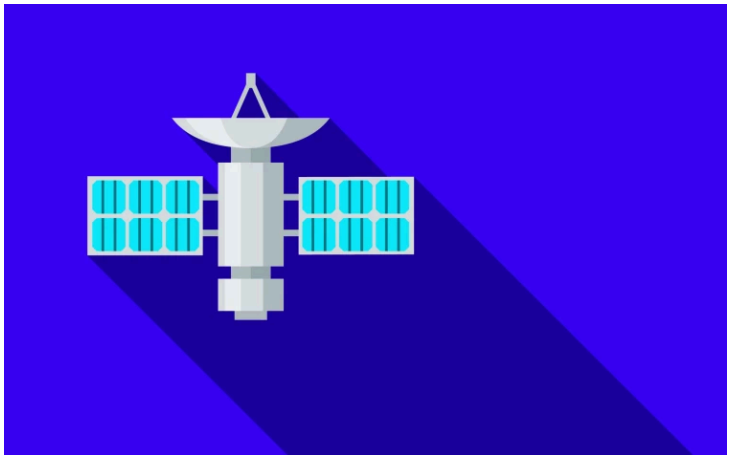
OOP (course 3): Inheritance, Abstract Classes, Interfaces and other amazing things



Chapter 1: Extends

Welcome back for Episode 3 of our Object Oriented Series! We're ready to get serious about Inheritance. And not just from that rich uncle of yours. I'm talking about extending classes, abstract classes, interfaces, stuff that really makes object oriented code nice but doesn't always look easy at first.

Don't worry this will all start to feel really familiar in a suprisingly small amount of time!

I'm already in the project that we've been working on through this series. If you don't have this yet download the code and use what's in the 'start' directory.

In my terminal I've also started the built in web server with `php -S localhost:8000`. Be careful to do that in the start directory of the project.

Creating a new RebelShip class

So far in our project we have just this one lonely ship object. We query things from the database and we load this ship. But exciting things are happening and we have a new problem! We want to model two different types of ships. We have normal ships from the empire and since those are kinda evil we also now want rebel ships to set them straight!

In the browser you can see we have two rebel ships in here coming from the database.

I would really like rebel ships to fundamentally work differently. For example, they break down less often and have higher jedi powers. Let me show you what I mean.

Create a new PHP class called `RebelShip`:

lib/Model/RebelShip.php

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

}
```

Easy! Since rebel ships aren't exactly like boring old Empire ships let's create a new class or blueprint that models how these work.

Head on into `bootstrap.php` and require the `RebelShip` file there:

bootstrap.php

```
↕ // ... lines 1 - 10
11 require_once __DIR__.'/lib/Model/RebelShip.php';
↕ // ... lines 12 - 15
```

We don't have an `autoloader` yet so we still have to worry about these require statements.

Rebel ships are different than Empire ones but they do share about 99% of their attributes. For example, they both have wings, fire power, defense power, etc.

Class Inheritance with extends

My first instinct should be to go into `Ship.php` and copy all of the contents and paste that into `RebelShip.php` since most of it will probably apply. But I shouldn't need to remind you that this would be a silly amount of duplication in our code which would make everyone sad. This is our chance to let classes help us not be sad by using the extends keyword.

By saying `class RebelShip extends Ship` everything that's in the `Ship` class is automatically inside of `RebelShip`:

lib/Model/RebelShip.php

```
↕ // ... lines 1 - 2
3 class RebelShip extends Ship
4 {
5
6 }
```

It's as if all the properties and methods of `Ship` are now a part of the `RebelShip` blueprint.

In `index.php` we can say `$rebelShip = new RebelShip('My new rebel ship');` and we can just add this to the `$ships` array:

index.php

```
↕ // ... lines 1 - 6
7 $ships = $shipLoader->getShips();
8
9 $rebelShip = new RebelShip('My new rebel ship');
10 $ships[] = $rebelShip;
↕ // ... lines 11 - 124
```

Remember, down here we iterate over the ships and call things like `getName()`, `getWeaponPower()` and `getJediFactor()` which don't actually live inside of `RebelShip`:

index.php

```
// ... lines 1 - 72
                    <?php foreach ($ships as $ship): ?>
                        <tr>
                            <td><?php echo $ship->getName(); ?></td>
                            <td><?php echo $ship->getWeaponPower(); ?></td>
// ... lines 77 - 85
                        </tr>
                    <?php endforeach; ?>
// ... lines 88 - 124
```

But when we refresh, it works perfectly!

Lesson number 1: when you have one class that extends another, it inherits (you'll hear that word a lot) all of the stuff inside that parent class. So we can call methods like `getName()` or `getNameAndSpecs()` on `RebelShip` because it inherits that from `Ship`.

Adding new Methods?

Really, `RebelShip` works just like a normal class. If you want to, you can add completely new functions. Let's do that with `public function getFavoriteJedi()` that has an array of some cool Jedis. Then use `array_rand` to select one of those:

lib/Model/RebelShip.php

```
// ... lines 1 - 2
class RebelShip extends Ship
{
    public function getFavoriteJedi()
    {
        $coolJedis = array('Yoda', 'Ben Kenobi');
        $key = array_rand($coolJedis);

        return $coolJedis[$key];
    }
}
```

Since this was all done on `RebelShip`, head over to `index.php` and call that method. `var_dump($rebelShip->getFavoriteJedi()` and you can see with my autocomplete it's showing me all of my public functions on both `Ship` and `RebelShip`:

index.php

```
// ... lines 1 - 8
$rebelShip = new RebelShip('My new rebel ship');
// ... lines 10 - 11
var_dump($rebelShip->getFavoriteJedi());die;
// ... lines 13 - 126
```

You can even see that the `RebelShip` methods are displayed bolder and methods from the parent class are lighter.

When we refresh, we see our favorite random Jedi, it works perfectly! Extending classes is great for reusing code without the sad duplication.

Chapter 2: Override

Let's take out this dummy code and get to the real stuff. Our database is created via this `init_db` script which you can execute from the command line whenever the mood strikes to make sure that your database is setup correctly. DING!

This creates a table with a `team` column. In here we can see that the first two team columns are team `rebel` and the second two are team `empire`. Since these two ships work differently, inside of our `ShipLoader` where we take that data and turn it into ship objects, I want to create ship objects for the empire and the rebels.

So let's do that, `if ($shipData['team'] == 'rebel')` which is the key inside the database. Then we'll have `$ship = new RebelShip($shipData['name']);`. Else, we'll throw in our normal ship line, which represents the Empire ship:

lib/Service/ShipLoader.php

```
↕ // ... lines 1 - 2
3 class ShipLoader
4 {
↕ // ... lines 5 - 44
45     private function createShipFromData(array $shipData)
46     {
47         if ($shipData['team'] == 'rebel') {
48             $ship = new RebelShip($shipData['name']);
49         } else {
50             $ship = new Ship($shipData['name']);
51         }
↕ // ... lines 52 - 57
58         return $ship;
59     }
↕ // ... lines 60 - 76
77 }
↕ // ... lines 78 - 79
```

Ok, this doesn't have anything to do with Object Oriented coding, it's just a nice example of a use case for multiple classes. We have a database table, and you can create different objects from that table. This is nice because we'll be able to have these two objects behave differently.

Overriding Class Methods

So far `RebelShip` and `Ship` have all the same stuff except for the one extra method I have on `RebelShip` that I'm not using.

If we go back and refresh, everything still works perfectly! Now, technically I'm fairly certain that two of these are `RebelShip` objects and two are `Ship` objects but we can't really tell right now. Clearly we need to add identifiers so we know who to cheer on.

To do this, start by adding `public function getType()` to our `Ship` and return a description, like 'Empire':

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 134
135     public function getType()
136     {
137         return 'Empire';
138     }
139 }
```

Since we added that to `Ship`, we can call `getType` on both `Ship` and `RebelShip`.

Back in `index.php` towards the bottom add a new column for this called `Type` and `echo $ship->getType();`:

index.php

```
↕ // ... lines 1 - 57
58              <table class="table table-hover">
↕ // ... line 59
60                  <thead>
61                      <tr>
↕ // ... lines 62 - 65
66                          <th>Type</th>
↕ // ... line 67
68                      </tr>
69                  </thead>
70                  <tbody>
71                      <?php foreach ($ships as $ship): ?>
72                          <tr>
↕ // ... lines 73 - 76
77                              <td><?php echo $ship->getType(); ?></td>
↕ // ... lines 78 - 84
85                          </tr>
86                      <?php endforeach; ?>
87                  </tbody>
88              </table>
↕ // ... lines 89 - 123
```

Back to the browser and refresh. Everything has joined to fight for the Empire! Which makes sense. Both ship classes use this same method.

