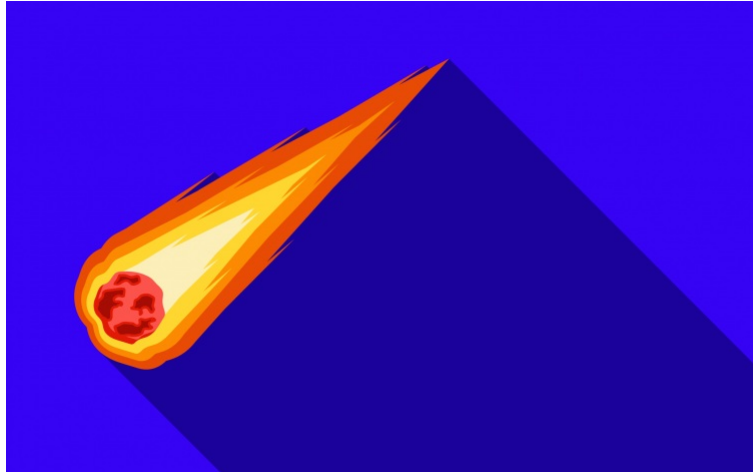# OOP (course 4): Static methods, Namespaces, Exceptions & Traits! Woh!

With <3 from SymfonyCasts

Chapter 1: The Wonder of Class Constants

Hey friends! I'm so glad you're here for part *4* of "Baking Delicious Chocolate Chip Cookies". Wait, they're telling me that's not right. Oh, ok, I'm so glad you're here for part *4* of our Object Oriented Programming series!

After the first 3 parts, you guys are already dangerous, so I'm impressed you're still showing up and aren't off coding something cool. You made the right choice: in this course we're going to really have fun with some of the coolest parts of OO, showing off features that we haven't mentioned yet. This is packed with the *final* pieces that will let you recognize all the different OO things that you see in other people's code. There's lots to get through, so let's go!

Get the Starting Code!

If you're serious about getting *really* good at this stuff, code along with me. To do that, download the source code from this page, unzip it, and move into the start directory. When you do that, you'll have the same code that I have here. Open up the README file and follow the instructions inside to get things setup.

When that's done, open your favorite terminal application, move into the directory, and - like we've done in the previous courses - start the built in php web server by running:

php -S localhost:8000

This is a great server to use for development. Then, in our browser, we can go to http://localhost:8000. Here is our beautiful Battles app!

New Feature! Battle Types

People have been *clamoring* for a new feature: a way to battle that *forces* Jedi powers to be used or completely avoided. Let's add this - it'll show off a new cool thing: class constants.

Open index.php and scroll down. Right *after* the ship select boxes, but *before* the submit button, I'll paste some HTML for a *new* select box:

138 lines index.php

```
... lines 1 - 29
<html>
... lines 31 - 55
<body>
<div class="container">
... lines 58 - 92
<div class="battle-box center-block border">
<div>
<form method="POST" action="/battle.php">
... lines 96 - 119
<div class="text-center">
<label for="battle_type">Battle Type</label>
<select name="battle_type" id="battle_type" class="form-control drp-dwn-width center-block">
<option value="normal">Normal</option>
<option value="no_jedi">No Jedi Powers</option>
<option value="only_jedi">Only Jedi Powers</option>
</select>
</div>
<br/>
<button class="btn btn-md btn-danger center-block" type="submit">Engage</button>
</form>
</div>
</div>
</div>
</body>
</html>
```

Let's refresh and see what it looks like. Ok, it's a new drop-down called "Battle Type" with option for "Normal", "No Jedi Powers" and "Only Jedi Powers". If you look at the code, this is a single select field that has a name of battle_type.

Here's the idea: each *type* will cause the BattleManager to battle these two ships in *slightly* different ways. Let's hook this up as *simply* as possible.

Since the field is named battle_type, open battle.php - the file that handles the submit. Right before calling the battle() method, create a new variable called $battleType set to $_POST['battle_type']. Then, pass $battleType as a new *fifth* argument to the battle() method:

110 lines battle.php

```
... lines 1 - 33
$battleType = $_POST['battle_type'];
$battleResult = $battleManager->battle($ship1, $ship1Quantity, $ship2, $ship2Quantity, $battleType);
... lines 36 - 110
```

Hooking up the Logic: No Magic Yet

Let's add that! Open BattleManager and find battle(). Give this a new fifth argument: $battleType:

76 lines lib/Service/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
... lines 5 - 9
public function battle(AbstractShip $ship1, $ship1Quantity, AbstractShip $ship2, $ship2Quantity, $battleType)
{
... lines 12 - 66
}
... lines 68 - 74
}
```

Great! *We* know that this will be one of three special strings, either normal, no_jedi or only_jedi. We can use those to change the behavior.

First, the two blocks near the top should *only* be run if Jedi powers are being used. Add to the if statement: if $battleType != 'no_jedi', then we can run this. Copy that and add it to the second block:

```php
... lines 1 - 2
class BattleManager
{
... lines 5 - 9
public function battle(AbstractShip $ship1, $ship1Quantity, AbstractShip $ship2, $ship2Quantity, $battleType)
{
... lines 12 - 17
while ($ship1Health > 0 && $ship2Health > 0) {
// first, see if we have a rare Jedi hero event!
if ($battleType != 'no_jedi' && $this->didJediDestroyShipUsingTheForce($ship1)) {
... lines 21 - 24
}
if ($battleType != 'no_jedi' && $this->didJediDestroyShipUsingTheForce($ship2)) {
... lines 27 - 30
}
... lines 32 - 44
}
... lines 46 - 66
}
... lines 68 - 74
}
```

Perfect! If the battle type is normal or only_jedi, these blocks will execute.

Next, the last two lines are when the two ships battle each other normally. If we're on only_jedi mode, this shouldn't happen. Surround them with an if statement: if ($battleType != 'only_jedi') then run these lines:

