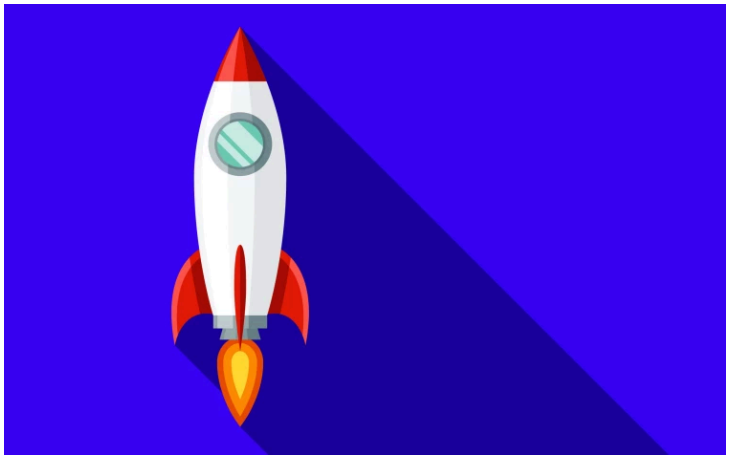
Object Oriented Programming (Course 1)



Chapter 1: The Project

Welcome KnpU Peeps! I am *so* glad you're here today - I am *not* kidding - because we're intro'ing into one of my absolute favorite topics: object-oriented programming. This is what gets me up in the morning, excited to go to work, this is why I love to code. And I hope, you'll be as geeked about these new tools as I am.

Like always, we're going to learn this stuff by building a real app! Don't be lazy - code along with me to really get the feel for this stuff. Go to the screencast page and click to download the code. Unzip that file - you'll see a "start" directory. I'll rename this to `oo` and move it into a `Sites/` directory in my Home folder.

There's not much going on yet guys - just 3 PHP files and some CSS and JS files. That's it. Let's take the app for a test drive!

You can of course use a web server like Apache and setup a Virtual Host, but I perfer the built-in PHP web server. Open up a terminal. Move into the `oo` directory. From here, to start the built-in web server you can call php with the `-S` option and pass it an address:

```
cd ~/Sites/oo
php -S localhost:8000
```

It'll hang there, and that means it's working: we instantly have a web server at the address that's serving from our directory. Let's go to the browser and try it out:

```
http://localhost:8000
```

Voilà! Welcome to OO Battleships of Space! This awesome app does one important thing: it lets you fight one ship against another. We have 4 ships, each has a "weapon power", which is your offense, "strength", which is your defense and "Jedi Factor". This last one randomly causes one ship to go all "Luke-Skywalker" on another and and destroy it instantly.

Ok, let's put 4 "Jedi Starfighters" against 1 giant "Super Star Destroyer". A Super Star Destroyer is a lot more powerful, so it'll probably win.

Stunning upset! The Jedi Starfighters won! Probably they used their Jedi ways to find some crazy weakness. Of course, we can go back and do a re-match: 4 Jedi Starfighters against another Super Star Destroyer, and now the Destroyer wins.

How the App Works

Behind this, we have exactly 3 PHP files. First is `index.php`, which is the homepage. It requires `functions.php` and calls `get_ships()` from it:

index.php

```php
<?php
require __DIR__.'/functions.php';

$ships = get_ships();
// ... lines 5 - 106
```

All that does is create this nice associative array of 4 ships. Each has `name`, `weapon_power`, `jedi_factor` and `strength` keys:

functions.php

```
// ... lines 1 - 2
function get_ships()
{
    return array(
        'starfighter' => array(
            'name' => 'Jedi Starfighter',
            'weapon_power' => 5,
            'jedi_factor' => 15,
            'strength' => 30,
        ),
        'cloakshape_fighter' => array(
            'name' => 'CloakShape Fighter',
            'weapon_power' => 2,
            'jedi_factor' => 2,
            'strength' => 70,
        ),
        'super_star_destroyer' => array(
            'name' => 'Super Star Destroyer',
            'weapon_power' => 70,
            'jedi_factor' => 0,
            'strength' => 500,
        ),
        'rz1_a_wing_interceptor' => array(
            'name' => 'RZ-1 A-wing interceptor',
            'weapon_power' => 4,
            'jedi_factor' => 4,
            'strength' => 50,
        ),
    );
}
// ... lines 32 - 92
```

Back in `index.php`, we use those below in a `foreach` to create a table:

index.php

```
↕ // ... lines 1 - 3
4 $ships = get_ships();
↕ // ... lines 5 - 65
66                     <?php foreach ($ships as $ship): ?>
67                         <tr>
68                             <td><?php echo $ship['name']; ?></td>
69                             <td><?php echo $ship['weapon_power']; ?></td>
70                             <td><?php echo $ship['jedi_factor']; ?></td>
71                             <td><?php echo $ship['strength']; ?></td>
72                         </tr>
73                     <?php endforeach; ?>
↕ // ... lines 74 - 106
```

And we use it again to create the options in the select drop-downs:

index.php

```
↕ // ... lines 1 - 78
79                     <form method="POST" action="/battle.php">
↕ // ... lines 80 - 81
82                         <select class="center-block form-control btn drp-
dwn-width btn-default btn-lg dropdown-toggle" name="ship1_name">
83                             <option value="">Choose a Ship</option>
84                             <?php foreach ($ships as $key => $ship): ?>
85                                 <option value="<?php echo $key; ?>"><?php
echo $ship['name']; ?></option>
86                             <?php endforeach; ?>
87                         </select>
↕ // ... lines 88 - 99
100                    </form>
↕ // ... lines 101 - 106
```

When we submit, it POST's to `battle.php`. That *also* calls `get_ships()`, reads some `$_POST` data to figure out which ships are fighting and how many, and eventually calls a `battle()` function that finds the winnner. I'll show you that later:

battle.php

```
↕ // ... lines 1 - 3
4 $ships = get_ships();
↕ // ... line 5
6 $ship1Name = isset($_POST['ship1_name']) ? $_POST['ship1_name'] : null;
7 $ship1Quantity = isset($_POST['ship1_quantity']) ?
  $_POST['ship1_quantity'] : 1;
8 $ship2Name = isset($_POST['ship2_name']) ? $_POST['ship2_name'] : null;
9 $ship2Quantity = isset($_POST['ship2_quantity']) ?
  $_POST['ship2_quantity'] : 1;
↕ // ... lines 10 - 25
26 $ship1 = $ships[$ship1Name];
27 $ship2 = $ships[$ship2Name];
28
29 $outcome = battle($ship1, $ship1Quantity, $ship2, $ship2Quantity);
↕ // ... lines 30 - 97
```

Then we use that `$outcome` to show a status report below:

battle.php

```
↕ // ... lines 1 - 28
29 $outcome = battle($ship1, $ship1Quantity, $ship2, $ship2Quantity);
↕ // ... lines 30 - 66
67                 <h3 class="text-center audiowide">
68                     Winner:
69                     <?php if ($outcome['winning_ship']): ?>
70                         <?php echo $outcome['winning_ship']['name']; ?>
71                     <?php else: ?>
72                         Nobody
73                     <?php endif; ?>
74                 </h3>
↕ // ... lines 75 - 97
```

There's our app! It's got no object-oriented code yet. And you know what? That makes me sad.

Time to fix it!

Chapter 2: A Class and an Object

Let's create a fresh file that we can play around with - call it `play.php`. Now we can warn the rebels that It's a TRAAAP!

play.php

```php
<?php

echo 'IT\'S A TRAAAAAAPPPP!';
```

Put `play.php` in the URL... and there it is! We've conquered the echo statement!

Creating a Class

Now to the cool stuff! The first super-important big awesome-crazy thing in object-oriented programming is.... a class! To create one, write the keyword `class` then the name of it - this name can be almost anything alphanumeric. Finish things off with an open curly brace and a close curly brace. Nice work.

play.php

```php
// ... lines 1 - 2
class Ship
{

}
// ... lines 7 - 9
```

Don't worry about *what* this "class" thing is yet. But if you refresh, you can see that creating a class doesn't actually *do* anything.

Creating an Object

Creating a class, check! And with that, we can see the second super-important big awesome-crazy thing in object-oriented programming: an object! Once you have a class, you can instantiate a new *object* from that class, and it looks like this.