Time for the next really powerful thing with inheritance. In addition to adding methods to a sub class like `RebelShip` you can override methods. Copy the `getType` from `Ship` and paste it into `RebelShip` and change what it returns to 'Rebel':

lib/Model/RebelShip.php

```
↕ // ... lines 1 - 2
3  class RebelShip extends Ship
4  {
↕ // ... lines 5 - 12
13     public function getType()
14     {
15         return 'Rebel';
16     }
↕ // ... lines 17 - 21
22 }
```

`RebelShip` copies the entire blue print of `Ship` but it can replace any of those pieces. When we refresh now, we have two 'Rebel' ships in addition to our two 'Empire' ships. Excellent!

Overridden Methods are not Called

A key part of this is that the parent `getType` class is never called for all rebel ship objects it is completely replaced. If I echo 'Parent Function' inside of `getType` in the `Ship` class and refresh, we see our ugly text echoing for the Empire ships and not the Rebel ships. This is thanks to our parent function not being called in `RebelShip`.

On to more methods, another one on `Ship` is `isFunctional` which we setup to have a 30% chance of a ship being broken, which is what our cute cloud here indicates:

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 16
17     public function __construct($name)
18     {
↕ // ... lines 19 - 20
21         $this->underRepair = mt_rand(1, 100) < 30;
22     }
23
24     public function isFunctional()
25     {
26         return !$this->underRepair;
27     }
↕ // ... lines 28 - 138
139 }
```

But, we all know that the Rebels are really scrappy and they don't have the luxury of letting their ships get broken. Even if they are kinda broken they still fly and make it work. Which is just one more reason why the rebels are awesome.

So I need to set this up so the Rebel ships are never showing as broken which we can do really easily by overriding `isFunctional` inside of `RebelShip`. Let's update this to `return true;` which will never show a rebel ship as broken:

lib/Model/RebelShip.php

```
↕ // ... lines 1 - 2
3 class RebelShip extends Ship
4 {
↕ // ... lines 5 - 17
18     public function isFunctional()
↕ // ... line 19
20         return true;
21     }
22 }
```

When we refresh now the Rebel ships always have sunshine, and the Empire ships sometimes have adorable clouds.

By having two classes we are starting to shape the different behaviors and properties of each, while still keeping most things in common and not duplicated.

Chapter 3: Protected Visibility

Let's keep making our Rebel ships work a bit differently than the Empire's. In this dropdown you can see a short summary of each ship that is currently functional. It shows their name, weapon power, jedi power and strength which all comes from the `getNameAndSpecs` function. But I would like a way to tell which ships in this list align with the rebels, so let's add that word in parenthesis at the end.

As usual to do that, we'll override this in `RebelShip`. Copy the `getNameAndSpecs` function and paste it over here. And then just add '(rebel)' at the end:

lib/Model/RebelShip.php

```
↕ // ... lines 1 - 2
3  class RebelShip extends Ship
4  {
↕ // ... lines 5 - 22
23     public function getNameAndSpecs($useShortFormat = false)
24     {
25         if ($useShortFormat) {
26             return sprintf(
27                 '%s: %s/%s/%s (Rebel)',
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 (Rebel)',
36                 $this->name,
37                 $this->weaponPower,
38                 $this->jediFactor,
39                 $this->strength
40             );
41         }
42     }
43 }
```

Now you may be thinking "guys, that's some serious code duplication...". Well you're absolutely right, and we'll get to fixing that!

For now what we've got is pretty straightforward, so let's refresh and... oh, check out our dropdown. We've got an `Undefinded property RebelShip::$name` error.

You can't access private things in sub-classes

Back in PhpStorm, you can see `$this->name` is highlighted with an error message of 'Member has private access'. Interesting. So far, I've told you that since `RebelShip` extends `Ship` it has access to everything inside of it like the properties and methods as if they also exist inside of `RebelShip`. However, this error really does seem to be saying something different than that.

We can see that in `Ship` there is a name property so why isn't this working? The answer has to do with this word `private` in front of `$name`:

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 6
7     private $name;
↕ // ... lines 8 - 138
139 }
```

All functions and properties so far are either `private` or `public`. If a function or a property is `private` it means you can only access it from within the ship class:

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 16
17     public function __construct($name)
18     {
19         $this->name = $name;
↕ // ... lines 20 - 21
22     }
↕ // ... lines 23 - 138
139 }
```

Like here where we say `$this->name`. As we can see here, `private` functions and properties can't be accessed inside of subclasses. So only things inside of the `Ship` class can access this `private $name;` property.

I always recommend that you make things `private` until you need to access them from outside of the class you're working in.

Introducing: protected

Now, there is another designation between `private` and `public` which is called `protected`. `Protected` works exactly like `private` except that subclasses can access it, so when we change it here the error goes away:

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 6
7     protected $name;
↕ // ... lines 8 - 138
139 }
```

Cool! Let's do a temporary fix for the error we're getting by making all of these things `protected`:

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 6
7     protected $name;
8
9     protected $weaponPower = 0;
10
11    protected $jediFactor = 0;
12
13    protected $strength = 0;
↕ // ... lines 14 - 138
139 }
```

Everything in our `RebelShip` file looks happy again so let's refresh. Ah ha! Our dropdown is back in business and showing the rebel designation.

I just mentioned that our fix was 'temporary' because I don't actually want to make these `protected` I really prefer to keep things `private` whenever possible. So even though these properties are `private` we have `public` functions that access them like `getName`, `getStrength`, `getWeaponPower`:

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 6
7     private $name;
8
9     private $weaponPower = 0;
10
11    private $jediFactor = 0;
12
13    private $strength = 0;
14
↕ // ... lines 15 - 33
34    public function getName()
35    {
36        return $this->name;
37    }
38
↕ // ... lines 39 - 47
48    public function getStrength()
49    {
50        return $this->strength;
51    }
52
↕ // ... lines 53 - 81
82    public function getWeaponPower()
83    {
84        return $this->weaponPower;
85    }
86
↕ // ... lines 87 - 89
90    public function getJediFactor()
91    {
92        return $this->jediFactor;
93    }
94
↕ // ... lines 95 - 138
139 }
```

Which means that in the subclass we can just use these instead of the properties. Let's go ahead and just change those in `RebelShip`. And to save me some effort I'll copy and paste these from the if to the else:

lib/Model/RebelShip.php

```
// ... lines 1 - 2
class RebelShip extends Ship
{
// ... lines 5 - 22
    public function getNameAndSpecs($useShortFormat = false)
    {
        if ($useShortFormat) {
            return sprintf(
                '%s: %s/%s/%s (Rebel)',
                $this->getName(),
                $this->getWeaponPower(),
                $this->getJediFactor(),
                $this->getStrength()
            );
        } else {
            return sprintf(
                '%s: w:%s, j:%s, s:%s (Rebel)',
                $this->getName(),
                $this->getWeaponPower(),
                $this->getJediFactor(),
                $this->getStrength()
            );
        }
    }
}
```

I like this, I mean I already have these `public` functions so why not use them? It allows me to keep these properties `private` which is looking ahead a little bit, but the more things you have marked as `private` the easier it's going to be to maintain and update your code later.

Back to the browser and refresh, and things still work!

private and protected Methods

Let's temporarily add a new `private` function to `Ship` called `getSecretDoorCodeToTheDeathstar()`. Since only Empire ships should have access to this you can see why setting it as `private` makes sense. And let's return the secret code 'Ra1nb0ws':

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 6
7     private $name;
8
9     private $weaponPower = 0;
10
11    private $jediFactor = 0;
12
↕ // ... lines 13 - 139
140   private function getSecretDoorCodeToTheDeathstar()
141   {
142       return 'Ra1nb0ws';
143   }
144 }
```

Over in `RebelShip` I should not be able to access this new function since we set it to `private`:

lib/Model/RebelShip.php

```
↕ // ... lines 1 - 2
3 class RebelShip extends Ship
4 {
↕ // ... lines 5 - 22
23    public function getNameAndSpecs($useShortFormat = false)
24    {
25        return $this->getSecretDoorCodeToTheDeathstar();
↕ // ... lines 26 - 43
44    }
45 }
```

We see the 'Member has private access' error so when we refresh we can check the dropdown to confirm that things aren't working.
`Fatal error: Call to private method Ship::getSecretDoorCodeToTheDeathstar()`
and we need to view the source to see the full error message.

But, if we go back and change that function to `protected`, our error is gone, the rebels have access to the secret door code and life is good:

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 139
140     protected function getSecretDoorCodeToTheDeathstar()
141     {
142         return 'Ra1nb0ws';
143     }
144 }
```

Remove all that nonsense. The moral of the story is this, make things `private` at first, `proctected` once you need to access them in a subclass. And finally `public` when you need to use it outside of its class and subclass.