```php
... lines 1 - 2
class BattleManager
{
... lines 5 - 9
public function battle(AbstractShip $ship1, $ship1Quantity, AbstractShip $ship2, $ship2Quantity, $battleType)
{
... lines 12 - 17
while ($ship1Health > 0 && $ship2Health > 0) {
... lines 19 - 32
// now battle them normally
if ($battleType != 'only_jedi') {
$ship1Health = $ship1Health - ($ship2->getWeaponPower() * $ship2Quantity);
$ship2Health = $ship2Health - ($ship1->getWeaponPower() * $ship1Quantity);
}
... lines 38 - 44
}
... lines 46 - 66
}
... lines 68 - 74
}
```

Awesome! Now, there's just *one* little last detail: if two ships are fighting in only_jedi mode, and both have *zero* Jedi powers, they'll get caught in this loop and fight forever! To prevent that, above the while, add a new $i = 0 variable:

```php
... lines 1 - 2
class BattleManager
{
... lines 5 - 9
public function battle(AbstractShip $ship1, $ship1Quantity, AbstractShip $ship2, $ship2Quantity, $battleType)
{
... lines 12 - 16
$i = 0;
while ($ship1Health > 0 && $ship2Health > 0) {
... lines 19 - 44
}
... lines 46 - 66
}
... lines 68 - 74
}
```

Then, at the bottom, if $i = 100, we're probably stuck in a loop. Just set $ship1Health = 0; and $ship2Health = 0 and increment $i below that:

```
... lines 1 - 2
class BattleManager
{
... lines 5 - 9
public function battle(AbstractShip $ship1, $ship1Quantity, AbstractShip $ship2, $ship2Quantity, $battleType)
{
... lines 12 - 16
$i = 0;
while ($ship1Health > 0 && $ship2Health > 0) {
... lines 19 - 38
// prevent 2 non-jedi ships from fighting forever in only_jedi mode
if ($i == 100) {
$ship1Health = 0;
$ship2Health = 0;
}
$i++;
}
... lines 46 - 66
}
... lines 68 - 74
}
```

Done!

Give it a try!. Select one Jedi Starfighter, one CloakShape fighter, and choose "Only Jedi Powers". Hit engage and ... the Jedi Starfighter used its Jedi powers for a stunning victory! If we refresh, one of the ships will use its Jedi powers *every* single time.

Magic Strings Make Kittens Cry

Feature complete! And it was easy. So... what's the problem? Look at these strings: normal, no_jedi and only_jedi:

138 lines index.php

```
... lines 1 - 29
<html>
... lines 31 - 55
<body>
<div class="container">
... lines 58 - 92
<div class="battle-box center-block border">
<div>
<form method="POST" action="/battle.php">
... lines 96 - 119
<div class="text-center">
... line 121
<select name="battle_type" id="battle_type" class="form-control drp-dwn-width center-block">
<option value="normal">Normal</option>
<option value="no_jedi">No Jedi Powers</option>
<option value="only_jedi">Only Jedi Powers</option>
</select>
</div>
... lines 128 - 131
</form>
</div>
</div>
</div>
</body>
</html>
```

They're kind of magic. I mean, we chose them randomly and if you misspell one somewhere, you won't get an error, but things won't work right.

To make things worse, in BattleManager, when you see these strings, it's not clear what *other* strings might be possible. Are there other battle types we're forgetting to handle? And if we wanted to add or remove a battle type, what other files would we need to change? It's really common to have "magic strings" like these, but they can become hard to keep track of: you end up referencing these exact little strings in many places.

Class Constants to the Rescue

Of course, object-oriented code has an answer! It's called "class constants", and it works like this. Inside any class, you can use a special keyword called const followed by a word - which is usually in all uppercase - like TYPE_NORMAL and equals a value - normal. Repeat this for const TYPE_NO_JEDI = 'no_jedi' and const TYPE_ONLY_JEDI = 'only_jedi':

80 lines lib/Service/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
const TYPE_NORMAL = 'normal';
const TYPE_NO_JEDI = 'no_jedi';
const TYPE_ONLY_JEDI = 'only_jedi';
... lines 8 - 78
}
```

Constants are like variables, except they can never be changed. You can call the constants anything - by adding TYPE_ before each one, it helps me remember what these are used for - battle types. You can also add these to *any* class. I choice BattleManager because these types are used here.

Using Class Constants

As soon as you do this, you can replace the random string with BattleManager::TYPE_NO_JEDI. Below that, use BattleManager::TYPE_ONLY_JEDI:

80 lines lib/Service/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
    ... lines 5 - 13
    public function battle(AbstractShip $ship1, $ship1Quantity, AbstractShip $ship2, $ship2Quantity, $battleType)
    {
        ... lines 16 - 21
        while ($ship1Health > 0 && $ship2Health > 0) {
            ... line 23
            if ($battleType != BattleManager::TYPE_NO_JEDI && $this->didJediDestroyShipUsingTheForce($ship1)) {
                ... lines 25 - 28
            }
            if ($battleType != BattleManager::TYPE_NO_JEDI && $this->didJediDestroyShipUsingTheForce($ship2)) {
                ... lines 31 - 34
            }
            ... lines 36 - 37
            if ($battleType != BattleManager::TYPE_ONLY_JEDI) {
                ... lines 39 - 40
            }
            ... lines 42 - 48
        }
        ... lines 50 - 70
    }
    ... lines 72 - 78
}
```

That will work the *exact* same way as before. In index.php, do the same thing: <?php echo BattleManager::TYPE_NORMAL. Copy that and replace it with TYPE_NO_JEDI and TYPE_ONLY_JEDI:

138 lines index.php

```
... lines 1 - 29
<html>
    ... lines 31 - 55
    <body>
    <div class="container">
        ... lines 58 - 92
        <div class="battle-box center-block border">
        <div>
        <form method="POST" action="/battle.php">
            ... lines 96 - 119
            <div class="text-center">
                ... line 121
                <select name="battle_type" id="battle_type" class="form-control drp-dwn-width center-block">
                <option value="<?php echo BattleManager::TYPE_NORMAL ?>">Normal</option>
                <option value="<?php echo BattleManager::TYPE_NO_JEDI ?>">No Jedi Powers</option>
                <option value="<?php echo BattleManager::TYPE_ONLY_JEDI ?>">Only Jedi Powers</option>
                </select>
            </div>
            ... lines 128 - 131
        </form>
        </div>
        </div>
    </div>
    </body>
</html>
```

To prove it still works, refresh this page. Everything's still happy!

In a sense, nothing changed! But now, these magic strings have a *single* home: at the top of BattleManager. If we ever needed to *change* these strings, we can do it in just *once* place.

This *also* gives these strings some context - these are obviously related to BattleManager, and we can probably look here to see how they're used. We can also *document* what they mean by adding some details above each type:

83 lines lib/Service/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
    // normal battle mode
    const TYPE_NORMAL = 'normal';
    // don't allow jedi powers
    const TYPE_NO_JEDI = 'no_jedi';
    // you can *only* win with jedi powers
    const TYPE_ONLY_JEDI = 'only_jedi';
    ... lines 11 - 81
}
```

Now, check this out. When some *other* developer looks inside index.php, instead of seeing some magic, meaningless strings like before, they'll see these constants and think:

> Oh, BattleManager::TYPE_NORMAL. Let me go look in that class to see what this means. Oh hey, there's even some documentation!