Create a variable called `$myShip` and then use the `new` keyword, followed by the name of the class, then open parenthesis, close parenthesis:

play.php

```
// ... lines 1 - 2
class Ship
{

}

// but it doesn't do anything yet...
$myShip = new Ship();
// ... lines 10 - 12
```

It kind of looks like we're calling a function called `Ship()`, except for the `new` keyword in front that tells PHP that `Ship` is a class, and we're instantiating a new object from it.

Before we explain any of this - refresh again. Still no changes. We have this new thing called an object that's set to `$myShip`, but it doesn't cause anything to happen.

The Skinny on Classes and Objects

Ok, let me explain this class and object stuff. When this stuff finally "clicks", you'll know that you really *get* object-oriented programming. So listen carefully, and I'll come back to this later as we add more stuff.

Class: A Worksheet with Labels and Blank Lines

Pretend with me: you're the manager of a shipping-dock on the Deathstar. To avoid any force-choking, when each ship lands, you need to take an inventory of it: what's its name, size, does it have a warp drive, what's the fuel level, weapon power, defense strength and other stuff. And also imagine, that even though you're in a flying space death machine, you don't have computers: you track everything by making copies of a template worksheet you designed in Excel.

A class is like that empty template, with blank lines for the ship's name and size, a yes/no option for warp drive and blanks for fuel-level, weapon power and defense strength. It's not actually a ship of course, but it defines all the properties that a ship might have.

Object: A Completed Worksheet

Ok, I think you've got the gist on classes, now let's go over objects. This is where I'm supposed to tell you to think of an object like a *real* ship that's landed in our dock. That's correct, but I think it's closer to think of an object like a completed worksheet that we've filled out for a ship: complete with its name, weapon details, strength and fuel-levels.

If 10 ships land, then we'll print out 10 blank worksheets and fill each in with different details. If we re-fuel one of those ships, we'll update the fuel level on its worksheet.

But each ship is using the same template, or class. If we wanted to also track a ship's weight, we'd need to go back to the template and add a field for it there.

Right now, our class is empty. That's like a template with no blank fields to fill in. Not helpful - time to fix that!

Objects are like Uptight (Structured) Arrays

Ok, clear your mind quickly and think about arrays. An array holds data on keys: we choose a key - like `hasWarpDrive` - then put something there. Simple.

An object works *exactly* like an array, except instead of calling these storage spaces keys, we call them properties. But basically, they work the same way.

There is one big difference between an array and an object. With an array, you can just invent a new key and set data on it. But with an object, you need to pre-register the possible properties it might have in its class.

Back at our loading dock, this means an array is like a template where each line has two blanks: one for the value and one for what that value is. We might fill in a line with the ship's name, but not the fuel level. And maybe we'll write down the color of the ship on another line, and because I don't really like my job, I'll put my current favorite song on the last line. If I give 10 sheets to my manager, they'll have no guarantee *what* data I may or may not have recorded. Simple, but unstructured.

But an object is like the sheets we were talking about earlier: it has a list of exactly what we want to track with a blank next to each for that value. So if we want to be able to store the name of each ship, we'll need to add a spot for it on the template. In object-oriented land, we need to add a `name` property to our `Ship` class.

Adding a Property to a Class

Let's do it! Say `public` then `$`, then the name of your property. In this case, our name is actually `name`:

play.php

```
// ... lines 1 - 2
class Ship
{
    public $name;
}

// but it doesn't do anything yet...
$myShip = new Ship();
// ... lines 10 - 12
```

This doesn't do anything, but now `Ship` objects are allowed to hold data on a `name` property. And how do we set that? We do it with this syntax here:

play.php

```
// ... lines 1 - 2
class Ship
{
    public $name;
}

// but it doesn't do anything yet...
$myShip = new Ship();
$myShip->name = 'TIE Fighter';
// ... lines 11 - 13
```

Setting the property

Now we can read the data from that property using the same syntax:

play.php

```
// ... lines 1 - 8
$myShip = new Ship();
$myShip->name = 'TIE Fighter';

echo 'Ship Name: '.$myShip->name;
```

Try it out! It's working great.

Does this feel familiar? It works *exactly* like an array, except instead of the square-bracket syntax, we use this dash, greater-than syntax, let's call that an "arrow".

The *real* difference between an object and an array is that an object has a class that defines all possible properties that it can hold, instead of being able to store any random keys you dream up, like an array.

TIP Technically, you can set a property (e.g. `$myShip->weight = 100`) on an object without creating a property for it in the class. But this is frowned upon, and not seen commonly in higher-quality code. So don't do it!

Classes: Data Structure Docs

And with this simple difference, you get a *huge* benefit. A class is like programmer documentation. If I give you an array, you have no idea what data is on it. You can `var_dump` it to see, then just hope that it'll always have the same keys in the future.

But if I hand you a `Ship` object, you *know* something about it. You know that it will *always* have a `name` property, and it'll never have `email` or `phone` properties: because they're not in the class. The class gives us a skeleton and some rules we know it'll follow.

Now let's talk about something even better than this: methods.

Chapter 3: Class Methods

Arrays and objects have a lot in common: arrays have keys while objects have properties, but both store data.

Objects have one other big, incredible, mind-blowing advantage: methods. A method is a fancy word for a function, and we're all comfortable with those. The difference is that a method lives *inside* of a class.

To create a method, start with `public function`. After this, it looks like a normal function - let's call ours `sayHello`, and add the normal parenthesis and curly braces stuff. Inside, we can do anything - so let's echo `Hello`:

play.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
5     public $name;
6
7     public function sayHello()
8     {
9         echo 'Hello!';
10    }
11 }
↕ // ... lines 12 - 20
```

To call the method, let's use our object "arrow" syntax. Let's add an `<hr/>` so I don't confuse myself, then `$myShip`, then arrow, then `sayHello()`:

play.php

```
↕ // ... lines 1 - 13
14 $myShip = new Ship();
↕ // ... lines 15 - 18
19 $myShip->sayHello();
```

The only difference between accessing a property and a method is the open parenthesis and close parenthesis, so don't forget that. And a method is *just* like a traditional function, except it lives inside of the class, so you have to call it on the object. And when we refresh, success!

This is kind of *amazing* because we have little packages of data that can now perform actions and *do* things through methods. Arrays can't do *anything* like that.

Referencing Properties from Inside a Class

So let's add another - a `getName()` method. Remember how functions can return a value? Sure, methods can do that too. Let's not *actually* return the name of *this* ship yet, let's fake it:

play.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 11
12     public function getName()
13     {
14         return 'FAKE NAME';
15     }
16 }
↕ // ... lines 17 - 25
```

Down below, we call it the exact same way, except we can echo what it returns:

play.php

```
↕ // ... lines 1 - 18
19 $myShip = new Ship();
20 $myShip->name = 'TIE Fighter';
21
22 echo 'Ship Name: '.$myShip->getName();
↕ // ... lines 23 - 25
```

Now refresh! There's our fake name. Of course, what I *really* want to do is return the name of the ship that I'm calling this method on. I know, not the most interesting function, but it gives us a new problem. When we have the `$myShip` variable, we *know* how to access a property - just use the arrow syntax on it. But when we're *inside* of the class, how can we get the value of the `name` property?

The answer is with a very special variable called `$this`:

play.php

```
// ... lines 1 - 2
class Ship
{
// ... lines 5 - 11
    public function getName()
    {
        return $this->name;
    }
}
// ... lines 17 - 25
```

Here's the rule: when you're inside of a method, you magically have access to a variable called `$this`. And `$this` is whatever `Ship` object that we're calling the method on, in our case our favorite `$myShip` object whose name is "Jedi Starship". In *all* cases, the variable is called `$this`, that's just what the PHP Jedi elders decided this magic name should be.

When we refresh, hey - there's our ship's *real* name!