Chapter 4: Calling Parent Class Methods

We covered that when you override a function, you override it entirely. In `RebelShip` we're overriding `getNameAndSpecs`:

lib/Model/RebelShip.php

```
↕ // ... lines 1 - 2
3 class RebelShip extends Ship
4 {
↕ // ... lines 5 - 22
23     public function getNameAndSpecs($useShortFormat = false)
24     {
25         if ($useShortFormat) {
26             return sprintf(
27                 '%s: %s/%s/%s (Rebel)',
28                 $this->getName(),
29                 $this->getWeaponPower(),
30                 $this->getJediFactor(),
31                 $this->getStrength()
32             );
33         } else {
34             return sprintf(
35                 '%s: w:%s, j:%s, s:%s (Rebel)',
36                 $this->getName(),
37                 $this->getWeaponPower(),
38                 $this->getJediFactor(),
39                 $this->getStrength()
40             );
41         }
42     }
43 }
```

which means that when this method is called on a `RebelShip` object the `getNameAndSpecs` inside of the original `Ship` class, i.e. the parent class, is never called. In this case that's sort of a problem because it leaves us with all this code duplication. It would be way better if we could somehow call the parent method, `getNameAndSpecs` inside of `Ship`, and then just add this '(rebel)' part to the end.

We saw in the last chapter, that from within `RebelShip` you can call methods that exist in the parent class as long as they are `public` or `protected`. Let's try that here. Add

`$val = $this->getNameAndSpecs()`. Pass in the `$useShortFormat` and then `$val .= ('Rebel');` and finally `return $val;`:

lib/Model/RebelShip.php

```
↕ // ... lines 1 - 2
3 class RebelShip extends Ship
4 {
↕ // ... lines 5 - 22
23     public function getNameAndSpecs($useShortFormat = false)
24     {
25         $val = $this->getNameAndSpecs($useShortFormat);
26         $val .= ' (Jedi)';
27
28         return $val;
29     }
30 }
```

Doesn't that look a whole lot nicer? Yes, yes it does.

Let's give our experiment here a try. Refresh! Hmmm something is wrong... `(!) Fatal error: Maximum`, let's view the source code since this error is stuck in our select box. Ah there we go: `(!) Fatal error: Maximum function nesting level of '200' reached, aborting!`. This means that we have a loop in our code, on index line 98 we call `getNameAndSpecs` and then on line 25 of `RebelShip` we call `getNameAndSpecs` again. This isn't working because when we call `$this->getNameAndSpecs`, it's literally calling this same method again inside of `RebelShip` not the parent function in `Ship`.

The parent Keyword

The way you get this to call the parent function is with a special key word called `parent::`:

lib/Model/RebelShip.php

```
↕ // ... lines 1 - 22
23     public function getNameAndSpecs($useShortFormat = false)
24     {
25         $val = parent::getNameAndSpecs($useShortFormat);
26         $val .= ' (Jedi)';
27
28         return $val;
29     }
↕ // ... lines 30 - 31
```

Let's try this again in our browser, refresh, and checking our dropdown everything is working again. Except, maybe I could use a space here to make things look nicer. There we go.

Don't worry about this `parent` keyword too much it's used in exactly one situation calling: a parent function that you overrode.

We'll see this `::` syntax again later when we talk about static things.

Chapter 5: Creating an Abstract Ship

There is one more thing that is special about the Rebel Ships. Since, they're the good guys we're going to give them some extra Jedi power.

Inside of `Ship` we have a `jediFactor` which is a value that is set from the database and a `getJediFactor()` function:

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 10
11     private $jediFactor = 0;
12
↕ // ... lines 13 - 89
90     public function getJediFactor()
91     {
92         return $this->jediFactor;
93     }
↕ // ... lines 94 - 138
139 }
```

In the `BattleManager` this is used to figure out if some super awesome Jedi powers are used during the battle.

For Rebel Ships, the Jedi Powers work differently than Empire ships. They always have at least some Jedi Power, sometimes there's a lot and sometimes it's lower, depending on what side of the galaxy they woke up on that day. So, instead of making this a dynamic value that we set in the datbase let's create a `public function getJediFactor()` that returns the `rand()` function with levels between 10 and 30:

lib/Model/RebelShip.php

```
↕ // ... lines 1 - 2
3 class RebelShip extends Ship
4 {
↕ // ... lines 5 - 30
31     public function getJediFactor()
32     {
33         return rand(10, 30);
34     }
35 }
```

Setting it up like this overrides the function in the `Ship` parent class.

Back in the browser, when we refresh we can see the Jedi Factor keeps changing on the first two Rebel ships only.

Fat Classes

Over in PhpStorm, when we look at this function now, `Ship` has a Jedi Factor property but `RebelShip` doesn't need that at all. Since `RebelShip` is extending `Ship` it is still inheriting that property. While this doesn't hurt anything it is a bit weird to have this extra property on our class that we aren't using at all. And this is also true for the `isFunctional()` method. In `RebelShip` it's always true:

lib/Model/RebelShip.php

```
↕ // ... lines 1 - 2
3 class RebelShip extends Ship
4 {
↕ // ... lines 5 - 17
18     public function isFunctional()
19     {
20         return true;
21     }
↕ // ... lines 22 - 34
35 }
```

But in `Ship` it reads from an `underRepair` property, and again that's just not needed in `RebelShip`:

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship
4 {
↕ // ... lines 5 - 14
15     private $underRepair;
↕ // ... lines 16 - 23
24     public function isFunctional()
25     {
26         return !$this->underRepair;
27     }
↕ // ... lines 28 - 138
139 }
```

The point being, `Ship` comes with extra stuff that we are inheriting but not using in `RebelShip`.

These classes are like blueprints, so maybe, instead of having `RebelShip` extend `Ship` and inherit all these things it won't use, we should have a third class that would hold the properties and methods that actually overlap between the two called `AbstractShip`. From here, `Ship` and `RebelShip` would both extend `AbstractShip` to get access to those common things.

This is a way of changing the class heirachy so that each class has only what it actually needs.

Creating an AbstractShip

Let's start this! Create a new PHP Class called `AbstractShip`:

lib/Model/AbstractShip.php

```
↕ // ... lines 1 - 2
3 class AbstractShip
4 {
↕ // ... lines 5 - 138
139 }
```

Since it is the most abstract idea of a ship in our project. To start, I'm going to copy everything out of the `Ship` class and paste it into `AbstractShip`:

lib/Model/AbstractShip.php

```
↕ // ... lines 1 - 2
3 class AbstractShip
4 {
5     private $id;
6
7     private $name;
8
9     private $weaponPower = 0;
↕ // ... lines 10 - 16
17     public function __construct($name)
18     {
19         $this->name = $name;
20         // randomly put this ship under repair
21         $this->underRepair = mt_rand(1, 100) < 30;
22     }
23
24     public function isFunctional()
25     {
26         return !$this->underRepair;
27     }
↕ // ... lines 28 - 138
139 }
```

I know this looks like where we just were, but trust me we're going somewhere with this.