So anytime you have a special string or other value that has some special meaning but will never change, make it a constant and stay happy.

Chapter 2: Static Methods

A really important thing just happened: for the first time *ever*, we referred to something on our class by using its *class name*. To use the constant, we said BattleManager::TYPE_NO_JEDI:

80 lines lib/Service/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
... lines 5 - 13
    public function battle(AbstractShip $ship1, $ship1Quantity, AbstractShip $ship2, $ship2Quantity, $battleType)
    {
... lines 16 - 21
        while ($ship1Health > 0 && $ship2Health > 0) {
            // first, see if we have a rare Jedi hero event!
            if ($battleType != BattleManager::TYPE_NO_JEDI && $this->didJediDestroyShipUsingTheForce($ship1)) {
... lines 25 - 28
            }
... lines 30 - 48
        }
... lines 50 - 70
    }
... lines 72 - 78
}
```

That makes sense, but notice: it's *completely* different than how we've referred to class properties and methods so far. Normally, we create a new object by saying new BattleManager():

72 lines lib/Service/Container.php

```
... lines 1 - 2
class Container
{
... lines 5 - 62
    public function getBattleManager()
    {
        if ($this->battleManager === null) {
            $this->battleManager = new BattleManager();
        }

        return $this->battleManager;
    }
}
```

For us, this lives inside the Container. But here's the important part: to reference a method or property, we use the *object* by saying $battleManager-> followed by the method name:

110 lines battle.php

```
... lines 1 - 34
$battleResult = $battleManager->battle($ship1, $ship1Quantity, $ship2, $ship2Quantity, $battleType);
... lines 36 - 110
```

For constants, it's totally different. We don't ever need to instantiate an object. Instead, at any point, you can just say the class name ::TYPE_NO_JEDI:

80 lines lib/Service/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
... lines 5 - 13
    public function battle(AbstractShip $ship1, $ship1Quantity, AbstractShip $ship2, $ship2Quantity, $battleType)
    {
... lines 16 - 21
        while ($ship1Health > 0 && $ship2Health > 0) {
            // first, see if we have a rare Jedi hero event!
            if ($battleType != BattleManager::TYPE_NO_JEDI && $this->didJediDestroyShipUsingTheForce($ship1)) {
... lines 25 - 28
            }
... lines 30 - 48
        }
... lines 50 - 70
    }
... lines 72 - 78
}
```

So sometimes, we need to create an object and reference that *object*. But other times, we don't need an object: we just use the class name. What's going on?

Static versus Non Static

Here's the deal: constants are *static*, and so far, all of our properties and methods are *non-static*.

You see, whenever you add something to a class - like a property or a method - you can choose to attach it to an individual instance of the object or to the class itself. When you choose to attach something to a class, it's said to be "static".

Let's look at a real example. In AbstractShip, the properties id, name, weaponPower and strength are *not* static:

123 lines lib/Model/AbstractShip.php

```
... lines 1 - 2
abstract class AbstractShip
{
private $id;

private $name;

private $weaponPower = 0;

private $strength = 0;
... lines 12 - 121
}
```

That means that if you have two Ship objects, each has a different id, name, weaponPower and strength. If you change the name in one Ship it does *not* affect any other ship objects.

But, if we were to change these properties to static - which *is* something you can do - then suddenly the name property would be global to *all* ships, meaning two ship objects could *not* have different names. This would be the *one* name for all AbstractShip.

Remember - a class is like a blueprint for a ship, and an object is like a real, physical ship. Since each real ship has a different name, it makes sense to make the $name property *non-static*. This attaches the name to each individual object.

But other times, it may make sense to attach a property to the *blueprint* itself, meaning to the class. For example, suppose that the very design of the ships guarantees that each should have a minimum strength of 100. Since that is a property of ships in general, we might add a new private static property called $minimumStrength and use that to prevent individual ships from setting their specific $strength lower than this.

Class Constants are Static

So, with properties or methods, you can choose static or non-static. But constants, well, they're static by their very nature. And that makes sense: the TYPE constants in BattleManager are global to the BattleMangaer class in general - it wouldn't make sense for them to be different for different objects.

When you reference something statically, you always reference it by saying the class name, ::, and then whatever you're referencing.

The Special Self Keyword

Before we try an example, there's another special property of static things. Notice that we're inside BattleManager and we're referencing the BattleManager class. If you want to, you can change this to, self::TYPE_NO_JEDI:

83 lines lib/Service/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
... lines 5 - 16
public function battle(AbstractShip $ship1, $ship1Quantity, AbstractShip $ship2, $ship2Quantity, $battleType)
{
... lines 19 - 24
while ($ship1Health > 0 && $ship2Health > 0) {
// first, see if we have a rare Jedi hero event!
if ($battleType != self::TYPE_NO_JEDI && $this->didJediDestroyShipUsingTheForce($ship1)) {
... lines 28 - 31
}
if ($battleType != self::TYPE_NO_JEDI && $this->didJediDestroyShipUsingTheForce($ship2)) {
... lines 34 - 37
}
// now battle them normally
if ($battleType != self::TYPE_ONLY_JEDI) {
... lines 42 - 43
}
... lines 45 - 51
}
... lines 53 - 73
}
... lines 75 - 81
}
```

In the same way that $this refers to the current object, self refers to the *class* that we're inside of. So this didn't change our behavior: it's just a nice shortcut.

Now, let's see a real-life static method in action.

Chapter 3: Static or Non-Static Methods?

In index.php, the three battle types are hard coded right in the HTML:

138 lines index.php

