back to the Bike assignment and change it so that it would allow chaining of methods!

The practice of having OOP return its own instance is pretty common and this can be done with other languages (JavaScript, Ruby, Python, C++, Java) as well.  Note that in Ruby, you do not  **return $this**, rather you **return self**.

**Assignment: HTML Helper!**

Create a class called 'HTML\_Helper' that has the following two methods:

* **print\_table** -> takes an array that has multiple rows of information (imagine you queried the database and got all the list of users where each row has information like ["first\_name" => "Michael", "last\_name" => "Choi", "nick\_name" => "Sensei"]. This function takes this multi-dimensional arrays and prints out a beautiful HTML table with all the information in it. Make the keys appear in the first row (e.g. First Name, Last Name, Nick Name) and make the values of each row appear in the subsequent rows (e.g. Michael, Choi, Sensei).
* **print\_select** -> as the input, this method takes an array AND the name that should be used for the select tag; the method returns a string that is formatted in HTML. For example, say that*$sample\_array = ("CA", "WA", "UT", "TX", "AZ", "NY")*and we call*print\_select("state", $sample\_array)*method.  
    
  This would return a string that looks like:

<select name="state">

<option value="CA">CA</option>

<option value="WA">WA</option>

</select>

IMPORTANT - usually when you create a class, you don't normally have the methods in the class generate HTML output. For this assignment, though, you will make an exception and have the methods in the class generate HTML output directly. Classes that are created to help the rendering of HTML are often called 'Helpers' classes.

START WORKING ON THIS

**OOP ADVANCE**

# Procedural vs. OOP

In this tab, we'll talk about some of the benefits of OOP vs Procedural Programming.

## Benefits of OOP

A lot of people ask, why use OOP when using OOP takes much longer? It's true that OOP takes much longer IF your project is really small. However, as your project gets bigger and bigger, OOP will save you hours/weeks/months of development time. Following are some key benefits of using Object Oriented Programming.

1. **It allows you to leverage other people's codes (libraries, APIs, frameworks)**

Remember that a class is just a collection of functions and attributes and serves as a blueprint. Let's say that you are writing a library with a lot of useful functions you need when creating lots of web applications. Let's say that you've built so many web applications that you know what functions/methods other developers would need and you decided to, therefore, create a framework (basically, a fancy word for a bunch of classes) with built-in functions for other developers to use.

With OOP, you can create several classes for other developers to inherit and use. Since all of your functions/methods are contained in your own classes, other developers do not even need to modify your class/functions to use your functions. They can simply create their own class and have it inherit some of the classes you wrote (and thereby gaining access to all of the cool functions you built)! This saves a lot of time and allows you to leverage other people's codes.

1. **It allows you to maintain and organize complex projects into more manageable chunks**

Just think of it this way. If you had 5 files in your folder, is there any need to create multiple folders? Probably not. What if you have 1000 files? Or 100,000 files? As the number of files gets larger and larger, wouldn't it help to organize your folders in different folders that do specific things (e.g. photos, wall, etc)? Objects/Classes are exactly like folders in this scenario. Doing things the OOP way takes a little longer when the project is small (when you only have 5 files in the folder) but when the project gets bigger, OOP helps you to organize your codes more neatly.

## ****When Not to Use OOP****

Use the **DRY**(**Don't Repeat Yourself**) principle. Not everything should be Object Oriented. When dealing with a stand alone function that serves a specific purpose, there is no need to create a class. Reserve using OOP when you notice yourself using a series of functions that work together for one main output or cause.

# Inheritance

Our world is unique. It has Humans, but the Humans of our world are separated into three distinct classes. In this tab, we are going to create subclasses of Human: classes Wizard, Ninja, and Samurai. Creating subclasses is useful when you want to create a more specialized version of a current class that you have. If it is just a specialized class it means that the current functionality will mostly remain the same except for some minor adjustments. By using class inheritance, we don't have to repeat ourselves. This is our current class of Human.