Now, let's write `Ship extends AbstractShip`:

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship extends AbstractShip
4 {
5 }
```

And do the same thing in `RebelShip` changing it from `Ship` to `AbstractShip`:

lib/Model/RebelShip.php

```
↕ // ... lines 1 - 2
3 class RebelShip extends AbstractShip
↕ // ... lines 4 - 36
```

Then in `bootstrap` add our require line for our new class:

bootstrap.php

```
// ... lines 1 - 9
require_once __DIR__.'/lib/Model/AbstractShip.php';
require_once __DIR__.'/lib/Model/Ship.php';
require_once __DIR__.'/lib/Model/RebelShip.php';
// ... lines 13 - 16
```

Perfecto!

After just that change, refresh the browser and see what's happening. Hey nothing is broken, which makes sense since nothing has really changed in our code's functionality -- yet.

Let's trim down `AbstractShip` to only the items that are truly shared between our two ships.

First, `jediFactor` is specific to `Ship` so let's move it over there:

lib/Model/Ship.php

```
// ... lines 1 - 2
class Ship extends AbstractShip
{
    private $jediFactor = 0;
// ... lines 6 - 21
}
```

And then we'll update the references to it in `AbstractShip` to what the two classes share, which is a `getJediFactor()` function:

lib/Model/AbstractShip.php

```
↕ // ... lines 1 - 2
3 class AbstractShip
4 {
↕ // ... lines 5 - 50
51     public function getNameAndSpecs($useShortFormat = false)
52     {
53         if ($useShortFormat) {
54             return sprintf(
55                 '%s: %s/%s/%s',
56                 $this->name,
57                 $this->weaponPower,
58                 $this->getJediFactor(),
59                 $this->strength
60             );
61         } else {
62             return sprintf(
63                 '%s: w:%s, j:%s, s:%s',
64                 $this->name,
65                 $this->weaponPower,
66                 $this->getJediFactor(),
67                 $this->strength
68             );
69         }
70     }
↕ // ... lines 71 - 120
121 }
```

So let's copy and paste that function into `Ship`:

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship extends AbstractShip
4 {
↕ // ... lines 5 - 9
10     public function getJediFactor()
11     {
12         return $this->jediFactor;
13     }
↕ // ... lines 14 - 21
22 }
```

`RebelShip` already has one so that class is good to go already. Now in `AbstractShip` the `getJediFactor()` function will either call the version of the function in `Ship` or `RebelShip` depending on what is being loaded. There are a few other things I want to share with you about this, but we'll get to those later.

Now let's move `setJediFactor()` from `AsbtractShip` into `Ship`:

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship extends AbstractShip
4 {
↕ // ... lines 5 - 17
18     public function setJediFactor($jediFactor)
19     {
20         $this->jediFactor = $jediFactor;
21     }
22 }
```

and that should do it! Now, `Ship` still has all the functionality that it had before, it extends `AbstractShip`, and only contains its unique code. And `RebelShip` no longer inherits the `jediFactor` property and anything that works with it. Now each file is simpler, and only has the code that it actually needs. Back to the browser to test that everything still works. Oh look an error!

> *"Call to undefined method RebelShip::setJediFactor() on ShipLoader line 55."*

Let's check that out.

Ah, it's because down here when we create a ship object from the database, we always call `setJediFactor()` on it, and that doesn't make sense anymore. So we'll move this up and only call it for the `Ship` class:

lib/Service/ShipLoader.php

```
↕ // ... lines 1 - 2
3 class ShipLoader
4 {
↕ // ... lines 5 - 44
45     private function createShipFromData(array $shipData)
46     {
47         if ($shipData['team'] == 'rebel') {
48             $ship = new RebelShip($shipData['name']);
49         } else {
50             $ship = new Ship($shipData['name']);
51             $ship->setJediFactor($shipData['jedi_factor']);
52         }
53
54         $ship->setId($shipData['id']);
55         $ship->setWeaponPower($shipData['weapon_power']);
56         $ship->setStrength($shipData['strength']);
57
58         return $ship;
59     }
↕ // ... lines 60 - 76
77 }
↕ // ... lines 78 - 79
```

Refresh again, no error, perfect!

Back to `AbstractShip`, we have the `underRepair` property which is only used by `Ship`, so let's move that over:

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship extends AbstractShip
4 {
↕ // ... lines 5 - 6
7     private $underRepair;
↕ // ... lines 8 - 32
33     public function isFunctional()
34     {
35         return !$this->underRepair;
36     }
37 }
```

And, let's also move over the `isFunctional()` method from `AbstractShip` as well, since `RebelShip` has its own `isFunctional()` method already. Finally, the last place that this is used is in the construct function. The random number for under repair is set here, so just remove that one piece but leave the `$this->name = $name;` where it is since it is shared by both types of ships. In the `Ship` class we'll override the construct function, I'll keep the same argument. Using our trick

from earlier I'll call the `parent::__construct($name);` and then paste in the under repair calculation line:

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship extends AbstractShip
4 {
↕ // ... lines 5 - 8
9     public function __construct($name)
10    {
11        parent::__construct($name);
↕ // ... lines 12 - 13
14        $this->underRepair = mt_rand(1, 100) < 30;
15    }
↕ // ... lines 16 - 36
37 }
```

The last thing that's extra right now in the `AbstractShip` class is the `getType()` method. Both ships need a `getType()` function, but this one here is specific to the `Ship` class so we'll cut and paste that over:

lib/Model/Ship.php

```
↕ // ... lines 1 - 2
3 class Ship extends AbstractShip
4 {
↕ // ... lines 5 - 37
38    public function getType()
39    {
40        return 'Empire';
41    }
42 }
```

Back to the browser and refresh, everything looks great. The Rebel Ships aren't breaking and Jedi Factors are random, awesome!

This is the same functionality we had a second ago but the `RebelShip` class is a lot simpler. It only inherits what it actually uses from `AbstractShip`. Which means that our new class truly is the blueprint for the things that are shared by all the ship classes. `Ship` extends `AbstractShip` as does `RebelShip` and then each add their own specific code.

While this isn't a new concept, it is a new way of thinking of how to organize your "class hierarchy".

Chapter 6: Abstract Classes

Since everything seems to be working on our site, let's start a battle! Four Jedi Starfighters against three Super Star Destroyers. Engage.

Ahh an error!

> *"Argument 1 passed to BattleManager::battle() must be an instance of Ship, instance of RebelShip given"*

And this is apparently happening on `battle` line 32:

battle.php

```
↕ // ... lines 1 - 33
34 $battleResult = $battleManager->battle($ship1, $ship1Quantity, $ship2, $ship2Quantity);
↕ // ... lines 35 - 109
```

And `BattleManager` line 10:

lib/Service/BattleManager.php

```
↕ // ... lines 1 - 2
3 class BattleManager
4 {
↕ // ... lines 5 - 9
10     public function battle(Ship $ship1, $ship1Quantity, Ship $ship2, $ship2Quantity)
11     {
↕ // ... lines 12 - 56
57     }
↕ // ... lines 58 - 64
65 }
```

Back to our IDE and open up `battle.php`.

Trouble With Type Hints

Down on line 32, what we see is that `$ship1` is actually a `RebelShip` object, which makes sense since one of the ships I selected was a Rebel. But it expected that to be a normal `Ship` class. Over

in `BattleManager` look at the battle function to see the problem! We type hinted our arguments with the `Ship` class:

lib/Service/BattleManager.php

```
↕ // ... lines 1 - 2
3 class BattleManager
4 {
↕ // ... lines 5 - 9
10     public function battle(Ship $ship1, $ship1Quantity, Ship $ship2,
       $ship2Quantity)
11     {
↕ // ... lines 12 - 56
57     }
↕ // ... lines 58 - 64
65 }
```

Which tells PHP to only allow `Ship` classes or subclasses to be passed here.

The issue is that `RebelShip` is no longer a subclass of `Ship` and so now we have this error. The good news, the fix is simple! We don't care if we get a ship object in battle anymore. What we actually care about is that we get an `AbstractShip` object or any of its subclasses which we know includes `Ship` and `RebelShip`:

lib/Service/BattleManager.php

```
↕ // ... lines 1 - 2
3 class BattleManager
4 {
↕ // ... lines 5 - 9
10     public function battle(AbstractShip $ship1, $ship1Quantity, AbstractShip
       $ship2, $ship2Quantity)
11     {
↕ // ... lines 12 - 56
57     }
↕ // ... lines 58 - 64
65 }
```

Refresh and give this another try, we get the exact same error. Let's see we're being notified about something in `BattleManager` on line 58. Scroll down and look there:

lib/Service/BattleManager.php

```
↕ // ... lines 1 - 2
3 class BattleManager
4 {
↕ // ... lines 5 - 58
59     private function didJediDestroyShipUsingTheForce(Ship $ship)
60     {
↕ // ... lines 61 - 63
64     }
65 }
```

Ah yes, it's this type hinting right here. This function is called up here, and we pass it the ship object, so let's update this one to be expecting an `AbstractShip`:

lib/Service/BattleManager.php

```
↕ // ... lines 1 - 2
3 class BattleManager
4 {
↕ // ... lines 5 - 58
59     private function didJediDestroyShipUsingTheForce(AbstractShip $ship)
60     {
↕ // ... lines 61 - 63
64     }
65 }
```

Let's try this again! Cool, one more error! This one is having issues with `BattleResult::__construct()`. In our IDE we can see that when we instantiate the `BattleResult` object we pass it the `$winningShip` and the `$losingShip`:

lib/Service/BattleManager.php