```
... lines 1 - 29
<html>
... lines 31 - 55
<body>
<div class="container">
... lines 58 - 92
<div class="battle-box center-block border">
<div>
<form method="POST" action="/battle.php">
... lines 96 - 119
<div class="text-center">
<label for="battle_type">Battle Type</label>
<select name="battle_type" id="battle_type" class="form-control drp-dwn-width center-block">
<option value="<?php echo BattleManager::TYPE_NORMAL ?>">Normal</option>
<option value="<?php echo BattleManager::TYPE_NO_JEDI ?>">No Jedi Powers</option>
<option value="<?php echo BattleManager::TYPE_ONLY_JEDI ?>">Only Jedi Powers</option>
</select>
</div>
... lines 128 - 131
</form>
</div>
</div>
</div>
</body>
</html>
```

So what happens if we decide to add a *fourth* battle type to BattleManager. No problem: add a new constant, then update the battle() method logic for whatever cool thing this new type does.

But surprise! If we forget to *also* add the new type to index.php, then nobody's going to be able to use it. Really, I'd prefer BattleManager to be *completely* in charge of the battle types so that it's the *only* file I need to touch when something changes.

Using a Normal, Non-Static Method

To do that, create a new function in BattleManager that will return all of the types and their descriptions: call it public function getAllBattleTypesWithDescription():

92 lines lib/Service/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
... lines 5 - 75
public function getAllBattleTypesWithDescriptions()
{
... lines 78 - 82
}
... lines 84 - 90
}
```

Here, return an array with the type as the key and the description that should be used in the drop-down as the value:

92 lines lib/Service/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
... lines 5 - 75
public function getAllBattleTypesWithDescriptions()
{
return array(
self::TYPE_NORMAL => 'Normal',
self::TYPE_NO_JEDI => 'No Jedi Powers',
self::TYPE_ONLY_JEDI => 'Only Jedi Powers'
);
}
... lines 84 - 90
}
```

Awesome! Next, if we call this method in index.php, we can remove the hardcoded values there. Of course, this method is *non-static*. That means that we need to call this method on a BattleManager *object*. Create a new one by saying $battleManager = $container->getBattleManager():

141 lines index.php

```
... lines 1 - 11
$battleManager = $container->getBattleManager();
... lines 13 - 141
```

Now add $battleTypes = $battleManager->getAllBattleTypesWithDescription():

141 lines index.php

```
... lines 1 - 11
$battleManager = $container->getBattleManager();
$battleTypes = $battleManager->getAllBattleTypesWithDescriptions();
... lines 14 - 141
```

Finally, scroll down. In place of the hardcoded values, foreach over $battleTypes as $battleType => $typeText. End the foreach and make the option dynamic by printing $battleType and <?php echo $typeText; ?>:

141 lines index.php

```
... lines 1 - 32
<html>
... lines 34 - 58
<body>
<div class="container">
... lines 61 - 95
<div class="battle-box center-block border">
<div>
<form method="POST" action="/battle.php">
... lines 99 - 122
<div class="text-center">
<label for="battle_type">Battle Type</label>
<select name="battle_type" id="battle_type" class="form-control drp-dwn-width center-block">
<?php foreach ($battleTypes as $battleType => $typeText): ?>
<option value="<?php echo $battleType ?>"><?php echo $typeText; ?></option>
<?php endforeach; ?>
</select>
</div>
... lines 131 - 134
</form>
</div>
</div>
</div>
</body>
</html>
```

Ok! Give it a try! Click the "Battle Again" link. And yes! The drop-down has the same three values as before.

Why not make the Method Static?

Here's where things get interesting! We made getAllBattleTypesWithDescription() *non-static*. Could we make it static instead?

To know, ask yourself these two questions:

1. Does it make sense - philosphically - for the getAllBattleTypesWithDescription() method to be attached to the *class* instead an object? I would say yes: the battle types and descriptions will *not* be different for different BattleManager objects; these are global to the class.

2. Does the method need the $this variable? If you need to reference non-static properties using $this, then the method *must* be non-static. But we're not using $this.

So let's make this method static by saying public static function:

92 lines lib/Service/BattleManager.php

```
... lines 1 - 2
class BattleManager
{
... lines 5 - 75
public static function getAllBattleTypesWithDescriptions()
{
... lines 78 - 82
}
... lines 84 - 90
}
```

The only thing that changes now is *how* we call our method. First, we don't need a BattleManager object at all. Instead, just say BattleManager::getAllBattleTypesWithDescription():

140 lines index.php

```
... lines 1 - 11
$battleTypes = BattleManager::getAllBattleTypesWithDescriptions();
... lines 13 - 140
```

Ok, try it out! It works!

When to use Static versus Non-Static

So look, this static versus non-static stuff can be tough. And in a lot of other tutorials, you'll see this taught in reverse: they'll show you static stuff first, because it's a little easier. *Then* they'll teach non-static properties and methods.

But guess what: that's not how *good* programmers code in the real world: they make most things *non-static*. And to start, I want you guys to also make everything not static. Then, as you get more comfortable, you will start to see different situations where it's okay to make some things static. It's actually much easier to change things from non-static to static than the other way around. And when you make things non-static, it forces you to build better code. And isn't that why we're here?

Chapter 4: Namespaces make Class Names Longer

We all know that the name of this class is BattleManager. When we want to use it, we reference BattleManager. No matter what we do - static or non-static - if we want to work with this class we call it by its name, BattleManager. Simple.

Why am I pointing out the painfully obvious? Because we're about to make this class name *longer*, but maybe not how you'd expect. We're going to use a namespace.

Let's see some Namespaces

At first, *why* namespaces exist might not be obvious, so hold onto that question. Let's see how they work first.

Above any class, you can - if you want to - add a namespace keyword followed by some string. Like, Battle or something more complicated like Battle\HiGuys\NiceNameSpace. A namespace is a string, and you can give it different parts by separating each with a backslash \ - that's the slash that feels a little wrong when you type it - it's usually an escape character.

To keep things simple, just set the namespace to Battle for now:

94 lines lib/Service/BattleManager.php

```
... lines 1 - 2
namespace Battle;
class BattleManager
... lines 6 - 94
```

As soon as we did that, we actually *changed* the name of this class: it is no longer called BattleManager. In fact, you can see that PhpStorm now highlights our code with an "Undefined class BattleManager" error. Thanks to the namespace, the class is *now* called Battle\BattleManager:

140 lines index.php