Adding more Properties

Our `Ship` class has just one property. Let's go back and look at the `get_ships()` function I wrote before we started. Here, each ship has a key for `name`, `weapon_power`, `jedi_factor` and `strength`. Let's add three more properties to *our* class: `weaponPower`, `jediPower` and `strength`. I camel-cased `weaponPower` and `jediPower` - that's kind of a standard, but you can do whatever you want:

play.php

```
// ... lines 1 - 2
class Ship
{
    public $name;

    public $weaponPower;

    public $jediFactor;

    public $strength;
// ... lines 12 - 21
}
// ... lines 23 - 31
```

Default Property Values

You can also give a property a default value. So if you create a new `Ship` object and we never set the `weaponPower`, let's default its value to zero. Let's do that for `jediFactor` and `strength` too.

play.php

```
↕ // ... lines 1 - 2
3  class Ship
4  {
5      public $name;
6
7      public $weaponPower = 0;
8
9      public $jediFactor = 0;
10
11     public $strength = 0;
↕ // ... lines 12 - 21
22 }
↕ // ... lines 23 - 31
```

These new properties aren't special, so we can use them like before. Let's `var_dump` the `weaponPower` property:

play.php

```
↕ // ... lines 1 - 24
25 $myShip = new Ship();
↕ // ... lines 26 - 30
31 echo '<hr/>';
32 var_dump($myShip->weaponPower);
```

When we refresh, it dumps zero. Cool! That's using our default value because we never actually set the `weaponPower`. Let's set it now - `$myShip->weaponPower = 10;`:

play.php

```
↕ // ... lines 1 - 24
25 $myShip = new Ship();
26 $myShip->name = 'TIE Fighter';
27 $myShip->weaponPower = 10;
↕ // ... lines 28 - 31
32 echo '<hr/>';
33 var_dump($myShip->weaponPower);
```

Now we see 10. Awesome!

Chapter 4: Methods that Do work

Our two methods are simple. So now let's add something useful. We'll be printing out the details of our ships all over the place, so I need a way to see the `weaponPower`, the `strength` and the `jediFactor` all at once - like a little summary. One handy way to do this is with a new method that creates that summary string and returns it to us.

Make a new function `getNameAndSpecs()`. Let's use the nice `sprintf` function with %s wildcards for the ship's name, followed by the `weaponPower`, `jediFactor` and `strength`. Now, we need to pass it what to fill in for those `%s` placeholders. To reference the name, use the magic `$this` variable: `$this->name`. Do the same thing for `weaponPower`, `jediFactor` and `strength`:

play.php

```
↕ // ... lines 1 - 2
3  class Ship
4  {
5      public $name;
6
7      public $weaponPower = 0;
8
9      public $jediFactor = 0;
10
11     public $strength = 0;
↕ // ... lines 12 - 22
23     public function getNameAndSpecs()
24     {
25         return sprintf(
26             '%s (w:%s, j:%s, s:%s)',
27             $this->name,
28             $this->weaponPower,
29             $this->jediFactor,
30             $this->strength
31         );
32     }
33 }
↕ // ... lines 34 - 46
```

And of course, make sure you have a `return` statement in front of all of this! Let's enjoy our hard work by echo'ing the method below: `$myShip`, arrow, `getNameAndSpecs` then the open and close parenthesis so PHP knows this is a method, not a property by that name.

play.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 22
23     public function getNameAndSpecs()
24     {
↕ // ... lines 25 - 31
32     }
33 }
↕ // ... lines 34 - 35
36 $myShip = new Ship();
↕ // ... lines 37 - 42
43 echo '<hr/>';
44 echo 'Ship Description: '.$myShip->getNameAndSpecs();
↕ // ... lines 45 - 46
```

Ready to try it? Refresh! There's our weird-little summary. We can use this across our app, and if we ever want to change how it looks, we only need to update one spot.

Method Arguments

Being PHP pro's, we of course know that functions can have arguments. Once again, a method in a class is no different. Isn't that nice? Let's say that sometimes we need an even *shorter* summary. Add an argument to the method called `$useShortFormat`. Now, use an `if` statement to choose between two different formats:

play.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 22
23     public function getNameAndSpecs($useShortFormat)
24     {
25         if ($useShortFormat) {
26             return sprintf(
27                 '%s: %s/%s/%s',
28                 $this->name,
29                 $this->weaponPower,
30                 $this->jediFactor,
31                 $this->strength
32             );
33         } else {
34             return sprintf(
35                 '%s: w:%s, j:%s, s:%s',
36                 $this->name,
37                 $this->weaponPower,
38                 $this->jediFactor,
39                 $this->strength
40             );
41         }
42     }
43 }
↕ // ... lines 44 - 58
```

We'll just take out the w, j and s and put slashes instead. Cool! Now my editor is angry because `getNameAndSpecs()` requires an argument. So pass `false` when we call it, then call it again and use `true`.

play.php

```
↕ // ... lines 1 - 45
46 $myShip = new Ship();
↕ // ... lines 47 - 53
54 echo $myShip->getNameAndSpecs(false);
55 echo '<hr/>';
56 echo $myShip->getNameAndSpecs(true);
↕ // ... lines 57 - 58
```

Refresh, Perfect!

Chapter 5: Multiple Objects

This object-oriented, or OO, stuff gets *really* fun once we have multiple objects. Afterall, it takes at least 2 ships to start a battle.

But first, let's summarize all of this printing stuff into a normal, traditional, flat function called `printShipSummary()`. It'll take in a `Ship` object as an argument, which I'll call `$someShip`. Now, copy all the echo stuff into the function.

The argument to the function could be called anything. Since I chose to call it `$someShip`, all of the `$myShip` variables below need to be updated to `$someShip` also. This is just classic behavior of functions - nothing special. I'll use a trick in my editor:

play.php

```
↕ // ... lines 1 - 44
45 function printShipSummary($someShip)
46 {
47     echo 'Ship Name: '.$someShip->getName();
48     echo '<hr/>';
49     $someShip->sayHello();
50     echo '<hr/>';
51     echo $someShip->getNameAndSpecs(false);
52     echo '<hr/>';
53     echo $someShip->getNameAndSpecs(true);
54 }
↕ // ... lines 55 - 62
```

Ok, saving time!

Back at the bottom of the file, call this like any traditional function, and pass it the `$myShip` variable, which we know is a `Ship` object:

play.php

```
↕ // ... lines 1 - 44
45 function printShipSummary($someShip)
↕ // ... lines 46 - 53
54 }
↕ // ... lines 55 - 56
57 $myShip = new Ship();
58 $myShip->name = 'TIE Fighter';
59 $myShip->weaponPower = 10;
60
61 printShipSummary($myShip);
```

So we're throwing around some objects, but this is just normal, flat, procedural programming. When we refresh it's exactly the same.

Create Another Ship

Now, to the good stuff! Let's create a *second* Ship object. The first is called `Jedi Starship` and has 10 `weaponPower`. Let's create `$otherShip`. And just like if 2 ships landed in your dock, one will have one name, and another will be named something different. Let's call this one: Imperial Shuttle. Set its `weaponPower` to 5 and a strength of 50:

play.php

```
↕ // ... lines 1 - 59
60 $myShip = new Ship();
↕ // ... lines 61 - 63
64 printShipSummary($myShip);
↕ // ... line 65
66 $otherShip = new Ship();
67 $otherShip->name = 'Imperial Shuttle';
68 $otherShip->weaponPower = 5;
69 $otherShip->strength = 50;
↕ // ... lines 70 - 73
```

These two separate objects both have data inside of them, but they function indepedently. The only thing they share is that they're both `Ship` objects, which means that they both share the same rules: that they have these 4 properties and these 3 methods. But the property *values* will be totally different between the two.

This means that we can print the second Ship's summary and see its specs:

play.php

```
// ... lines 1 - 65
$otherShip = new Ship();
$otherShip->name = 'Imperial Shuttle';
$otherShip->weaponPower = 5;
$otherShip->strength = 50;

echo '<hr/>';
printShipSummary($otherShip);
```

Now we get a print-out of two independent Ships where each has different data.

Chapter 6: Objects Interact

Since the goal of our app is to let two ships fight each other, things are getting interesting. For example, we could fight `$myShip` against `$otherShip` and see who comes out as the winner.

But first, let's imagine we want to know whose strength is higher. Of course, we could just write an `if` statement down here and manually check `$myShip`'s strength against `$otherShip`.