```
↕ // ... lines 1 - 9
10     public function battle(AbstractShip $ship1, $ship1Quantity, AbstractShip $ship2, $ship2Quantity)
11     {
↕ // ... lines 12 - 55
56         return new BattleResult($usedJediPowers, $winningShip, $losingShip);
57     }
↕ // ... lines 58 - 66
```

Over in `BattleResult` we see that these are also typehinted with `Ship`. Update those two:

lib/Model/BattleResult.php

```
↕ // ... lines 1 - 2
3 class BattleResult
4 {
↕ // ... lines 5 - 13
14     public function __construct($usedJediPowers, AbstractShip $winningShip = null, AbstractShip $losingShip = null)
15     {
↕ // ... lines 16 - 18
19     }
↕ // ... lines 20 - 53
54 }
```

This is nice, our code is a lot more flexible now. Before, it had to be a `Ship` instance. Now we don't care what class you have as long as it extends `AbstractShip`.

Refresh again! Awesome, battling is back on.

What Methods are *really* on AbstractShip?

Now we have a few minor, but interesting, problems. First, in `AbstractShip` head down to `getNameAndSpecs()` and we see that `getJediFactor()` is highlighted with an error that says "Method `getJediFactor()` not found in class AbstractShip". Now, this is working because we do have a `getJediFactor()` method in `Ship` and `RebelShip`. When we call `getNameAndSpecs()` it's able to call `getJediFactor()`. But this should look a little fishy to you. There is no `getJediFactor()` function inside of `AbstractShip`, so just looking at this class you should feel suspicious and question whether or not this works.

Here's what's going on, we have an implied rule that says, "Yo, every class that extends `AbstractShip` must have a `getJediFactor()` function." If it doesn't everything is going to break when we call this function with a 'method not found' error. We aren't enforcing this rule. So we could easily create a new ship class, extend `AbstractShip`, and forget to add a `getJediFactor()` function. Our application would break and no battles would be happening. Sad times.

Abstract Functions to the Rescue

You're in luck, there's a feature called Abstract Classes that can handle this issue for us. I'll scroll up, but really the position of this doesn't matter. Add a new `abstract public function getJediFactor();`:

lib/Model/AbstractShip.php

```
↕ // ... lines 1 - 2
3 abstract class AbstractShip
4 {
↕ // ... lines 5 - 15
16     abstract public function getJediFactor();
↕ // ... lines 17 - 111
112 }
```

You may notice there are two different things about this. One is the word `abstract` before `public function` and the other is that I just have a semicolon on the end, I didn't actually make a function. The best part, this line doesn't add any functionality to our app, but it does force any class that extends this to have this method.

For example, if `RebelShip` didn't have this `getJediFactor()` method, then when we refresh the browser we'll get a huge error that says: "Hey! RebelShip must have a getJediFactor function!". This is because it has been defined as an abstract function inside of the parent class.

Up until now we could have instantiated an abstract ship directly with `new AbstractShip()` we didn't actually want to but it was possible. But, once you have an abstract function in here, that is no longer an option, it's only purpose then becomes to be a blueprint for other classes to extend.

Marking a Class as Abstract

Up here at the top of the file you can see that there is an error highlight with a message that says "Class must be declared abstract or implement method `getJediFactor()`". Once your class has an abstract function you need to add the `abstract` keyword in front of it, which enforces the rule that you can't say `new AbstractShip()`:

lib/Model/AbstractShip.php

```
↕ // ... lines 1 - 2
3 abstract class AbstractShip
↕ // ... lines 4 - 113
```

Now when we scroll down, we can see that `getJediFactor()` isn't highlighted anymore since we know that inside `AbstractShip` any subclasses will be forced to have that. Back to the browser and refresh! Everything still works just fine.

Related to this, there is one more little thing we need to fix up. Start in `ShipLoader`, notice that our `getShips()` and `findOneById()` functions still have PHPDoc above them that say they return a

ship object. That's not the biggest deal, but it would be more accurate if it said `AbstractShip` - because this actually returns a mixture of `RebelShip` and `Ship` objects:

lib/Service/ShipLoader.php

```
↕ // ... lines 1 - 2
3 class ShipLoader
4 {
↕ // ... lines 5 - 11
12     /**
13      * @return AbstractShip[]
14      */
15     public function getShips()
16     {
↕ // ... lines 17 - 25
26     }
27
28     /**
29      * @param $id
30      * @return AbstractShip
31      */
32     public function findOneById($id)
33     {
↕ // ... lines 34 - 42
43     }
↕ // ... lines 44 - 76
77 }
↕ // ... lines 78 - 79
```

Now check this out, inside of `index.php`, remember this `$ships` variable we get by calling that `getShips()` function?

index.php

```
↕ // ... lines 1 - 6
7 $ships = $shipLoader->getShips();
↕ // ... lines 8 - 123
```

So that returns an array of `AbstractShip` objects. When we loop over it, the `isFunctional()` and the `getType()` functions aren't found:

index.php

```
↕ // ... lines 1 - 70
71                     <?php foreach ($ships as $ship): ?>
↕ // ... lines 72 - 74
75                             <td><?php echo $ship->getJediFactor(); ?></td>
↕ // ... line 76
77                             <td><?php echo $ship->getType(); ?></td>
↕ // ... lines 78 - 85
86                     <?php endforeach; ?>
↕ // ... lines 87 - 123
```

The message here says "Method `getType()` not found in class `AbstractShip`". This is just like the `getJediFactor()` problem we just fixed. We don't have a `getType()` function inside of here. Both of our subclasses do, which is why our app still works, but technically we're not enforcing that. Any new subclasses to `AbstractShip` could easily end up missing these functions which would again stop all the battles.

What we need is another abstract public function for `getType()` and `isFunctional()`:

lib/Model/AbstractShip.php

```
↕ // ... lines 1 - 2
3  abstract class AbstractShip
4  {
↕ // ... lines 5 - 20
21     abstract public function getType();
22
↕ // ... lines 23 - 25
26     abstract public function isFunctional();
↕ // ... lines 27 - 121
122 }
```

This doesn't change anything in our application, it just forces our subclasses to have those methods. And now `index.php` is really happy again!

That's the power of abstract classes, you can have a whole bunch of shared logic in there, but if there are a couple of pieces that you can't fill in in your abstract class because they are specific to your subclasses, no problem! Just put them in there as abstract functions and your subclasses will be forced to have those.

In my example these are abstract public functions but you could also have abstract protected functions as well. Which one you use just depends on your use case. It's a very powerful feature of object oriented code.

Chapter 7: Broken Ship

Here's the really beautiful thing about abstract classes. You may create some of these because you have a situation similar to the one we've been working on in this project. Or, you may be using someone else's code like a third party library that you downloaded via the Composer package manager. You might even read in that library's documentation that if you want to create a new ship class you just need to extend `AbstractShip`.

What's really great is that `AbstractShip` now tells you exactly what you need to do to create a new ship class with its three abstract functions that you must fill in:

lib/Model/AbstractShip.php

```
↕ // ... lines 1 - 2
3 abstract class AbstractShip
4 {
↕ // ... lines 5 - 15
16     abstract public function getJediFactor();
↕ // ... lines 17 - 20
21     abstract public function getType();
↕ // ... lines 22 - 25
26     abstract public function isFunctional();
↕ // ... lines 27 - 121
122 }
```

A third group has joined the battle and we have a new type of ship. They're not very good mechanics, so we'll call this a broken ship. This is simple, the ship is always broken.

Create a new php class called `BrokenShip`. Of course now make it extend `AbstractShip`:

lib/Model/BrokenShip.php

```
↕ // ... lines 1 - 2
3 class BrokenShip extends AbstractShip
4 {
5
6 }
```

Let's pretend like we don't know that there are any abstract methods in the parent class. So we won't do anything here except putting in the extends code. Head over to `bootstrap.php` and require our useless new `BrokenShip`:

bootstrap.php

```
↕ // ... lines 1 - 12
13 require_once __DIR__.'/lib/Model/BrokenShip.php';
↕ // ... lines 14 - 17
```

Back in `index.php` for now, let's just add `$brokenShip = new BrokenShip();` and add it to our ships array:

index.php