```
... lines 1 - 11
$battleTypes = Battle\BattleManager::getAllBattleTypesWithDescriptions();
... lines 13 - 140
```

Refresh to prove it. Great!

So... that's really it! When you add a namespace above a class, the full class name becomes that namespace, a \, and then class name. *Every* place we reference this class name will now need to change - like inside of Container. We'll do that in a few minutes - we've got a few other things to do first.

So Why do Namespaces Exist?

Now that you know how namespaces work, you're probably wondering, why do these even exist? How does this help me in my coding? Well, the short answer is... it doesn't help you. In fact, namespaces weren't *meant* to help you - they were meant to help external library developers. So I guess, if you're one of those it does help.

In a nut shell, as you go further into development, you'll start to use a lot of 3rd-party, libraries written by other people. That's cool because those libraries will give *us* new classes to help solve problems.

The reason that namespaces exist is to avoid collisions in those external libraries. Imagine we're using library A and library B, but that they *both* have a class called Battle. Without namespaces, we'd be lost in space: we wouldn't be able to use both libraries. But if each library has a unique namespace, we won't collide: they'll simply be called something like LibraryA\Battle and LibraryB\Battle.

This means that namespaces *do* help us, but only indirectly. When we're working with namespaces it just makes our class names longer.

The use Statement

There is *one* other thing that you need to know with namespace: it's the mystical use statement.

When you want to reference a class, it's perfectly valid to type out the *entire* long class name right where you need to use it. But in practice, you won't see this very often. Instead, people typically add a use statement at the top of the file that references the *full* class name: Battle\BattleManager:

143 lines index.php

```
... lines 1 - 2
use Battle\BattleManager;
... lines 4 - 143
```

As soon as you do, when you need to work with the class, you can once again write out *only* the short class name:

143 lines index.php

```
... lines 1 - 2
use Battle\BattleManager;
... lines 4 - 14
$battleTypes = BattleManager::getAllBattleTypesWithDescriptions();
... lines 16 - 143
```

And while you'll only have *one* namespace per file, you'll have as many use statements as you need.

To be clear, the use statement does *not* change how namespaces work: it's just a shortcut. When PHP executes this file, it sees class BattleManager and says:

> Huh, BattleManager? Let me check all of the use statements at the top of this file.

PHP then looks to see if any of the use statements *end* in the word BattleManager. If it finds one, it basically copies the long class name and pastes it below *right* before executing the file. What I just did manually is what PHP basically does at run-time.

So use statements are just this nice, extra feature. And technically, you could avoid using them and instead write-out full class names right where you need them.

Ok! We're going to do a lot more with namespaces. But first, we need to turn to a very related topic called *autoloading*.

Go Deeper!

If you still have questions about namespaces - check out our short course PHP Namespaces in 120 Seconds or just leave a comment.

Chapter 5: Autoloading Awesomeness

Have you ever heard of an autoloader? Even if you *have*, you might not know what they do or how they work.

What does an Autoloader Do?

Autoloaders change everything. In PHP, you can't reference a class or a function unless you - or someone - requires or includes that file first. That's why - in bootstrap.php - we have a require statement for *every* file:

20 lines bootstrap.php
```
... lines 1 - 8
require_once __DIR__.'/lib/Service/Container.php';
require_once __DIR__.'/lib/Model/AbstractShip.php';
require_once __DIR__.'/lib/Model/Ship.php';
require_once __DIR__.'/lib/Model/RebelShip.php';
require_once __DIR__.'/lib/Model/BrokenShip.php';
require_once __DIR__.'/lib/Service/BattleManager.php';
require_once __DIR__.'/lib/Service/ShipStorageInterface.php';
require_once __DIR__.'/lib/Service/PdoShipStorage.php';
require_once __DIR__.'/lib/Service/JsonFileShipStorage.php';
require_once __DIR__.'/lib/Service/ShipLoader.php';
require_once __DIR__.'/lib/Model/BattleResult.php';
```

Without these, we can't access the classes inside of them.

This is no bueno: it means that I have to remember to add another line here, whenever I create a new class. You know why else it's not good? Suppose I don't use all of these classes during some requests? Well, right now, I'm loading *every* class into memory, even if we never need to use them. This is actually slowing down my app!

Well, guess what: in modern PHP, you *never* see require or include statements. They're gone. How is that possible? The Answer is: *autoloaders*.

First, kill the BattleManager.php require statement:

19 lines bootstrap.php
```
... lines 1 - 12
require_once __DIR__.'/lib/Model/BrokenShip.php';
require_once __DIR__.'/lib/Service/ShipStorageInterface.php';
... lines 15 - 19
```

Not surprisingly, we get an error:

    Class Battle\BattleManager not found.

Adding your Autoloader

How do we fix this? The answer is by calling a very special function from the core of PHP called spl_autoload_register(). Pass this a single argument: a function with a $className argument. We'll use an anonymous function:

29 lines bootstrap.php
```
... lines 1 - 2
spl_autoload_register(function($className) {
... lines 4 - 10
});
... lines 12 - 29
```

Here's the deal: as soon as you call spl_autoload_register, right before PHP throws the dreaded "class not found" error like this, it will call *our* function and pass it the class name. Then, if we - somehow - can locate the file that contains this class and require it, PHP will continue on like normal with no error.

In fact, in modern PHP development, this is how *every single class* is loaded. In some cases, this little function is called *hundreds* of times on every request.

Making your Autoloader Work (a little)

Let's start coding our autoloader with some simple logic: if ($className == 'Battle\BattleManager'), then we know where that file lives. require __DIR__.'/lib/Service/BattleManager.php'. Then, add a return:

29 lines bootstrap.php
```
... lines 1 - 2
spl_autoload_register(function($className) {
if ($className == 'Battle\BattleManager') {
require __DIR__.'/lib/Service/BattleManager.php';
return;
}
// we don't support this class!
});
... lines 12 - 29
```

We're done!

For now, if the autoloader function is called for any other class, we'll do nothing. PHP will throw its normal "class not found" error.

With *just* that, refresh. Mind blown. We just got our app to work *without* manually requiring the BattleManager.php file. Of course, right now, this isn't much better than having a require statement. Actually, it's *more* work.

Creating a Smarter Autoloader

How could we make this function smarter? How could we make it automatically find *new* classes and files as we add them to the system?

Well I have an idea. BattleManager lives in the Service directory. What if we changed its namespace to match that? Or to get crazier, what if we gave *every* class a namespace that matches its directory?

If we did that, the autoload function could use the namespace to locate any file. The class - Service\BattleManager would live at Service/BattleManager.php. It's brilliant!

94 lines lib/Service/BattleManager.php