But we could also add a new method inside of the `Ship` class itself. Let's create a new method that'll tell us if one Ship's strength is greater than another's. We'll call it: `doesGivenShipHaveMoreStrength()`. And of course, it needs a Ship object as an argument:

play.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 43
44     public function doesGivenShipHaveMoreStrength($givenShip)
45     {
↕ // ... line 46
47     }
48
49 }
↕ // ... lines 50 - 86
```

So just like with our `printShipSummary()` function, when we call this function, we'll pass it a `Ship` object. What I want to do is compare the Ship object being passed to whatever Ship object we're calling this method on. Before I fill in the guts, I'll show you how we'll use it: if `$myShip->doesGivenShipHaveMoreStrength()` and pass it `$otherShip`. This will tell us if `$otherShip` has more strength than `$myShip` or not. If it does, we'll echo `$otherShip->name` has more strength. Else, we'll print the same thing, but say `$myShip` has more strength.

play.php

```
// ... lines 1 - 78
echo '<hr/>';

if ($myShip->doesGivenShipHaveMoreStrength($otherShip)) {
    echo $otherShip->name.' has more strength';
} else {
    echo $myShip->name.' has more strength';
}
```

Inside of the `doesGivenShipHaveMoreStrength()`, the magic `$this` will refer to the `$myShip` object, the one whose name is `Jedi Starship`. So all we need to do is return `$givenShip->strength` greater than my strength:

play.php

```
// ... lines 1 - 2
class Ship
{
// ... lines 5 - 43
    public function doesGivenShipHaveMoreStrength($givenShip)
    {
        return $givenShip->strength > $this->strength;
    }

}
// ... lines 50 - 86
```

Ok, let's try it! When we refresh, we see that the Imperial Shuttle has more strength. And that makes sense: the Imperial Shuttle has 50 compared with 0 for `$myShip`, because it's using the default value.

Let me add another separator and let's double-check to see if this is working by setting the strength of `$myShip` to 100.

Ok, refresh now! Now the Jedi Starship is stronger. Undo that last change.

So how cool is this? Not only can we have multiple objects, but they can interact with each other through methods like this one. I'll show you more of this later.

Chapter 7: My Editor is Confused

But before we go on, we need to help my editor. It's confused. Inside `printShipSummary()`, my editor doesn't seem to recognize the `sayHello()` method on `Ship`, it thinks it doesn't exist. But down at the bottom of the file, when I call `doesGivenShipHaveMoreStrength()`, it's not highlighted in yellow - that means my editor *does* see that this method exists. So what gives? Why doesn't it recognize the `sayHello()` function?

If you *just* look at the `printShipSummary()` function, all that my editor knows is that we're passing in *some* argument called `$someShip`, but it doesn't know *what* it is. Is it a string? A boolean? A `Ship` object? *We* know that this will be a `Ship` object, because we're creating `Ship` objects below and passing those as the argument. But our editor has no idea. And for that reason, it doesn't know to look on the `Ship` class to see that there's a `sayHello()` function.

You don't need to fix this, it's totally fine. But if you want to, you can use PHP documentation to give your editor a little hint about what the heck this `$someShip` variable is. By using this syntax, you can say this this is a `Ship` object:

play.php

```
// ... lines 1 - 50
/**
 * @param Ship $someShip
 */
function printShipSummary($someShip)
{
    echo 'Ship Name: '.$someShip->getName();
    echo '<hr/>';
    $someShip->sayHello();
    echo '<hr/>';
    echo $someShip->getNameAndSpecs(false);
    echo '<hr/>';
    echo $someShip->getNameAndSpecs(true);
}
// ... lines 64 - 86
```

And as soon as I do that, those ugly yellow highlights go away, and I even get auto-completion on new code I write.

As nice as this is, it makes no *functional* difference - your code isn't behaving any different than before. This is *just* a "nice" thing you can do to help you and your editor get along.

Chapter 8: Using Objects

This `play.php` file is cute, but it's not our real application. we *did* make this nice `Ship` class, so let's use. It'll clean up our code and give us more power. Sounds good to me!

Moving Ship into Ship.php

But first, the `Ship` class lives inside `play.php`, and this is just a garbage file we won't use anymore. Usually, a class will live all alone in its own file. Create a `lib/` directory, and a file called `Ship.php`. There's no technical reason why I called the directory `lib/`, it just sounds nice. And the same goes for the filename - we could call it anything. But to keep my sanity, putting the `Ship` class inside `Ship.php` makes a lot more sense than putting it inside of a file called `ThereIsDefinitelyNotAShipClassInHere.php`. So even though nothing technical forces us to do this, put one class per file, and then go celebrate how clean things look.

Go copy the `Ship` class, and put it into `Ship.php`:

lib/Ship.php

```php
<?php

class Ship
{
    public $name;

    public $weaponPower = 0;

    public $jediFactor = 0;

    public $strength = 0;

    public function sayHello()
    {
        echo 'Hello!';
    }

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

    public function getNameAndSpecs($useShortFormat)
    {
        if ($useShortFormat) {
            return sprintf(
                '%s: %s/%s/%s',
                $this->name,
                $this->weaponPower,
                $this->jediFactor,
                $this->strength
            );
        } else {
            return sprintf(
                '%s: w:%s, j:%s, s:%s',
                $this->name,
                $this->weaponPower,
                $this->jediFactor,
                $this->strength
            );
        }
    }

    public function doesGivenShipHaveMoreStrength($givenShip)
    {
        return $givenShip->strength > $this->strength;
```

```
47      }
48
49  }
```

Don't put a closing PHP tag, because you don't need it. PHP will reach the end of the file, and close it automatically.

I'll head back to `play.php`, just like when you have functions in an external file you have to require that file to have access to it. So we'll `require once __DIR__'/lib/Ship.php'`:

play.php

```
1  <?php
2
3  require __DIR__.'/lib/Ship.php';
↕  // ... lines 4 - 40
```

The `__DIR__` is a constant that points to this directory. So this makes sure that we're requiring exactly `/lib/Ship.php` relative to this file.

Get rid of requires?

If you're familiar with modern apps you'll notice that they don't have this `require` statement, we'll talk about that in the future. There is a way called `autoloading` to not even need require statements. But for now we do need it.

So now that we've moved that out let's refresh. Well look at that, it still works!

Creating Ship Objects... for Real

So now that we have this `Ship` class inside of a `Ship.php` file we can start using it from within our real application. From our `get_ships()` function, I don't want to return this array inside of an array thing anymore. I want to do awesome things like return objects.

We'll start with adding our require statement:

functions.php

```
1  <?php
2
3  require __DIR__.'/lib/Ship.php';
4
↕  // ... lines 5 - 107
```

Next, let's transform our brave starfighter into a `Ship` object. We do that with `$ship = new Ship();` and then we'll just start setting the details: `name = 'Jedi Starfighter'`, `weaponPower` of 5, `jediFactor` of 15 and `strength` of 30:

functions.php

```
// ... lines 1 - 4
function get_ships()
{
    $ships = array();

    $ship1 = new Ship();
    $ship1->name = 'Jedi Starfighter';
    $ship1->weaponPower = 5;
    $ship1->jediFactor = 15;
    $ship1->strength = 30;
    $ships['starfighter'] = $ship1;

    return $ships;
// ... lines 17 - 45
}
// ... lines 47 - 107
```

Perfect!

I'm commenting out this bottom array, we're not going to use that at all anymore. Instead we're going to return an array with just this one ship in it, we'll add more to our fleet later.

Remember, we're calling this back in `index.php`, in that file we call `get_ships();` that *use* to return an array of ship arrays. Now it returns an array of `Ship` objects, of which there will only be the one starfighter. Let's `var_dump` this to see what it looks like:

index.php

```
<?php
require __DIR__.'/functions.php';

$ships = get_ships();
var_dump($ships);die;
// ... lines 6 - 107
```

Take off the file name, so we load `index.php`:

```
http://localhost:8000
```

And there it is! We have an array with one item in it which is our `Ship` object. Look at those sweet spaceship stats.

Treat that Object like an Object (->)

Let's take that `var_dump` off and see what that does to our app. When we refresh we see an exciting error that tells us we cannot use object of type `Ship` as array on line 68. This is an error that you might see, so let's see what's happening on line 68:

index.php

```
// ... lines 1 - 66
                    <?php foreach ($ships as $ship): ?>
                        <tr>
                            <td><?php echo $ship['name']; ?></td>
                            <td><?php echo $ship['weapon_power']; ?></td>
                            <td><?php echo $ship['jedi_factor']; ?></td>
                            <td><?php echo $ship['strength']; ?></td>
                        </tr>
                    <?php endforeach; ?>
// ... lines 75 - 107
```

Ok, we're using `$ship['name']`. Before when each ship was an array, that made sense, now we know when you reference an object you need to use an arrow. So if you do have an object and you try to use the square bracket syntax that is the error that you will see. Lucky you!

I don't want to keep seeing errors so let's fix the other ones as well:

index.php