*class* Human

{

public $health;

public $clan;

public $strength = 3;

public $intelligence = 3;

public $stealth = 3;

public *function* \_\_construct()

{

echo "I am alive";

$this->health = 100;

}

public *function* \_\_get($property)

{

if (property\_exists($this, $property))

{

return $this->property;

}

}

public *function* \_\_set($property, $value)

{

if (property\_exists($this, $property))

{

$this = $value;

}

return $this;

}

public *function* trashTalk()

{

echo "You want a piece of me?";

}

public *function* attack($human)

{

$this->trashTalk();

$luck = rand(0, 100);

if($luck \* $this->$intelligence > 1000 && $luck > $human->stealth)

{

$human->health -= $this->strength;

return true;

}

else

{

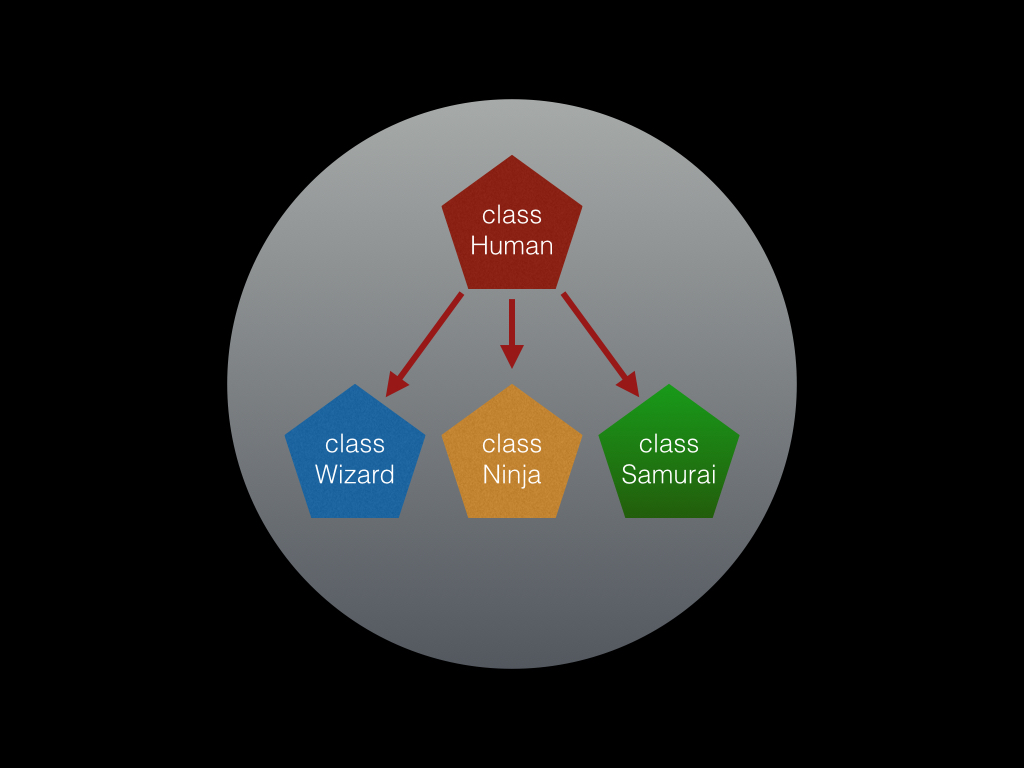
return false;

}

}

}

Say we wanted to create a new class and we wanted it to be able to use all of the functions and properties that we already had in another class, but there were some additional functions and properties that we wanted to add to it. In this case, we would have our new class **inherit** or **extend** the original class or what would be considered the **parent** class. Looking back on our construction example... Here is a visual representation of what inheritance would look like.



As you can see class Wizard, class Ninja, and class Samurai all inherit the **blueprint** of class Human but in addition, they add their own **blueprint.**

# ****Extending an existing class****

To **extend** a class, all we have to do when creating the new class is use the keyword **extends** followed by the name of the class that we wish to extend. The new class will then have access to all the properties and methods within that class. Our new class will also inherit the **\_\_construct** function that is within the **parent** class. This means that when an instance is created for our new class it will call the **\_\_construct** function in the **parent** class.