```
... lines 1 - 2
namespace Service;
class BattleManager
... lines 6 - 94
```

Now that we've changed the namespace to Service, we need to update any references to BattleManager - like in index.php. Change the use statement to Service.

143 lines index.php

```
... lines 1 - 2
use Service\BattleManager;
... lines 4 - 143
```

Yes!

Finally, in bootstrap.php, instead of manually checking for *just* this one class, say that the path is always equal to __DIR__/lib/ then str_replace() - we'll replace the back slash with a forward slash:

29 lines bootstrap.php

```
... lines 1 - 2
spl_autoload_register(function($className) {
$path = __DIR__.'/lib/'.str_replace('\\', '/', $className).'.php';
... lines 5 - 10
});
... lines 12 - 29
```

Notice I put *two* back-slashes. Since this is the escape character, if you only have one, it looks like you're escaping the next quote character. So, to get one slash, we need to use two - it's an ugly detail. Anyways, replace backslashes with a forward slash and pass that the class name. Finally, add the .php at the end.

Just in case, check to see if that file exists. If it does, require $path:

29 lines bootstrap.php

```
... lines 1 - 2
spl_autoload_register(function($className) {
$path = __DIR__.'/lib/'.str_replace('\\', '/', $className).'.php';
if (file_exists($path)) {
require $path;
}
// we don't support this class!
});
... lines 12 - 29
```

That's it. Go back, refresh, and... everything still works.

And now, we are incredibly dangerous. We can now get rid of every single require statement really easily. Let's do it!

Chapter 6: More Fun with use Statements

I hate needing all these require statements. But thanks to our autoloader, the *only* thing we need to do is give every class the namespace that matches its directory. This will be a little bit of work because we didn't do it up front - life is much easier when you use namespaces like this from the very beginning. But, we'll learn some other stuff along the way.

The AbstractShip class lives in the Model directory, so give it the namespace Model:

125 lines lib/Model/AbstractShip.php

```
... lines 1 - 2
namespace Model;

abstract class AbstractShip
... lines 6 - 125
```

Copy that and do the same thing in BattleResult, BrokenShip, RebelShip, Ship and FriendShip -- just kidding there's none of that in epic code battles:

57 lines lib/Model/BattleResult.php

```
... lines 1 - 2
namespace Model;

class BattleResult
... lines 6 - 57
```

22 lines lib/Model/BrokenShip.php

```
... lines 1 - 2
namespace Model;

class BrokenShip extends AbstractShip
... lines 6 - 22
```

38 lines lib/Model/RebelShip.php

```
... lines 1 - 2
namespace Model;

class RebelShip extends AbstractShip
... lines 6 - 38
```

45 lines lib/Model/Ship.php

```
... lines 1 - 2
namespace Model;

class Ship extends AbstractShip
... lines 6 - 45
```

Perfect. BattleManager already has the correct Service namespace:

94 lines lib/Service/BattleManager.php

```
... lines 1 - 2
namespace Service;

class BattleManager
... lines 6 - 94
```

In Container, paste that same one:

74 lines lib/Service/Container.php

```
... lines 1 - 2
namespace Service;

class Container
... lines 6 - 74
```

Repeat that in JsonFileShipStorage, PdoShipStorage, ShipLoader and ShipStorageInterface:

34 lines lib/Service/JsonFileShipStorage.php

```
... lines 1 - 2
namespace Service;

class JsonFileShipStorage implements ShipStorageInterface
... lines 6 - 34
```

35 lines lib/Service/PdoShipStorage.php

```
... lines 1 - 2
namespace Service;

class PdoShipStorage implements ShipStorageInterface
... lines 6 - 35
```

63 lines lib/Service/ShipLoader.php

```
... lines 1 - 2
namespace Service;

class ShipLoader
... lines 6 - 63
```

27 lines lib/Service/ShipStorageInterface.php

```
... lines 1 - 2
namespace Service;

interface ShipStorageInterface
... lines 6 - 27
```

These all live in the Service directory.

Missing use Statements = Common Error

Ok! Let's see what breaks! Go back and refresh. The first error we get is:

> Class Container not found in index.php

Ok, you're going to see *a lot* of class not found errors in your future. When you see them, read the error very closely: it always contains a hint. This says class Container is not found. Well, we don't have a class called Container: our class is called Service\Container. This tells me that in index.php on line 6, we're referencing the class name *without* the namespace. Sure enough, we have new Container:

143 lines index.php

```
... lines 1 - 6
$container = new Container($configuration);
... lines 8 - 143
```

To fix this, we *could* say Service\Container here *or* we can add a use statement for Service\Container. Let's do that:

145 lines index.php

```
... lines 1 - 3
use Service\Container;
... lines 5 - 8
$container = new Container($configuration);
... lines 10 - 145
```

And I can already see that we'll have the same problem down below with BrokenShip: PhpStorm is trying to warn me! Add a use Model\BrokenShip to take care of that:

145 lines index.php

```
... lines 1 - 4
use Model\BrokenShip;
... lines 6 - 13
$brokenShip = new BrokenShip('I am so broken');
... lines 15 - 145
```

We'll probably have the same problem in battle.php - so open that up. Yep, add use Service\Container:

112 lines battle.php

```
... line 1
use Service\Container;
... lines 3 - 5
$container = new Container($configuration);
... lines 7 - 112
```

Looking good!

Reading the Error Messages... Closely

Try it again! Ok:

> Class Service\RebelShip not found in ShipLoader.

Remember what I just said about reading the error messages closely? This one has a clue: it's looking for Service\RebelShip. But we don't have a class called Service\RebelShip - our class is called Model\RebelShip:

38 lines lib/Model/RebelShip.php

```
... lines 1 - 2
namespace Model;
class RebelShip extends AbstractShip
... lines 6 - 38
```

The problem exists where we're *referencing* this class - so in ShipLoader at line 43.

This is the most *common* mistake with namespaces: we have new RebelShip, but we *don't* have a use statement on top for this:

63 lines lib/Service/ShipLoader.php