```
// ... lines 1 - 65
                    <?php foreach ($ships as $ship): ?>
                        <tr>
                            <td><?php echo $ship->name; ?></td>
                            <td><?php echo $ship->weaponPower; ?></td>
                            <td><?php echo $ship->jediFactor; ?></td>
                            <td><?php echo $ship->strength; ?></td>
                        </tr>
                    <?php endforeach; ?>
// ... lines 74 - 106
```

Awesome!

Head back to the browser and refresh and things are looking kinda better! Sweet! Well, at least we have a different fatal error in our dropdown here, cannot use object of type `Ship` as array, we'll fix that in a second.

Back in our editor, because this is an object we can use our methods on it. In our `Ship` class we have this `getName()` method:

lib/Ship.php

```
// ... lines 1 - 2
class Ship
{
// ... lines 5 - 17
    public function getName()
    {
        return $this->name;
    }
// ... lines 22 - 48
}
```

Down here let's switch out name for `getName();`

index.php

```
// ... lines 1 - 67
                            <td><?php echo $ship->getName(); ?></td>
// ... lines 69 - 106
```

When we refresh, we see it does the exact same thing.

Now, let's fix this little error we have in the select menu. In `index.php` you can see it's the same thing as before, we're using the `$ship` like an array, so change this to use `getName();` here and down there as well:

index.php

```
↕ // ... lines 1 - 81
82                          <select class="center-block form-control btn drp-
dwn-width btn-default btn-lg dropdown-toggle" name="ship1_name">
83                              <option value="">Choose a Ship</option>
84                              <?php foreach ($ships as $key => $ship): ?>
85                                  <option value="<?php echo $key; ?>"><?php
echo $ship->getName(); ?></option>
86                              <?php endforeach; ?>
87                          </select>
↕ // ... lines 88 - 91
92                          <select class="center-block form-control btn drp-
dwn-width btn-default btn-lg dropdown-toggle" name="ship2_name">
93                              <option value="">Choose a Ship</option>
94                              <?php foreach ($ships as $key => $ship): ?>
95                                  <option value="<?php echo $key; ?>"><?php
echo $ship->getName(); ?></option>
96                              <?php endforeach; ?>
97                          </select>
↕ // ... lines 98 - 106
```

Refresh, and now things look just fine!

We have this `getNameAndSpecs()` function, so perhaps when I'm choosing a ship I might want to see its important stats since I'm going to use it to save the day:

lib/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 22
23     public function getNameAndSpecs($useShortFormat)
24     {
↕ // ... lines 25 - 41
42     }
↕ // ... lines 43 - 48
49 }
```

So instead of `getName()` I'll use `getNameAndSpecs()`. First, I'm going to make the short format an optional argument so we don't always have to fill that in:

lib/Ship.php

```
↕ // ... lines 1 - 22
23     public function getNameAndSpecs($useShortFormat = false)
24     {
↕ // ... lines 25 - 41
42     }
↕ // ... lines 43 - 50
```

Let's make these updates in `index.php` and now refresh the browser:

index.php

```
↕ // ... lines 1 - 83
84                             <?php foreach ($ships as $key => $ship): ?>
85                                 <option value="<?php echo $key; ?>"><?php
   echo $ship->getNameAndSpecs(); ?></option>
86                             <?php endforeach; ?>
↕ // ... lines 87 - 93
94                             <?php foreach ($ships as $key => $ship): ?>
95                                 <option value="<?php echo $key; ?>"><?php
   echo $ship->getNameAndSpecs(); ?></option>
96                             <?php endforeach; ?>
↕ // ... lines 97 - 106
```

We see our specs format in the select menu -- cool!

And that's it, switching to an object is not that big of a deal. Next we'll talk about what these public things in `Ship` are doing inside of here and what else we can have.

Chapter 9: Private Access

Let me show you a problem with our app, there's nothing stopping me from going in and setting the strength to something like `Jar Jar Binks`:

functions.php

```
// ... lines 1 - 4
function get_ships()
{
// ... lines 7 - 8
    $ship1 = new Ship();
// ... lines 10 - 12
    $ship1->strength = 'Jar Jar Binks';
// ... lines 14 - 45
}
// ... lines 47 - 107
```

Clearly this value makes absolutely no sense at all for many reasons.

Sure enough, when we refresh Jar Jar Binks prints out as the strength in the select menu. The `Ship` class lets us give this really bad data. If we tried to battle, this would probably break our app since you can't compare a strength of 10 to Jar Jar Binks mathematically. But if you disagree, I would love to see your math in the comments.

To fix this, I'll get to show you another strength of classes. So far everything has been `public`:

lib/Ship.php

```
// ... lines 1 - 2
class Ship
{
    public $name;

    public $weaponPower = 0;

    public $jediFactor = 0;

    public $strength = 0;
// ... lines 12 - 48
}
```

Public name, weapon power and so on but I haven't told you what that means.

Making a Property private

There's actually three different words that can go here: `public`, `private` and `protected`, but we'll only worry about the first two for now. As soon as you make a property `private` it can't be accessed from outside of the class. I'll show you what I mean:

lib/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 10
11     private $strength = 0;
↕ // ... lines 12 - 48
49 }
```

Now that it's marked as `private` my editor is highlighting strength saying, "No no no, you can't access strength anymore." So from outside of the class it's illegal to access a private property.

And sure enough, when I refresh it says, "Fatal error: Cannot access private property".

Adding a Setter Method

This is called a visibility modifier. Once you make something `private` if you want someone from the outside to be able to interact with that property you'll need to add `public` functions to be able to do that. In this case down here, we can create what's called a `setter`: `public function setStrength($strength)` it will take an argument called `$strength`, which will be a number:

lib/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 22
23     public function setStrength($number)
24     {
25         $this->strength = $number;
26     }
↕ // ... lines 27 - 58
59 }
```

And then we'll set it on that property. You see that this does not highlight red, so a private property can still be accessed from within a class using the magic `$this` keyword. It just can't be accessed outside of the class.

Here, instead of accessing the strength property directly, we can access the `setStrength` method:

functions.php

```
↕ // ... lines 1 - 4
5 function get_ships()
6 {
↕ // ... lines 7 - 8
9     $ship1 = new Ship();
↕ // ... lines 10 - 12
13     $ship1->setStrength('Jar Jar Binks');
↕ // ... lines 14 - 45
46 }
↕ // ... lines 47 - 107
```

When we refresh, it gets further!

Adding a Getter Method

It gets past that setter and now we're down to line 71. We're still accessing the `strength` property:

index.php

```
// ... lines 1 - 65
                    <?php foreach ($ships as $ship): ?>
// ... lines 67 - 70
                        <td><?php echo $ship->strength; ?></td>
// ... line 72
                    <?php endforeach; ?>
// ... lines 74 - 106
```

So let's fix that right here. Since we can't reference that anymore we need to go in and make a public function `getStrength()` and it will go grab the value from that private property and return it to us:

lib/Ship.php

```
// ... lines 1 - 2
class Ship
{
// ... lines 5 - 27
    public function getStrength()
    {
        return $this->strength;
    }
// ... lines 32 - 58
}
```

In `index` we can say `getStrength()` and that should take care of the problem:

index.php

```
// ... lines 1 - 65
                    <?php foreach ($ships as $ship): ?>
// ... lines 67 - 70
                        <td><?php echo $ship->getStrength(); ?></td>
// ... line 72
                    <?php endforeach; ?>
// ... lines 74 - 106
```

Head back and refresh and it works! Alright!

Avoiding Jar Jar Binks

The reason we did this, is that when you have a `public` property there's no way to control who sets it from the outside. Anyone could have set the `strength` and they could have set it to any crazy string, negative number or bad Star Wars character, none of which make any sense. As

soon as you make it `private`, it means that outsiders are going to have to call public methods, and this gives us a cool opportunity to run a check inside of here to say, "Hey! Is the strength a number? If not, let's throw an error."

In `setStrength()` we'll put in an `if` statement with the `is_numeric()` function, and if it's not numeric, then we're going to throw a `new Exception()` with a helpful message:

lib/Ship.php

```
↕ // ... lines 1 - 22
23     public function setStrength($number)
24     {
25         if (!is_numeric($number)) {
26             throw new \Exception('Strength must be a number, duh!');
27         }
28
29         $this->strength = $number;
30     }
↕ // ... lines 31 - 64
```

In case you aren't familiar with exceptions, they're a special internal object to php. It stops the flow and shows an error.

Now when we refresh we get this nice helpful error. This message is for us the developer. Instead of the application running and tripping up later when we accidentaly put in a bad strength, we are notified immediately.