*class* Wizard *extends Human*

{

public *function* heal()

{

$this->health += 10;

}

}

*class* Ninja *extends Human*

{

public *function* steal()

{

$this->stealth += 5;

}

}

*class* Samurai *extends Human*

{

public *function* sacrifice()

{

$this->health -= 5;

$this->strength += 10;

}

}

// creating an instance of Wizard, Ninja and Samurai

$harry = new Wizard();

$rain = new Ninja();

$tom = new Samurai();

// all instances still have human properties because its class extends the Human class

echo $harry->health;

echo $rain->health;

echo $tom->health;

// yet they are subclasses which mean they can extend the current functionality of Human class

// instances of the Wizard class can perform the heal method

$harry->heal();

echo $harry->health;

// instances of the Ninja class can perform the steal method

$rain->steal();

echo $rain->stealth;

// instances of the Samurai class can perform the sacrifice method

$tom->sacrifice();

echo $tom->health;

echo $tom->strength;

# Overwriting

In most cases, you would want to keep the properties and methods that you were inheriting from your parent class, but if you wanted to create a method or property to take the place of the existing one in the parent class you would just overwrite it.

*class* Wizard *extends Human*

{

public *function* heal()

{

$this->health += 10;

}

public *function* trashTalk()

{

echo "Happiness can be found even in the darkest of times";

}

}

*class* Ninja *extends Human*

{

public *function* steal()

{

$this->stealth += 5;

}

public *function* trashTalk()

{

echo "Now you see me...";

}

}

*class* Samurai *extends Human*

{

public *function* sacrifice()

{

$this->health -= 5;

$this->strength += 10;

}

public *function* trashTalk()

{

echo "The flower that blooms in adversity is the most beautiful of all";

}

}

$ron = new Wizard();

$sasuke = new Ninja();

$kenshin = new Samurai();

// all three instances have the method trashTalk which was declared in the Human blueprint which

// all three of the subclasses extends. However, each subclass overwrote the previous implementation

$ron->trashTalk();

$sasuke->trashTalk();

$kenshin->trashTalk();

# Preserving

In the last lesson, we saw how we can overwrite an existing property or method in our inherited class, but what if we want to keep the existing method and add something additional to it? We can use the **scope resolution operator (::)** to call the function from our parent class within the class we are overwriting.

## ****Scope Resolution Operator****

The **scope resolution operator** allows us to refer to our parent class to call a method from there directly. We can refer to any of the parent methods or properties by specifying the keyword **parent** followed by two colons **(::)**and the name of the property or method that you want to refer to from the parent class.

*class* Wizard *extends Human*

{

public *function* \_\_construct()

{

parent::\_\_construct();

$this->clan = "Wizard";

$this->strength = 5;

$this->intelligence = 40;

$this->stealth = 5;

}

public *function* heal()

{

$this->health += 10;

}

public *function* trashTalk()

{

echo "Happiness can be found even in the darkest of times";

}

public *function* attack($human)

{

parent::attack($human);

$this->heal();

}

}

*class* Ninja *extends Human*

{

public *function* steal()

{

$this->stealth += 5;

}

public *function* trashTalk()

{

echo "Now you see me...";

}

public *function* attack($human)

{

parent::attack($human);

$this->steal();

}

}

*class* Samurai *extends Human*

{

public *function* sacrifice()

{

$this->health -= 5;

$this->strength += 10;

}

public *function* trashTalk()

{

echo "The flower that blooms in adversity is the most beautiful of all";

}

public *function* attack($human)

{

parent::attack($human);

$this->sacrifice();

}

}

$ron = new Wizard();

$sasuke = new Ninja();

$kenshin = new Samurai();

// all three instances have the method trashTalk which was declared in the Human blueprint which

// all three of the subclasses extends. However, each subclass overwrote the previous implementation

$ron->trashTalk();