```
... lines 1 - 2
namespace Service;
class ShipLoader
{
... lines 7 - 40
private function createShipFromData(array $shipData)
{
if ($shipData['team'] == 'rebel') {
$ship = new RebelShip($shipData['name']);
} else {
$ship = new Ship($shipData['name']);
... line 47
}
... lines 49 - 54
}
... lines 56 - 60
}
```

This is the same problem we just solved in index.php, but with a small difference. Unlike index.php and battle.php, this file lives in a namespace called Service. That causes PHP to assume that RebelShip *also* lives in that namespace -- you know like roommates.

Here's how it works: when PHP parses this file, it sees the RebelShip class on line 43. Next, it looks up at the top of the file to see if there are any use statements that end in RebelShip. Since there aren't, it assumes that RebelShip also lives in the Service namespace, so Service\RebelShip.

Think about it: this is *just* like directories on your filesystem. If you are inside of a directory called Service and you say ls RebelShip, it's going to look for RebelShip inside of the Service directory.

But in index.php - since this doesn't hold a class - we didn't give this file a namespace. If you forget a use statement for BrokenShip here, this is equivalent to saying ls BrokenShip from the *root* of your

file system, instead of from inside some directory.

In both cases the solution is the same: add the missing use statement: use Model\RebelShip:

67 lines lib/Service/ShipLoader.php

```
... lines 1 - 2
namespace Service;
use Model\RebelShip;
... lines 6 - 8
class ShipLoader
... lines 10 - 67
```

Now PhpStorm *stops* highlighting this as an error. Much better.

We have the same problem below for Ship: add use Model\Ship:

67 lines lib/Service/ShipLoader.php

```
... lines 1 - 2
namespace Service;
use Model\RebelShip;
use Model\Ship;
... lines 7 - 8
class ShipLoader
... lines 10 - 67
```

Finally, there's one more spot in the PHP documentation itself. Because we don't have a use statement in this file yet for AbstractShip, PhpStorm assumes that this class is Service\AbstractShip. To fix that, add use Model\AbstractShip:

67 lines lib/Service/ShipLoader.php

```
... lines 1 - 2
namespace Service;
use Model\RebelShip;
use Model\Ship;
use Model\AbstractShip;
class ShipLoader
{
... lines 11 - 17
/**
* @return AbstractShip[]
*/
public function getShips()
{
... lines 23 - 31
}
/**
* @param $id
* @return AbstractShip
*/
public function findOneById($id)
{
... lines 40 - 42
}
... lines 44 - 64
}
```

Now, everything looks happy!

The moral of the story is this: whenever you reference a class, don't forget to put a use statement for it. Now, there is *one* exception to this rule. If you reference a class that happens to be in the *same* namespace as the file you're in - like ShipStorageInterface - then you don't need a use statement:

67 lines lib/Service/ShipLoader.php

```
... lines 1 - 8
class ShipLoader
{
... lines 11 - 12
public function __construct(ShipStorageInterface $shipStorage)
{
... line 15
}
... lines 17 - 64
}
```

PHP correctly assumes that ShipStorageInterface lives in the Service namespace. But you don't get lucky like this *too* often.

I already know we need to fix *one* more spot in BattleManager. Add a use statement for Model\BattleResults and another for Model\AbstractShip:

97 lines lib/Service/BattleManager.php

```
... lines 1 - 2
namespace Service;
use Model\BattleResult;
use Model\AbstractShip;
class BattleManager
... lines 9 - 97
```

Phew! I promise, this is all a lot easier if you just use namespaces from the beginning! Let's refresh the page. Our app is back to life, and the require statements are gone!

Chapter 7: Namespaces and Core PHP Classes

Let's close all our tabs and open up Container. In the last course, we created *two* different ways to load Ship objects: one that reads a JSON file - JsonFileShipStorage and another that reads from a database - PdoShipStorage:

74 lines lib/Service/Container.php

```
... lines 1 - 4
class Container
{
... lines 7 - 51
public function getShipStorage()
{
if ($this->shipStorage === null) {
//$this->shipStorage = new PdoShipStorage($this->getPDO());
$this->shipStorage = new JsonFileShipStorage(__DIR__.'/../../resources/ships.json');
}
... lines 58 - 59
}
... lines 61 - 72
}
```

And you could switch back and forth between these without breaking anything, thanks to our cool ShipStorageInterface. Change it to use the PDO version and refresh.

Woh, new error:

> Class Service\PDO not found in Container.php on line 28.

Let's check that out:

74 lines lib/Service/Container.php

```
... lines 1 - 4
class Container
{
... lines 7 - 24
public function getPDO()
{
if ($this->pdo === null) {
$this->pdo = new PDO(
... lines 29 - 31
);
$this->pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}
... lines 36 - 37
}
... lines 39 - 72
}
```

use Statements for core PHP Classes?

Here, we see the *exact* same error as before: "Undefined Class PDO". So far, the answer to this has always been:

> Oh, I must have forgotten a use statement. I referenced a class, so I probably need to add a use statement for it.

But here's the kicker: PDO is a *core* PHP class that happens to *not* live in a namespace. In other words, it's like a file that lives at the root of you file system: not in any directory.

So when PHP sees PDO mentioned, it looks at the top of the class for a use statement that ends in PDO, it doesn't find one, and it assumes that PDO lives in the Service namespace. But in fact, PDO lives at the root namespace.

The fix is easy: add a \ at the front of PDO:

74 lines lib/Service/Container.php

```
... lines 1 - 4
class Container
{
... lines 7 - 24
public function getPDO()
{
if ($this->pdo === null) {
$this->pdo = new \PDO(
... lines 29 - 31
);
$this->pdo->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);
}
... lines 36 - 37
}
... lines 39 - 72
}
```

This makes sense: if you think of namespaces like a directory structure, This is like saying ls /PDO . It doesn't matter *what* directory, or namespace, we're in, adding the \ tells PHP that this class lives at the root namespace. Update the other places where we reference this class.

The Opening Slash is Portable

This is true for *all* core PHP classes: none of them live in namespaces. So, *always* include that beginning \. Now, technically, if you were inside of a file that did *not* have a namespace - like index.php - then you don't need the opening \. But it's *always* safe to say new \PDO: it'll work in all files, regardless of whether or not they have a namespace.

When Type-Hints Fail

If you refresh now, you'll see another error that's caused by this same problem. But this one is less clear:

> Argument 1 passed to PDOShipStorage::__construct() must be an instance of Service\PDO, instance of PDO given.

This should jump out at you: "Instance of Service\PDO". PHP thinks that argument 1 to PDOShipStorage should be this, *nonsense* class. There is no class Service\PDO!

Check out PDOShipStorage: the __construct() argument is type-hinted with PDO:

35 lines lib/Service/PdoShipStorage.php