It even tells us that the error happened on `Ship.php` line 52 and we called the method on `functions.php` line 13. So let's go back into `functions.php` line 13 and of course there it is:

functions.php

```
↕ // ... lines 1 - 8
9      $ship1 = new Ship();
↕ // ... lines 10 - 12
13     $ship1->setStrength('Jar Jar Binks');
↕ // ... lines 14 - 107
```

We'll change that back to 30 and when we refresh life is good again:

functions.php

```
↕ // ... lines 1 - 8
9     $ship1 = new Ship();
↕ // ... lines 10 - 12
13    $ship1->setStrength(30);
↕ // ... lines 14 - 107
```

Make all the Things Private!

This idea of making your properties private and then adding getters and setters is really common. Even if you don't need the control like this now you might in the future. If you're already forcing outsiders to call your setter methods and you realize later that you need to do some sort of check you have the opportunity to do that by modifying your method.

A really common thing to do is to always make your properties private. So I'll update `jediFactor`, `weaponPower` and `name`:

lib/Ship.php

```
↕ // ... lines 1 - 2
3   class Ship
4   {
5       private $name;
6
7       private $weaponPower = 0;
8
9       private $jediFactor = 0;
10
11      private $strength = 0;
↕ // ... lines 12 - 101
102 }
```

The downside of this is that we'll need a `getName()`, `setName()`, `getWeaponPower()`, `setWeaponPower()` `getJediFactor()`, and a `setJediFactor()`.

That can be a lot of work and PHP doesn't give us a way to get around this, so we do need to write those. A lot of editors allow you to generate these, which is nice. In PHPStorm, go to code generate and then pick "Getters and Setters" and select the `weaponPower` and `jediFactor` fields. Name isn't in this list because we already have a `getName()`. I'll go back to code generate again and pick just setter this time and it recognizes that the `name` doesn't have a setter:

lib/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 65
66     public function getWeaponPower()
67     {
68         return $this->weaponPower;
69     }
↕ // ... lines 70 - 73
74     public function getJediFactor()
75     {
76         return $this->jediFactor;
77     }
↕ // ... lines 78 - 81
82     public function setName($name)
83     {
84         $this->name = $name;
85     }
↕ // ... lines 86 - 89
90     public function setWeaponPower($weaponPower)
91     {
92         $this->weaponPower = $weaponPower;
93     }
↕ // ... lines 94 - 97
98     public function setJediFactor($jediFactor)
99     {
100        $this->jediFactor = $jediFactor;
101    }
102 }
```

Now we have getters and setters on all of these properties. And by the way the name of this doesn't matter, we could get creative and call this `setWeaponPowerFooBar()`, but in your project try to be clear and concise.

Now that we've made everything private and we have these getters and setters, we need to use those everywhere instead of accessing the properties directly. Let's change this to `setName()`, this to `setWeaponPower()`, and `setJediFactor()`:

functions.php

```
↕ // ... lines 1 - 8
9     $ship1 = new Ship();
10    $ship1->setName('Jedi Starfighter');
11    $ship1->setWeaponPower(5);
12    $ship1->setJediFactor(15);
13    $ship1->setStrength(30);
↕ // ... lines 14 - 107
```

Maybe this feels like extra work right now, but if we had made it private in the beginning then we wouldn't have to go back and change them. Which is what I recommend that you do.

In `index.php` we have the same thing, we need to call `getWeaponPower()` and `getJediFactor()`:

index.php

```
↕ // ... lines 1 - 67
68                          <td><?php echo $ship->getName(); ?></td>
69                          <td><?php echo $ship->getWeaponPower(); ?>
</td>
70                          <td><?php echo $ship->getJediFactor(); ?></td>
71                          <td><?php echo $ship->getStrength(); ?></td>
↕ // ... lines 72 - 106
```

We're already calling `getStrength` and down here we're calling the public function `getNameAndSpecs`.

index.php

```
↕ // ... lines 1 - 84
85                                  <option value="<?php echo $key; ?>"><?php
echo $ship->getNameAndSpecs(); ?></option>
↕ // ... lines 86 - 106
```

So let's try that out and see if we missed anything. Refresh and everything looks really good and even the select menu shows up perfect. We're all set!

We now have all these wonderful hooks so that if anyone ever needs to get the `weaponPower` or set the `jediFactor`, we can do something before returning it. For example, in `getName()` you can actually use a `strtoupper` so whenever someone calls this we'll return the uppercase version:

```
public function getName()
{
```

```
    return strtoupper($this->name);
}
```

As cool as that is, I'll just undo it for now.

Creating all the Ship Objects

With all these private properties, getters and setters our `Ship` class is looking fit for action.

Back in `functions.php` we used to have these 3 other ships. Let's make object representations of those. We'll say `$ship2 = new Ship()` and then we just need to set the `name`, `weaponPower` `jediFactor` and `strength` for those other three ships. I'm going to save us a little bit of time and paste this in:

battle.php

```
↕ // ... lines 1 - 25
26 $ship1 = $ships[$ship1Name];
27 $ship2 = $ships[$ship2Name];
28
29 var_dump($ship1, $ship2);die;
↕ // ... lines 30 - 99
```

And there we go, `$ship2`, 3 and 4 have their data set on the array. What we're returning from here is an array of `Ship` objects.

When we go back and refresh everything looks perfect. And this is starting to look pretty good.

Next, we need to fix up `battle.php`. If we try to start a battle, we can see that it's super broken. And that makes sense, since we moved from arrays to objects and we haven't updated that page yet. But we've learned a ton so far so I'm sure this will be easy.

Chapter 10: Type Hinting?!

When we submit the form it goes to `battle.php` and we see this nasty error:

```
Argument 1 passed to battle() must be of the type array, object given
```

It comes from `functions.php` on line 74 and called on `battle.php` on line

1. Let's start with `battle.php`, sure enough we can see the problem is with the battle function, let me show you why. But first, let's dump `$ship1` and `$ship2`:

battle.php

```
↕ // ... lines 1 - 25
26 $ship1 = $ships[$ship1Name];
27 $ship2 = $ships[$ship2Name];
28
29 var_dump($ship1, $ship2);die;
↕ // ... lines 30 - 99
```

Let's see here, up top we call `getShips()` which returns an array of objects. Then we read the `$_POST` data to figure out which two ships are fighting. Then, down here, we get the ship objects off this array:

battle.php

```
↕ // ... lines 1 - 3
4 $ships = get_ships();
↕ // ... line 5
6 $ship1Name = isset($_POST['ship1_name']) ? $_POST['ship1_name'] : null;
7 $ship1Quantity = isset($_POST['ship1_quantity']) ?
  $_POST['ship1_quantity'] : 1;
8 $ship2Name = isset($_POST['ship2_name']) ? $_POST['ship2_name'] : null;
9 $ship2Quantity = isset($_POST['ship2_quantity']) ?
  $_POST['ship2_quantity'] : 1;
↕ // ... lines 10 - 25
26 $ship1 = $ships[$ship1Name];
27 $ship2 = $ships[$ship2Name];
↕ // ... lines 28 - 99
```

So this should dump two objects. And it does: we have the Jedi Starfighter and the Super Star Destroyer.

What is Type-Hinting?

Next, let's look in the `battle()` function which lives in `functions.php`:

functions.php

```
↕ // ... lines 1 - 73
74 function battle(array $ship1, $ship1Quantity, array $ship2,
   $ship2Quantity)
75 {
↕ // ... lines 76 - 120
121 }
↕ // ... lines 122 - 128
```

Here's the issue the `$ship1` and `$ship2` arguments have "array" in front of them. This tells PHP that this argument must be an array and if someone passes something other than an array, I want you to throw a huge error. So let's see that error again, it says:

```
Argument 1 passed to battle() must be of the type array, object given
```

This is called a type hint and the only purpose of a type hint in PHP is to get better errors: it doesn't change the behavior. We can just take the type hint off like this and that will fix the error:

functions.php

```
↕ // ... lines 1 - 73
74 function battle($ship1, $ship1Quantity, $ship2, $ship2Quantity)
75 {
↕ // ... lines 76 - 120
121 }
↕ // ... lines 122 - 128
```

And down here, knowing that `$ship1` is actually an object, instead of using the array syntax we can call the `getStrength()` method. Let's go ahead and dump `$ship1Health` to make sure it's working:

functions.php