$sasuke->trashTalk();

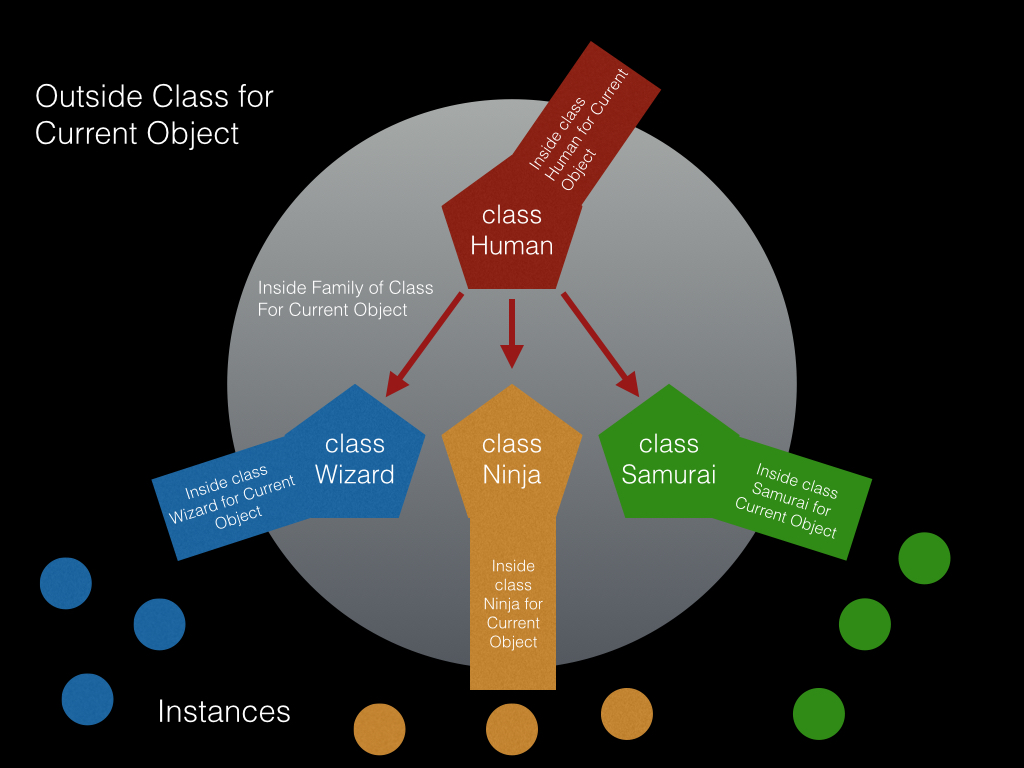
$kenshin->trashTalk();

**Visibility**

So far we've mentioned what it means for a property to be visible. Now we're going to see how we can change the visibility of our properties and methods, so that they can only be used by the class itself or its family (child classes, parent classes).

**Public**

We have been using the **public** keyword to denote methods and properties. This just means that these methods and properties are available not only to code within the class itself, but also to code *outside*the class. Let's define the different scopes from which you can access a method or property. From the diagram in the Inheritance lesson, here are the different **visibility scopes**:



**Visibility Scopes**

1. For attributes that are defined *in the same class* as the code that is trying to access it, you can do so by using **$this**. Example: code within the Human class can access the attribute *health*, which is defined within that same class. The code would reference the attribute as follows:   
   **$this->health = 100;**
2. For attributes that are defined *in a parent class*, you use the same syntax with **$this**. Example: code within the Wizard class can access the attribute *health*, which is defined in the parent Human class. The code would reference the attribute as follows:  
   **$this->health += 10;**
3. For attributes *defined anywhere else* (for attributes that are members of any other object), $this cannot see them. If we are currently within the class Bicycle, then *health*has no meaning to us (health is not an attribute of the Bicycle class). If, however, this code contains a local attribute ***owner*** of type Human, then we could reference the health attribute as follows:   
   **$this->owner->health -= 50;       // ouch -- the bike crashed!**

The keywords *public*, *protected*and *private*can each be used to set visibility. For every property or method, we should decide whether we want it to be accessible in *all three*visibility scopes, or just the *first two* scopes, or just the *first one*.