```
↕ // ... lines 1 - 8
9 $brokenShip = new BrokenShip('Just a hunk of metal');
10 $ships[] = $brokenShip;
↕ // ... lines 11 - 126
```

We can do this because we know that `BrokenShip` extends `AbstractShip`. And down here, when we use those ship objects we're just calling methods on the `AbstractShip`.

Back to the browser, refresh! Yes, what a huge beautiful error. It says:

> *"Class BrokenShip contains 3 abstract methods and must therefore be declared abstract or implement the remaining methods."*

And then it goes on and lists the methods.

In other words, it's saying "Hey buddy! You need to add those three methods into this class!" It's always giving you an out to declare the class abstract if you want to, and you might do this if you wanted an abstract class inside an abstract class with some additional public functions. But we've all seen where that goes in the move inception.

In our case we want this to be a concrete class, meaning one that we can instantiate. When we go over to `AbstractShip` we say "Oh yea, I see there's a `getJediFactor` function that I need to add." Take off the abstract to turn it into a real function, and since this ship is always broken we don't care about the Jedi factor so let's just return 0:

lib/Model/BrokenShip.php

```
↕ // ... lines 1 - 2
3 class BrokenShip extends AbstractShip
4 {
5     public function getJediFactor()
6     {
7         return 0;
8     }
↕ // ... lines 9 - 18
19 }
```

When we refresh after that we get the same error, but we're down to just 2 missing abstract methods, `getType` and `isFunctional`.

Head back into `AbstractShip` and grab those, pop off the abstract word at the beginning after we paste them into `BrokenShip`. And we'll fill in the details of `getType` by returning 'Broken'. And we'll fill in `isFunctional` by returning false:

lib/Model/BrokenShip.php

```
// ... lines 1 - 2
class BrokenShip extends AbstractShip
{
// ... lines 5 - 9
    public function getType()
    {
        return 'Broken';
    }

    public function isFunctional()
    {
        return false;
    }
}
```

Without really knowing anything I extended `AbstractShip` and that class told me exactly what I needed to have in my subclasses.

And when we refresh, we have one more error! We're missing argument 1 to `AbstractShip::__construct`. That's my bad. In `index.php` here `BrokenShip` still has a constructor argument which is the name so let's not forget to fill that in with "I am so broken":

index.php

```
// ... lines 1 - 8
$brokenShip = new BrokenShip('Just a hunk of metal');
// ... lines 10 - 126
```

Refresh again, and things look great! We've got our four original ships and our new broken one. Which is always broken with its little cute cloud.

We didn't have to update any of our other code because `BrokenShip` extends `AbstractShip` and has all the same methods as everything else which leaves everything working just as beautifully as before. Blueprint classes for the win!

Chapter 8: Abstracting a Class into 2 Smaller Pieces

To get our ships we use `ShipLoader` which queries the database and creates ship objects. This `queryForShips()` goes out, selects all the ships, and then later it is passed to this nice `createShipFromData()` function down here:

lib/Service/ShipLoader.php

```
↕ // ... lines 1 - 2
3 class ShipLoader
4 {
↕ // ... lines 5 - 14
15     public function getShips()
16     {
17         $ships = array();
18
19         $shipsData = $this->queryForShips();
20
21         foreach ($shipsData as $shipData) {
22             $ships[] = $this->createShipFromData($shipData);
23         }
24
25         return $ships;
26     }
↕ // ... lines 27 - 58
59 }
↕ // ... lines 60 - 61
```

This is the one we've been working in that creates the objects.

- Step 1: Query the database
- Step 2: Turn that data into objects

Suppose that we have a new requirement, sometimes we're going to get the ship data from the database but other times it will come from a different source, like a JSON file.

In the resources directory there's a new `ship.json` file, as you can see this holds the same info as we have in the database:

resources/ships.json

```
// ... line 1
[
    {
        "id": "1",
        "name": "Jedi Starfighter",
        "weapon_power": "5",
        "jedi_factor": "15",
        "strength": "30",
        "team": "rebel"
    },
// ... lines 11 - 26
    {
        "id": "4",
        "name": "RZ-1 A-wing interceptor",
        "weapon_power": "4",
        "jedi_factor": "4",
        "strength": "50",
        "team": "empire"
    }
]
```

Now why would we want our application to sometimes load from the database and other times from a JSON file? Say that when we're developing locally we don't have access to our database, so we use a JSON file. But when we push to production we'll switch back to the real database. Or, suppose that our ship library is so awesome that someone else wants to reuse it. However, this fan doesn't use a database, they only load them from JSON.

This leaves us needing to make our `ShipLoader` more generic.

Right now, all of the logic of querying things from the database is hardcoded in here. So let's create a new class whose only job is to load ship data through the database, or PDO.

Create a new class called `PdoShipStorage`:

lib/Service/PdoShipStorage.php

```
// ... lines 1 - 2
class PdoShipStorage
{
// ... lines 5 - 31
}
```

Looking back inside `ShipLoader` there are two types of queries that we make:

lib/Service/ShipLoader.php

```
↕ // ... lines 1 - 2
3 class ShipLoader
4 {
↕ // ... lines 5 - 31
32     public function findOneById($id)
33     {
34         $statement = $this->getPDO()->prepare('SELECT * FROM ship WHERE id =
   :id');
↕ // ... lines 35 - 42
43     }
44
↕ // ... lines 45 - 68
69     private function queryForShips()
70     {
71         $statement = $this->getPDO()->prepare('SELECT * FROM ship');
↕ // ... lines 72 - 75
76     }
77 }
↕ // ... lines 78 - 79
```

Sometimes we query for all of the ships and sometimes we query for just one ship by ID.

Back to our `PdoShipStorage` I'll create two methods, to cover both of those actions. First, create a `public function fetchAllShipsData()` which we'll fill out in just one second. Now, add `public function fetchSingleShipData()` and pass it the id that we want to query for:

lib/Service/PdoShipStorage.php

```
↕ // ... lines 1 - 2
3 class PdoShipStorage
4 {
↕ // ... lines 5 - 11
12     public function fetchAllShipsData()
13     {
↕ // ... lines 14 - 17
18     }
19
20     public function fetchSingleShipData($id)
21     {
↕ // ... lines 22 - 30
31     }
32 }
```

Before we go any further head back to our `bootstrap.php` file and make sure that we require this:

bootstrap.php

```
// ... lines 1 - 14
require_once __DIR__.'/lib/Service/PdoShipStorage.php';
// ... lines 16 - 19
```

Perfect!

What I want to do is move all the querying logic from `ShipLoader` into this `PdoShipStorage` class. Let's start with the logic that queries for one ship and pasting that over here:

lib/Service/PdoShipStorage.php

```
// ... lines 1 - 2
class PdoShipStorage
{
// ... lines 5 - 19
    public function fetchSingleShipData($id)
    {
        $statement = $this->pdo->prepare('SELECT * FROM ship WHERE id =
:id');
        $statement->execute(array('id' => $id));
        $shipArray = $statement->fetch(PDO::FETCH_ASSOC);

        if (!$shipArray) {
            return null;
        }

        return $shipArray;
    }
}
```

Notice, that we're not returning an object here this is just a really dumb class that returns data, an array in our case.

There is one problem, we have a `getPdo()` function inside of `ShipLoader` that references a pdo property. Point being, our PDO storage needs access to the PDO object, so we're going to use *dependency injection*, a topic we covered a lot in episode 2 . Add `public function __construct(PDO $pdo)` and store it as a property with `$this->pdo = $pdo;`:

lib/Service/PdoShipStorage.php

```
// ... lines 1 - 2
class PdoShipStorage
{
    private $pdo;

    public function __construct(PDO $pdo)
    {
        $this->pdo = $pdo;
    }
// ... lines 11 - 31
}
```

If this pattern is new to you just head back and watch the dependency injection video in episode 2 of the OO series.

Here we're saying, whomever creates our PDO ship storage class must pass in the pdo object. This is cool because we need it. Now I can just reference the property there directly.

Back in `ShipLoader` copy the entire `queryForShips()` and paste that into `fetchAllShipsData()` and once again reference the pdo property:

lib/Service/PdoShipStorage.php

```
// ... lines 1 - 2
class PdoShipStorage
{
// ... lines 5 - 11
    public function fetchAllShipsData()
    {
        $statement = $this->pdo->prepare('SELECT * FROM ship');
        $statement->execute();

        return $statement->fetchAll(PDO::FETCH_ASSOC);
    }
// ... lines 19 - 31
}
```

Now we have a class whose only job is to query for ship stuff, we're not using it anywhere yet, but it's fully ready to go. So let's use this inside of `ShipLoader` instead of the PDO object. Since we don't need PDO to be passed anymore swap that out for a `PdoShipStorage` object. Let's update that in a few other places and change the property to be called `shipStorage`:

lib/Service/ShipLoader.php