```
↕ // ... lines 1 - 73
74 function battle($ship1, $ship1Quantity, $ship2, $ship2Quantity)
75 {
76     $ship1Health = $ship1->getStrength() * $ship1Quantity;
77     var_dump($ship1Health);die;
↕ // ... lines 78 - 121
122 }
↕ // ... lines 123 - 129
```

Just by removing the type hint it tells PHP to stop making sure it's an array, just let anything in and be ok with it. Refresh! This time it's printing out 60 which means it's printing out the ship's mighty strength correctly.

Type-Hinting Saves your Butt

The type hint is a useful thing, not from a functionality standpoint, but for knowing when you're doing something wrong. Let's go back to `battle.php` and pretend that something went wrong here by changing our object `$ship1` to the string `foo`:

battle.php

```
↕ // ... lines 1 - 28
29 $outcome = battle('foo', $ship1Quantity, $ship2, $ship2Quantity);
↕ // ... lines 30 - 97
```

When we refresh this time, we get this really weird error:

```
Call to a member function getStrength() on a non-object
```

You're going to see this error a lot and it's coming from line 76:

functions.php

```
↕ // ... lines 1 - 73
74 function battle($ship1, $ship1Quantity, $ship2, $ship2Quantity)
75 {
76     $ship1Health = $ship1->getStrength() * $ship1Quantity;
77     var_dump($ship1Health);die;
↕ // ... lines 78 - 121
122 }
↕ // ... lines 123 - 129
```

It happens whenever you use the arrow syntax on something that isn't an object. It's a fatal error and PHP just dies immediately. We know because I just passed `foo`, that `$ship1` is no longer an object, it's just a string. And when we call this on it, everything dies. The issue is that from the error message, it isn't exactly clear where the mistake is. It's telling us that the problem is on line 76 in `functions.php`. And sure, that is where the error occurred. But the real problem is in `battle.php` where we are passing in a bad value to the `battle()` function.

Type-Hinting with a Class

So in addition to type-hinting with the array, when we use objects we can type hint with the class name. Which means we can actually type `Ship` here and we can do that here as well:

```
functions.php
↕ // ... lines 1 - 73
74 function battle(Ship $ship1, $ship1Quantity, Ship $ship2, $ship2Quantity)
↕ // ... lines 75 - 129
```

That is the exact same thing. It says, "Hey, PHP, if something is passed to this argument that's not a `Ship` object, I want you to throw a very clear error." So let's go see this new error! Refresh and there it is:

```
Argument 1 passed to battle() must be an instance of Ship, string given
on line 29 `battle.php`
```

This time it's very clear: it says it should have been a `Ship` object, but you're passing a string and it points us to the exact right spot. So type-hinting is optional, but it's a really good idea because it will make your code easier to debug later. It also has a second benefit: as soon as I type hinted this `$ship1` variable here, all of a sudden my editor knew what type of object `$ship1` was and offered me autocomplete. So it knows about `getStrength()` and all the other methods on that object.

Now that we know that these are objects, let's fix this method for all the array syntaxes. Let's see here we have a few more:

functions.php

```
↕ // ... lines 1 - 73
74 function battle(Ship $ship1, $ship1Quantity, Ship $ship2, $ship2Quantity)
75 {
76     $ship1Health = $ship1->getStrength() * $ship1Quantity;
77     $ship2Health = $ship2->getStrength() * $ship2Quantity;
↕ // ... lines 78 - 96
97         $ship1Health = $ship1Health - ($ship2->getWeaponPower() *
   $ship2Quantity);
98         $ship2Health = $ship2Health - ($ship1->getWeaponPower() *
   $ship1Quantity);
↕ // ... lines 99 - 120
121 }
↕ // ... lines 122 - 129
```

And then down here, which is called from above we have one more:

functions.php

```
↕ // ... lines 1 - 123
124 function didJediDestroyShipUsingTheForce(array $ship)
125 {
126     $jediHeroProbability = $ship['jedi_factor'] / 100;
127
128     return mt_rand(1, 100) <= ($jediHeroProbability*100);
129 }
```

And notice that this one is not giving me autocomplete because it's being type hinted as an array. This function is called all the way up here, it's passing a `$ship1` and `$ship2`, so it's passing a ship object:

functions.php

```
↕ // ... lines 1 - 73
74 function battle(Ship $ship1, $ship1Quantity, Ship $ship2, $ship2Quantity)
75 {
↕ // ... lines 76 - 82
83         if (didJediDestroyShipUsingTheForce($ship1)) {
↕ // ... lines 84 - 87
88         }
89         if (didJediDestroyShipUsingTheForce($ship2)) {
↕ // ... lines 90 - 93
94         }
↕ // ... lines 95 - 120
121 }
↕ // ... lines 122 - 129
```

Let's change that type hint to be a `Ship` instead of an `array`:

functions.php

```
// ... lines 1 - 122
function didJediDestroyShipUsingTheForce(Ship $ship)
{
    $jediHeroProbability = $ship->getJediFactor() / 100;

    return mt_rand(1, 100) <= ($jediHeroProbability*100);
}
```

And now we will get that nice autocompletion which will make sure the object is being passed there. Awesome, this function looks good!

Let's go back and refresh. And of course I get that same error because I forgot to go back and put `$ship1` here:

battle.php

```
// ... lines 1 - 28
$outcome = battle($ship1, $ship1Quantity, $ship2, $ship2Quantity);
// ... lines 30 - 97
```

Fixing the Objects inside $outcome

Let's try that again. We still get an error, but if you look closely you'll see that it is happening farther down the page. The battle function is being called and it's all working. This new error is from line 61, at this point you can probably even spot what that is:

```
Cannot use object of type Ship as an array
```

That's another syntax thing that we need to change.

So, let's go down to line 61 and sure enough there it is:

battle.php

```
// ... lines 1 - 60
                    <?php echo $ship1Quantity; ?> <?php echo
$ship1['name']; ?><?php echo $ship1Quantity > 1 ? 's': ''; ?>
// ... lines 62 - 97
```

We'll call `getName()` on our `$ship1` and `$ship2` objects:

battle.php

```
// ... lines 1 - 60
                    <?php echo $ship1Quantity; ?> <?php echo $ship1->getName(); ?><?php echo $ship1Quantity > 1 ? 's': ''; ?>
                    VS.
                    <?php echo $ship2Quantity; ?> <?php echo $ship2->getName(); ?><?php echo $ship2Quantity > 1 ? 's': ''; ?>
// ... lines 64 - 98
```

Now, real quick back up on `battle()`, what it returns is this `$outcome` variable, and I'm going to show you what that actually is. Down here, let's do a `var_dump()` on `$outcome`, put a `die` statement and refresh:

battle.php

```
// ... lines 1 - 65
            <?php var_dump($outcome); ?>
// ... lines 67 - 98
```

So the `battle()` function returns an array with three different keys on it: `winning_ship` which is a `Ship` object, `losing_ship` which is a `Ship` object and whether or not Jedi powers were used to have a really awesome comeback win (`used_jedi_powers`).

The important part is that `winning_ship` and `losing_ship` are `Ship` objects. Let's just remove this `var_dump` real quick. Down here, when we reference `$outcome['winning_ship']` we know that this is an object:

battle.php

```
// ... lines 1 - 70
                        <?php echo $outcome['winning_ship']['name']; ?>
// ... lines 72 - 98
```

And we want to call `getName()` on it. The same thing here. And then we'll do the same thing here as well:

battle.php

```
// ... lines 1 - 69
                        <?php echo $outcome['winning_ship']->getName(); ?>
// ... lines 71 - 78
                        The <?php echo $outcome['winning_ship']->getName(); ?>
// ... lines 80 - 82
                            overpowered and destroyed the <?php echo $outcome['losing_ship']->getName() ?>s
// ... lines 84 - 97
```

We're converting from that array syntax to the object syntax.

Moment of truth, do we have a working battle page? SUCCESS! Super Star Destoyer won. Let's try it again. We'll throw 10 Jedi Star Ships at our one Super Star Destroyer and it wins again. Come one Jedi's get it together! If you try enough times the Jedis do come out with a victory.

The key take away here is because we have a `Ship` class, when we have a `Ship` object, we know exactly what we can do with it. This is cool because whenever we pass around a `Ship`, object we can type hint it with `Ship` and our editor instantly knows what that is and what methods we can call on it. We're giving definition to our data instead of passing around arrays which have unknown and probably inconsistent keys.

Chapter 11: The Constructor!

Here's the next challenge, sometimes I want my ships to be broken. So when a ship comes in it might be able to fight or it might be under repair. This is a property on a `Ship`, a `Ship` could be under repair or not. So I'll add this as a new private property that has a boolean which can be true or false:

lib/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 12
13     private $underRepair;
↕ // ... lines 14 - 108
109 }
```