**Public Visibility Scope**

The public keyword makes the specified property or method available in *all three*visibility scopes.

# Protected

In the lesson on visibility, we were able to define the different scopes that exist and determined that the public keyword (called an **access modifier**) adds no scope restrictions. This means that anything defined as public can be accessed within any scope, as long as we have a handle to that object. What if we didn't want those that held a handle to an instance object to be able to directly access certain properties or methods? One way to limit access to an attribute is by setting its visibility to **protected.**

## ****Setting Visibility to Protected****

By putting the keyword **protected** in front of a property or method, you limit any access to **code within the class or its child classes**. The property or method can no longer be seen outside of the class, or even by its own parent classes. In the example below, we set the get\_parent\_property() method to protected.

*class* MyBaseClass

{

    public *function* \_\_construct($instance)

    {

        echo "I am called for object: " . $instance;

    }

    protected $base\_property = "I am a base property!  woohoo.";

    public *function* set\_the\_property($prop\_value)

    {

        $this->base\_property = $prop\_value;         //  I can always see/set all my own attributes

    }

}

*class* MyChildClass *extends MyBaseClass*

{

    public *function* directly\_set\_property($prop\_value)

{

        // 'protected' makes $base\_property available to child classes.

        $this->base\_property = $prop\_value;    // Yes, works nicely.

}

}

$base = new MyBaseClass('base');

$child = new MyChildClass('child');

// 'protected' makes $base\_property unavailable, except within self or child classes.

$base->base\_property = 86;         // Nope!  This will cause an error

$child->base\_property = 68;         // Nope!  This will cause an error

// Instead call 'public' method set\_the\_property(), which has access to the protected attribute

$base->set\_the\_property(42);       // Yes! This will work wondrously

$child->set\_the\_property(42);       // Yes!  Wow! -- works wonderfully

// Or you can call a public method on the subclass -- it can see the protected attribute as well

$base->directly\_set\_the\_property(86);       // Fail!  MyBaseClass doesn't contain a method by this name!

$child->directly\_set\_the\_property(42);      // Yes!  Wow! -- works wonderfully

# Private

In the previous lesson, we saw how to restrict access to properties and methods by using the protected access modifier. This removes any visibility outside the object or its children. What if we had a property or method that we didn't want our child class to access either? To do this, we set the visibility to **private**.

## ****Marking a member as private****

Setting the keyword **private**in front of the property or method makes it available only within the scope of the current class. It cannot be inherited or called by any other code. In the following example, we set our $base\_property member to **private.**

*class* MyBaseClass

{

    public *function* \_\_construct($instance)

    {

        echo "I am called for object: " . $instance;

    }

// Note: in the previous example, this was marked 'protected'

private $base\_property = 0;

    // Note: in the previous example, this was marked 'public'

    protected *function* set\_the\_property($prop\_value)

    {

        $this->base\_property = $prop\_value;         //  I can always see/set my own attributes

}

}

*class* MyChildClass *extends MyBaseClass*

{

    public *function* directly\_set\_property($prop\_value)

{

        // 'private' makes MyBaseClass::base\_property unavailable, even to child classes.

        $this->base\_property = $prop\_value;       // This causes an error -- cannot access prop\_value

}

    public *function* child\_sets\_property($prop\_value)

    {

        // 'protected' makes MyBaseClass::set\_the\_property() visible to child classes like this one.

$this->set\_the\_property($prop\_value);   // This will work well.

    }

}

$child = new MyChildClass('child');

$base = new MyBaseClass('base');

// 'private' makes $base\_property unavailable outside the *class* itself

$child->base\_property = 68;         // Nope! This will cause an error

$base->base\_property = 86;                          // Nope! This will cause an error

// $base\_property is even inaccessible by child classes.