```
↕ // ... lines 1 - 2
3 class ShipLoader
4 {
5     private $shipStorage;
6
7     public function __construct(PdoShipStorage $shipStorage)
8     {
9         $this->shipStorage = $shipStorage;
10    }
↕ // ... lines 11 - 58
59 }
↕ // ... lines 60 - 61
```

Cool!

Down in `getShips()` we used to call `$this->queryForShips();` but we don't need to do that anymore! Instead, say `$this->shipStorage->fetchAllShipsData();`:

lib/Service/ShipLoader.php

```
↕ // ... lines 1 - 2
3 class ShipLoader
4 {
↕ // ... lines 5 - 54
55    private function queryForShips()
56    {
57        return $this->shipStorage->fetchAllShipsData();
58    }
59 }
↕ // ... lines 60 - 61
```

Perfect, now scroll down and get rid of the `queryForShips()` function all together: we're not using that anymore. And while we're cleaning things out also delete this `getPDO()` function. We can delete this because up here where we reference it in `findOneById()` we'll do the same thing. Remove all the pdo querying logic, and instead say `shipArray = $this->shipStorage->fetchSingleShipData();` and pass it the ID:

lib/Service/ShipLoader.php

```
// ... lines 1 - 2
class ShipLoader
{
// ... lines 5 - 31
    public function findOneById($id)
    {
        $shipArray = $this->shipStorage->fetchSingleShipData($id);
// ... lines 35 - 36
    }
// ... lines 38 - 58
}
// ... lines 60 - 61
```

This class now has no query logic anywhere.

All we know is that we're passed in some `PdoShipStorage` object and we're able to call methods on it. It can make the queries and talk to whatever database it wants to, that's it's responsibility. In here we're just calling methods instead of actually querying for things.

`ShipLoader` and `PdoShipStorage` are now fully setup and functional. The last step is going into our container which is responsible for creating all of our objects to make a couple of changes. For example, when we have `new ShipLoader` we don't want to pass a pdo object anymore we want to pass in `PdoShipStorage`.

Just like before, create a new function called `getShipStorage()` and make sure we have our property up above. The `getShipStorage()` method is going to do exactly what you expect it to do. Instantiate a new `PdoShipStorage` and return it. The ship's storage class does need PDO as its first constructor argument which we do with `new PdoShipStorage($this->getPDO());`:

lib/Service/Container.php

```
↕ // ... lines 1 - 2
3 class Container
4 {
↕ // ... lines 5 - 12
13     private $shipStorage;
14
↕ // ... lines 15 - 49
50     public function getShipStorage()
51     {
52         if ($this->shipStorage === null) {
53             $this->shipStorage = new PdoShipStorage($this->getPDO());
54         }
55
56         return $this->shipStorage;
57     }
↕ // ... lines 58 - 69
70 }
```

Up in `getShipLoader()`, now pass `$this->getShipStorage()`:

lib/Service/Container.php

```
↕ // ... lines 1 - 2
3 class Container
4 {
↕ // ... lines 5 - 40
41     public function getShipLoader()
42     {
43         if ($this->shipLoader === null) {
44             $this->shipLoader = new ShipLoader($this->getShipStorage());
45         }
↕ // ... lines 46 - 47
48     }
↕ // ... lines 49 - 69
70 }
```

Everything used to be in `ShipLoader`, including the query logic. We've now split things up so that the query logic is in `PdoShipStorage` and in `ShipLoader` you're just calling methods on the `shipStorage`. Its real job is to create the objects from the data, wherever that data came from. In `Container.php` we've wired all this stuff up.

Phew, that was a lot of coding we just did, but when we go to the browser and refresh, everything still works exactly the same as before. That was a lot of internal refactoring. In `index.php` as always we still have `$shipLoader->getShips()`:

index.php

```
// ... lines 1 - 6
$ships = $shipLoader->getShips();
// ... lines 8 - 126
```

And that function still works as it did before, but the logic is now separated into two pieces.

The cool thing about this is that our classes are now more focused and broken into smaller pieces. Initially we didn't need to do this, but once we had the new requirement of needing to load ships from a JSON file this refactoring became necessary. Now let's see how to actually load things from JSON instead of PDO.

Chapter 9: AbstractShipStorage

Our goal is to make `ShipLoader` load things from the database or from a JSON file. In the resources directory I've already created a `JsonFileShipStorage` class.

Copy that into the service directory and let's take a look inside of here:

lib/Service/JsonFileShipStorage.php

```
// ... lines 1 - 2
class JsonFileShipStorage
{
    private $filename;

    public function __construct($jsonFilePath)
    {
        $this->filename = $jsonFilePath;
    }

    public function fetchAllShipsData()
    {
        $jsonContents = file_get_contents($this->filename);

        return json_decode($jsonContents, true);
    }

    public function fetchSingleShipData($id)
    {
        $ships = $this->fetchAllShipsData();

        foreach ($ships as $ship) {
            if ($ship['id'] == $id) {
                return $ship;
            }
        }

        return null;
    }
}
```

It has all of the same methods as `PdoShipStorage`. Except that this loads from a JSON file instead of querying from a database. Let's try and use this in our project.

First, head over to `bootstrap` of course and require `JsonFileShipStorage.php`:

bootstrap.php

```
↕ // ... lines 1 - 15
16 require_once __DIR__.'/lib/Service/JsonFileShipStorage.php';
↕ // ... lines 17 - 19
```

In theory since this class has all the same methods as `PdoShipStorage` we should be able to pass a `JsonFileShipStorage` object into `ShipLoader` and everything should just work. Really, the only thing `ShipLoader` should care about is that it's passed an object that has the two methods it's calling: `fetchAllShipsData()` and `fetchSingleShipData()`:

lib/Service/ShipLoader.php

```
↕ // ... lines 1 - 2
3  class ShipLoader
4  {
↕ // ... lines 5 - 31
32     public function findOneById($id)
33     {
34         $shipArray = $this->shipStorage->fetchSingleShipData($id);
↕ // ... lines 35 - 36
37     }
↕ // ... lines 38 - 54
55     private function queryForShips()
56     {
57         return $this->shipStorage->fetchAllShipsData();
58     }
59 }
↕ // ... lines 60 - 61
```

In `Container` let's give this a try. Down in `getShipStorage()` let's say, `$this->shipStorage = new JsonFileShipStorage()`. And we'll give it a path to our JSON of `__DIR__.'/../../resources/ships.json'`:

lib/Service/Container.php

```
↕ // ... lines 1 - 2
3 class Container
4 {
↕ // ... lines 5 - 49
50     public function getShipStorage()
51     {
52         if ($this->shipStorage === null) {
53             //$this->shipStorage = new PdoShipStorage($this->getPDO());
54             $this->shipStorage = new
   JsonFileShipStorage(__DIR__.'/../../resources/ships.json');
55         }
↕ // ... lines 56 - 57
58     }
↕ // ... lines 59 - 70
71 }
```

From this directory I'm going up a couple of levels, into `resources` and pointing at this `ships.json` file which holds all of our ship info:

resources/ships.json

```
↕ // ... line 1
2 [
3     {
4         "id": "1",
5         "name": "Jedi Starfighter",
6         "weapon_power": "5",
7         "jedi_factor": "15",
8         "strength": "30",
9         "team": "rebel"
10    },
↕ // ... lines 11 - 26
27    {
28        "id": "4",
29        "name": "RZ-1 A-wing interceptor",
30        "weapon_power": "4",
31        "jedi_factor": "4",
32        "strength": "50",
33        "team": "empire"
34    }
35 ]
```

Back to the browser and refresh. Ok no success yet, but as they say, try try again. Before we do that, let's check out this error:

> *"Argument 1 passed to `ShipLoader::__construct()` must be an instance of `PdoShipStorage`, instance of `JsonFileShipStorage` given."*

What's happening here is that in `ShipLoader` we have this type-hint which says that we only accept `PdoShipStorage` and our Json file is not an instance of that:

lib/Service/ShipLoader.php

```
↕ // ... lines 1 - 2
3 class ShipLoader
4 {
↕ // ... lines 5 - 6
7     public function __construct(PdoShipStorage $shipStorage)
8     {
↕ // ... line 9
10    }
↕ // ... lines 11 - 58
59 }
↕ // ... lines 60 - 61
```

The easiest way to fix this is to say `extends PdoShipStorage` in `JsonFileShipStorage`:

lib/Service/JsonFileShipStorage.php