The challenge is that whenever we create our objects inside of `functions.php`, I want to randomly set a `Ship` to be under repair or not under repair, and I want it to happen automatically. So just by creating a `Ship`, I want it to internally figure out if it is under repair or not.

public function __construct

This is where the idea of a constructor comes in. Whenever you create an object, you can actually hook into that process and say: "Hey whenever a `Ship` is created, call this function so I can set some stuff up." The way you do this is by creating a very special public function inside of your class called `__construct()`:

lib/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 14
15     public function __construct()
16     {
17         echo 'Automatically called!';
18     }
↕ // ... lines 19 - 108
109 }
```

The magic here is that name: it must be `__construct`. And just by having this it should be called everytime we say `new Ship()`.

Let's try it, refresh! And that's it: it's called four times, once for each of our ships. And it's called right when you say `new Ship()`, so if I throw in a `die()` statement right after creating a `Ship`, we're still going to see one of those called.

Setting up Data in __construct

Now we have a really powerful way to set up our data. Internally we can determine whether or not the `Ship` is under repair. We'll use `$this- underRepair = mt_rand(1, 100) < 30;`:

lib/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 14
15     public function __construct()
16     {
17         // randomly put this ship under repair
18         $this->underRepair = mt_rand(1, 100) < 30;
19     }
↕ // ... lines 20 - 114
115 }
```

This gives each ship a 30% chance of being broken...maybe a wing fell off!

To see this in action, let's cheat real quick and `var_dump` the `$ships` array. When we refresh we can see the first two ships are ready for action but the third one isn't. Refresh again and

they're all ready to fly. And a third time shows that the first two are busted and the other two are battle worthy. So that's working already!

Don't Create a Getter Function

Next, let's go into `index.php` and up top we have our table information. Let's include status which will tell us if our ship is under repair or not:

index.php

```
<?php
// ... lines 2 - 56
                <thead>
                    <tr>
                        <th>Ship</th>
// ... lines 60 - 62
                        <th>Status</th>
                    </tr>
                </thead>
// ... lines 66 - 118
```

Now so far, we don't have a way to access that new property. It's private and we don't have a getter or a setter and you don't necessarily need to create these. In fact, we don't want a setter: it's being set automatically inside of the class itself. But I *do* want to figure out if this `Ship` is functional or not. So what I'll do is create a new public function and I'll call it `isFunctional()`:

lib/Ship.php

```
// ... lines 1 - 2
class Ship
{
// ... lines 5 - 20
    public function isFunctional()
    {
        return !$this->underRepair;
    }
// ... lines 25 - 114
}
```

This will be the opposite of the `underRepair` value. If it is `underRepair`, then it is not functional and if it is functional then it is not `underRepair`. For the outsider whose going to be calling this function, they don't care what we're doing internally to figure that out.

Let's go back to `index.php` and create a nice if statement. If `$ship` is functional, `else`, and we'll put some adorable icons:

index.php

```
↕ // ... lines 1 - 66
67                     <?php foreach ($ships as $ship): ?>
68                         <tr>
↕ // ... lines 69 - 72
73                             <td>
74                                 <?php if ($ship->isFunctional()): ?>
75                                     <i class="fa fa-sun-o"></i>
76                                 <?php else: ?>
77                                     <i class="fa fa-cloud"></i>
78                                 <?php endif; ?>
79                             </td>
80                         </tr>
81                     <?php endforeach; ?>
↕ // ... lines 82 - 118
```

A sunshine for functional and a sad cloud for not functional.

Refresh and try it out, four sunshines and one cloud. Awesome!

Leveraging isFunctional() Like a Boss

Now it's really easy to do the next step. If a `Ship` is under repair, I don't want it to show up in this select menu. It's easy because we can just call `isFunctional` and it will take care of all the internal stuff for us. Down here we only want to print this out if the ship is in working order. And the same thing down here:

index.php

```
↕ // ... lines 1 - 91
92                                <?php foreach ($ships as $key => $ship): ?>
93                                    <?php if ($ship->isFunctional()): ?>
94                                        <option value="<?php echo $key; ?>"><?
    php echo $ship->getNameAndSpecs(); ?></option>
95                                    <?php endif; ?>
96                                <?php endforeach; ?>
↕ // ... lines 97 - 103
104                               <?php foreach ($ships as $key => $ship): ?>
105                                   <?php if ($ship->isFunctional()): ?>
106                                       <option value="<?php echo $key; ?>"><?
    php echo $ship->getNameAndSpecs(); ?></option>
107                                   <?php endif; ?>
108                               <?php endforeach; ?>
↕ // ... lines 109 - 118
```

Cool! Refresh, all sunshines. Refresh again -- there's a cloud. It looks like we're missing the Cloakshape Fighter due to repairs. And when you check the list, it isn't there! Perfect!

Adding Arguments to __construct

The `__construct()` function is something you are going to see a lot but it's a really easy idea. It just says if you have a function called `__construct()`, then it's automatically going to be called when you instantiate your object.

There is one other thing you can do: like most functions, it can have an argument. Let's put a `$name` argument here:

lib/Ship.php

```
↕ // ... lines 1 - 14
15      public function __construct($name)
16      {
↕ // ... line 17
18          // randomly put this ship under repair
19          $this->underRepair = mt_rand(1, 100) < 30;
20      }
↕ // ... lines 21 - 117
```

I'm not going to use it yet because I'm going to show you what happens when we do that.

Go back to `functions.php`. You can see that my editor is angry because it says required parameter `$name` missing:

functions.php

```
// ... lines 1 - 8
    $ship = new Ship();
// ... lines 10 - 129
```

So, you notice whenever we create a new `Ship` object, it's always `Ship()`, but you never pass anything in there. When you create an object, the stuff that goes in between the parenthesis are arguments that are passed to your `__construct()` function, if you have one. Because we have a `$name` argument here, now we need to pass a name there, just like that:

functions.php

```
// ... lines 1 - 8
    $ship = new Ship('Jedi Starfighter');
// ... lines 10 - 126
```

Now you can see that it is happy.

And what we can do inside of `Ship.php` is say, ok whatever `$name` they pass in, let's just set that to the `name` property:

lib/Ship.php

```
// ... lines 1 - 14
    public function __construct($name)
    {
        $this->name = $name;
        // randomly put this ship under repair
        $this->underRepair = mt_rand(1, 100) < 30;
    }
// ... lines 21 - 117
```

In `functions.php`, we don't have to call `setName()` anymore: we're passing it into the constructor and the name is being set that way. Let's update the other ones as well:

functions.php

```
// ... lines 1 - 8
    $ship = new Ship('Jedi Starfighter');
    //$ship->setName('Jedi Starfighter');
// ... lines 11 - 15
    $ship2 = new Ship('CloakShape Fighter');
// ... lines 17 - 21
    $ship3 = new Ship('Super Star Destroyer');
// ... lines 23 - 27
    $ship4 = new Ship('RZ-1 A-wing interceptor');
// ... lines 29 - 126
```

And we're good to go!

When to Pass Values to __construct

So, why would you do this? Why would you add a `$name` argument to the Ship's constructor and force it to be passed in versus the setter? It's really up to you. In our case, it doesn't make sense to have a `Ship` without a name. And before, that would have been possible had we just instantiated a new `Ship` and forgotten to call `setName()`. Then we would have been running around with a `Ship` object that had absolutely no name. How embarrassing.

Sometimes, when you have required information, you might choose to set them up as arguments to your constructor. It says "Hey, when you create a `Ship`, you have to pass me a name." We're not forcing the user to pass a `weaponPower`, `jediFactor` or `strength`, because those all have a nice default value of 0. So it makes sense not to force those, but we do force the name.

When you back up I just want you to realize that the `__construct` function is just like any other function. But if you give it that special name, it is automatically called and the arguments are passed to it.

And guess what! You just learned the fundamentals of object-oriented programming. Classes, check! Objects, super check! Methods, privacy, type-hinting, constructor and other stuff - all old news now. And there's even more great stuff to learn, like service classes, dependency injection, inheritance and interfaces. These will make you even more dangerous, and will also help you understand the outside libraries you use everyday. So keep going, and join us for episode 2.

Seeya next time!

With <3 from SymfonyCasts