$child->directly\_set\_property(68);               // Nope! We can call the method, but it then fails

// 'protected' makes set\_the\_property() invisible outside itself and child classes.

$child->set\_the\_property(68);       // Nope! We can't access that method from out here.

$base->set\_the\_property(86);                       // Nope! We can't access that method from out here.

// from inside the child\_sets\_property() public method, we can access protected attributes

$child->child\_sets\_property(42); // YES! This works wonderfully well.

// and remember that parent objects don't "reverse-inherit" the methods/members of their children!

$base->directly\_set\_property(86);                 // Nope! MyBaseClass contains no method with that name

$base->child\_sets\_property(86);                    // Fail! $base doesn't even have a child\_sets\_property()!

# Using OOP

Watch the included video and think about how using OOP can enhance your existing code. For your convenience, we have uploaded the[database.php](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_2250/handouts/chapter2250_3160_database.php" \t "_blank). Try it out! Build something cool with it!

## Assignment: Animal

### Objective

The objective of this assignment is to help you understand inheritances. Remember that a class is more than just a collection of properties and methods. If you want to create a new class but the new class has all the attributes and methods defined in another class, you can have it inherit the original class (called the parent class) instead of copying and pasting the codes in the original class to the new class. The children class can access all the attributes and methods of the parent class AND also have new attributes and methods for its instances to call. You could also have several unique children classes that are being created from the same parent class.

### To Do

Create a class called Animal with the following attributes: name and health. Give the animal following three methods: walk(), run(), and displayHealth(). Give a new animal a health of 100 when it gets created. When a walk method is invoked, have the health decrease by 1. When a run method is involved, have the health decrease by 5. When a displayHealth() method is invoked, display on screen the name of the Animal and the health.

Create an instance of the animal called 'animal' and have this animal walk three times, run twice, and have it display its health.

Now, create another class called Dog that inherits everything that the Animal does and has, but 1) have the default health be 150 and 2) add a new method called pet, which when invoked, increases the health by 5. Have the Dog walk three times, run twice, petted once, and have it display its health.

Now, create another class called Dragon that inherits everything that the Animal has and does, with these two changes: 1) have the default health be 170 and 2) add a new method called fly, which when invoked, decreases the health by 10. Have the Dragon walk three times, run twice, fly twice, and have it display its health. When the Dragon's displayHealth function is called have it say 'this is a dragon!' before it displays the default information (make sure you still call the parent's displayHealth function).

Now for the first instance of Animal (instance called 'animal'), try calling a method 'fly' or 'pet' and make sure it doesn't work. :)

START WORKING ON THIS

## Deck of Cards

Now that you've learned some advanced OOP concepts and data structures, try building a deck of cards. Your Deck class should contain 52 standard cards, 13 cards and 4 suits. Your Deck should have a shuffle method, a reset method, and the ability to deal cards (which will return the dealt card and remove it from that deck).

Next, create the Player class, which should have a name, a hand, and the ability to draw and discard cards. Using this data structure, try implementing a simple card game. What new methods would your classes need for Go Fish or Poker? Blackjack could be done with this Deck of Cards, no extra methods or classes needed! [Here](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_3303/handouts/chapter3303_4963_cards-png.zip) is a set of .png card faces and backs you can use.

Spend no longer than 5 hours implementing your card game!

START WORKING ON THIS

**Assignment: PHPUnit**

Later when we cover rails, we'll cover Test-Driven Development more in depth. If you are done with all the optional OOP assignments, please go through the following articles and get yourself familiar with PHPUnit. You can write some unit tests for some of the assignments you've done in this chapter.

Articles to read:

1. <http://code.tutsplus.com/tutorials/lets-tdd-a-simple-app-in-php--net-26186>
2. <http://www.phpclasses.org/blog/post/237-7-Reasons-Why-TDD-Failed-to-become-Mainstream.html>
3. <http://www.paulund.co.uk/test-driven-development-with-php>
4. <http://stackoverflow.com/questions/2213741/dead-simple-introduction-to-tdd-test-driven-development>