```
↕ // ... lines 1 - 2
3 class JsonFileShipStorage extends PdoShipStorage
↕ // ... lines 4 - 32
```

This makes the json file an instance of `PdoShipStorage`. Try refreshing that again. Perfect, our site is working.

But having to put that extends in our JSON file kinda sucks, when we do this we're overriding every single method and getting some extra stuff that we aren't going to use.

Creating a "Ship storage" contract

Instead, you should be thinking, "This is a good spot for Abstract Ship Storage!" And well, I agree so let's create that. Inside the `Service` directory add a new PHP Class called `AbstractShipStorage`. The two methods that this is going to need to have are: `fetchSingleShipData()` and `fetchAllShipsData()` so I'll copy both of those and paste them over to our new class.

Of course we don't have any body in these methods, so remove that. Now, set this as an `abstract` class. Make both of the functions `abstract` as well:

lib/Service/AbstractShipStorage.php

```
↕ // ... lines 1 - 2
3 abstract class AbstractShipStorage
4 {
5     abstract public function fetchAllShipsData();
6
7     abstract public function fetchSingleShipData($id);
8 }
```

Cool!

Now, `JsonFileShipStorage` can `extend AbstractShipStorage`:

lib/Service/JsonFileShipStorage.php

```
↕ // ... lines 1 - 2
3 class JsonFileShipStorage extends AbstractShipStorage
↕ // ... lines 4 - 32
```

And the same thing for `PdoShipStorage`:

lib/Service/PdoShipStorage.php

```
↕ // ... lines 1 - 2
3 class PdoShipStorage extends AbstractShipStorage
↕ // ... lines 4 - 33
```

With this setup we know that if we have a `AbstractShipStorage` it will definitely have both of those methods so we can go into the `ShipLoader` and change this type hint to `AbstractShipStorage` because we don't care which of the two storage classes are actually passed:

lib/Service/ShipLoader.php

```
↕ // ... lines 1 - 2
3 class ShipLoader
4 {
↕ // ... lines 5 - 6
7     public function __construct(AbstractShipStorage $shipStorage)
↕ // ... lines 8 - 58
59 }
↕ // ... lines 60 - 61
```

To be very well behaved developers, we'll go into our `Container` and above `getShipStorage()` change the type hint to `AbstractShipStorage`. Not a requirement, but it is a good idea.

Go back to the browser and refresh... oh, class `AbstractShipStorage` not found because we forgot to require it in our `bootstrap` file. We will eventually fix the need to have all of these require statements:

bootstrap.php

```
↕ // ... lines 1 - 14
15 require_once __DIR__.'/lib/Service/AbstractShipStorage.php';
↕ // ... lines 16 - 20
```

Refresh again and now it works perfectly.

We created an `AbstractShipStorage` because it allows us to make our `ShipLoader` more generic. It now doesn't care which one is passed, as long as it extends `AbstractShipStorage`.

But there's an even better way to handle this... interfaces!

Chapter 10: Interfaces

Notice, `AbstractShipStorage` unlike `AbstractShip`, doesn't actually have any logic in it:

lib/Service/AbstractShipStorage.php

```
↕ // ... lines 1 - 2
3 abstract class AbstractShipStorage
4 {
5     abstract public function fetchAllShipsData();
6
7     abstract public function fetchSingleShipData($id);
8 }
```

All it does is have a contract that guarantees anything that extends this has these two functions. It turns out that when you have an abstract class like this that only contains abstract functions and no real code, well it's the perfect opportunity to use an Interface.

An interface works just like an abstract class and here's how it looks. To start, we need to rename our class to `ShipStorageInterface` since this more closely matches what it is. And instead of `abstract class` it's now labeled as an `interface`:

lib/Service/ShipStorageInterface.php

```
↕ // ... lines 1 - 2
3 interface ShipStorageInterface
↕ // ... lines 4 - 9
```

Get it?

As soon as you do that you no longer need `abstract` in front of all the functions, but these work the same:

lib/Service/ShipStorageInterface.php

```
↕ // ... lines 1 - 2
3 interface ShipStorageInterface
4 {
5     public function fetchAllShipsData();
6
7     public function fetchSingleShipData($id);
8 }
```

On the `AbstractShipStorage` file in the tree, go to "Refactor" and click to "Rename" our file to `ShipStorageInterface`. I really like the consistency. And of course update our require line for this file in `bootstrap.php`:

bootstrap.php

```
↕ // ... lines 1 - 14
15 require_once __DIR__.'/lib/Service/ShipStorageInterface.php';
↕ // ... lines 16 - 20
```

Implement an Interface

Stepping back and looking at `ShipStorageInterface`. I want you to think of this as acting just like an abstract class with two functions that need to be filled in. An important difference is that you don't extend interfaces. Instead, we'll use a new keyword called `implements` and our updated class name `ShipStorageInterface`:

lib/Service/JsonFileShipStorage.php

```
↕ // ... lines 1 - 2
3 class JsonFileShipStorage implements ShipStorageInterface
↕ // ... lines 4 - 32
```

This new line says that the `JsonFileShipStorage` must include the functions inside of `ShipStorageInterface`.

If I deleted `fetchAllShipsData()` you can see that immediately PhpStorm is telling me:

> *"Hey buddy, you need to implement `fetchAllShipsData()`."*

So I'll comply and undelete that.

Update `PdoShipStorage` to `implement ShipStorageInterface`:

lib/Service/PdoShipStorage.php

```
↕ // ... lines 1 - 2
3 class PdoShipStorageInterface implements ShipStorageInterface
↕ // ... lines 4 - 33
```

Time to head over to `ShipLoader` and change the `AbstractShipStorage` type hint to `ShipStorageInterface` which is our way of saying that we don't care what class is passed here as long as it has the two methods that are in `ShipStorageInterface`:

lib/Service/ShipLoader.php

```
// ... lines 1 - 2
class ShipLoader
{
// ... lines 5 - 6
    public function __construct(ShipStorageInterface $shipStorage)
// ... lines 8 - 58
}
// ... lines 60 - 61
```

That's the only thing we care about. Well, that and getting to dinner on time.

Over in the `Container` we can also update the `@return` statement. It doesn't affect anything really, but it's a good practice to keep it updated. Back to the browser and refresh! Everything still works perfectly.

Interfaces are just like abstract classes that don't have any functionality, they only contain abstract functions. If you try to add a real function inside of an interface you can see that PhpStorm highlights it with the message:

> *"Interface method can't have body."*

And it will freak out when we refresh.

What's so Great about an Interface?

The purpose of an interface is to allow you to make your code very generic since you're not requiring a concrete class just an interface. Why do interfaces exist? Sheesh you ask a lot of questions! Well, the answer is that in PHP you can only extend one base class but you can implement many interfaces. I'm not going to go into detail on `ArrayAccess` interface which comes from the core of PHP but this is what it looks like to implement multiple interfaces. Allowing multiple interfaces makes them a bit more flexible than abstract classes.

Another cool thing about interfaces and abstract classes is that they become directions on what all ship storage objects must look like. So if someone in the future needed to create a new ship storage object that loaded things from say XML, all they would need to do is created a class that implements this interface and boom you're being told exactly what methods that XML ship storage class has to have.

Interfaces Document what you need to do

This is also our opportunity to add really good documentation on these. We can label this one as an integer that should return an array of data. You could even go further and say "Returns an array of ship arrays, with keys id, name, weaponPower, defense.":

lib/Service/ShipStorageInterface.php

```
// ... lines 1 - 2
interface ShipStorageInterface
{
    /**
     * Returns an array of ship arrays, each with the following keys:
// ... lines 7 - 11
     *
     * @return array
     */
    public function fetchAllShipsData();

    /**
     * Returns the single ship array for this id (see fetchAllShipsData)
     *
     * @param integer $id
     * @return array
     */
    public function fetchSingleShipData($id);
}
```

Adding as many details as possible here is good, that way if someone implements this interface later they'll know exactly what to put in their classes.

Interfaces in third-party Libraries

One last note about interfaces, they are a bit more advanced. It's not that they are difficult, but in your code you may not find many reasons to create these. How often is it that you need to make a class like `ShipLoader` and make it so flexible to work with a PDO ship storage or a Json file ship storage? In most apps you know which one way you are loading data. So it's actually ok to hardcode the implementation here with a concrete class like `PdoShipStorage`.

If you're creating a reusable library that you are going to share with the world then you will need a lot of flexibility and interfaces would be a good thing to use.

You may not create very many interfaces, but there is a very good chance that you will use a lot of them. For example, you might want to use a third party library in your project and their documentation will say:

> *“"If you want to create a custom ship storage object, then you will need to implement this interface that comes with the library."”*

So you will create your own custom class, implement the library's interface which then tells you which methods to fill in.

Understanding interfaces is really important because you will probably be implementing a lot of them.

Alright, that's it and I hope you find abstract classes, interfaces and inheritance as cool as I do!

See ya next time!

With <3 from SymfonyCasts