```
... lines 1 - 4
class PdoShipStorage implements ShipStorageInterface
{
    ... lines 7 - 8
    public function __construct(PDO $pdo)
    {
        ... line 11
    }

    public function fetchAllShipsData()
    {
        ... lines 16 - 18
        return $statement->fetchAll(PDO::FETCH_ASSOC);
    }

    public function fetchSingleShipData($id)
    {
        ... lines 24 - 25
        $shipArray = $statement->fetch(PDO::FETCH_ASSOC);
        ... lines 27 - 32
    }
}
```

But of course, this *looks* like Service\PDO to PHP, and that causes problems. Add the \ there as well:

35 lines lib/Service/PdoShipStorage.php

```
... lines 1 - 4
class PdoShipStorage implements ShipStorageInterface
{
    ... lines 7 - 8
    public function __construct(\PDO $pdo)
    {
        ... line 11
    }
    ... lines 13 - 33
}
```

Phew! We spent time on these because these are the mistakes and errors that we *all* make when starting with namespaces. They're annoying, unless you can debug them quickly. If you're ever not sure about a "Class Not Found" error, the problem is almost always a missing use statement.

Update the other spots that reference PDO:

35 lines lib/Service/PdoShipStorage.php

```
... lines 1 - 4
class PdoShipStorage implements ShipStorageInterface
{
    ... lines 7 - 13
    public function fetchAllShipsData()
    {
        ... lines 16 - 18
        return $statement->fetchAll(\PDO::FETCH_ASSOC);
    }

    public function fetchSingleShipData($id)
    {
        ... lines 24 - 25
        $shipArray = $statement->fetch(\PDO::FETCH_ASSOC);
        ... lines 27 - 32
    }
}
```

Finally, refresh! Life is good. You just saw the ugliest parts of namespaces.

Chapter 8: Composer Autoloading

Ok, guys: confession time. This cool little autoloader idea where we make our class name match our file name and our namespace match our directory structure ... well, that was *not* my idea. In fact, this idea has been around in PHP for years, and every modern project follows it. That's nice for consistency and organization, but it's also nice for a much more important reason: we can write a single autoloader function that can find *anyone's* code: our code or third-party code that we include in our project.

The Famous PSR-0

The idea of naming your classes and files in this way is called PSR-0. You see, there's a lovable group called the PHP FIG. It's basically the United Nations of PHP: they come together to agree on standards that everyone should follow. PSR-0 was the first standard... called 0 because we geeks start counting, well, at 0.

It simply says that Thou shalt call your class names the same as your filenames plus .php and you shall have your directory structures match up with your namespaces.

Hello Composer

Why do we care? Because instead of having to write this autoloader by hand, you can actually include an *outside* library that takes care of all of it for us. The library is called Jordi, I mean, Composer: you may have heard of it.

Let's get it: Go to getcomposer.org and hit download. Copy the lines up here: if you're on Windows, you may see slightly different instructions. Then move into your terminal, open a new tab, and paste those in:

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('SHA384', 'composer-setup.php') === 'e115a8dc7871f15d853148a7fbac7da27d6c0030b848d9b3dc09e2a0388afed865e6a3d6b3c0fad45c48e2b5fc1196ae') { echo 'Installer verified';
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

This is downloading Composer, which is just a single, executable file. Usually people use Composer to download external libraries they want to use in their project. It's PHP's package manager.

But it has a second superpower: autoloading. When this command finishes, you'll end up with a composer.phar file. This is a PHP executable. We'll come back to it in a second.

Configuring Autoloading

To tell Composer to do the autoloading for us, all you need is a small configuration file called composer.json. Inside, add an autoload key, then a psr-4 key, and empty quotes set to lib:

7 lines composer.json

```
{
"autoload": {
"psr-4": {
"": "lib/"
}
}
}
```

That's it.

Remember how I said this rule is called PSR-0? Well PSR-4 is a slight amendment to PSR-0, but they both refer to the same thing. This tells Composer that we want to autoload using the PSR-0 convention, and that it should look for *all* classes inside the lib directory. That's it.

Back in your terminal, run:

```
php composer.phar install
```

This command normally downloads any external packages that we need - but we haven't defined any. But it *also* generates some autoload files inside a new vendor/ directory.

To use those, open bootstrap.php, delete all the manual autoload stuff, and replace it with just require __DIR__vendor/autoload.php, which is one of the files that composer just generated:

10 lines bootstrap.php

```
... lines 1 - 2
require __DIR__.'/vendor/autoload.php';
... lines 4 - 10
```

That's it.

You also usually don't commit the vendor/ directory to your Git repository: team members just run this same command when they download the project.

Let's see if it works! Go back and refresh! It does! And as we add more classes and more directories to lib/, everything will keep working. AND, if you guys want to start downloading external libraries into your project via Composer, you can do that too and immediately reference those classes without needing to worry about require statements or autoloaders. Composer takes care of everything. Thanks Jordi!

Chapter 9: Throwing an Exception (and a Party)

Let's talk about something totally different: a powerful part of object-oriented code called *exceptions*.

In index.php, we create a BrokenShip object. I'm going to do something crazy, guys. I'm going to say, $brokenShip->setStrength() and pass it... banana:

146 lines index.php

```
... lines 1 - 4
use Model\BrokenShip;
... lines 6 - 13
$brokenShip = new BrokenShip('I am so broken');
$brokenShip->setStrength('banana');
... lines 16 - 146
```

That strength makes no sense. And if we try to battle using this ship, we should get *some* sort of error. But when we refresh... well, it *is* an error: but not exactly what I expected.

This error is coming from AbstractShip line 65. Open that up. I want you to look at 2 exceptional things here:

125 lines lib/Model/AbstractShip.php

```
... lines 1 - 4
abstract class AbstractShip
{
... lines 7 - 44
public function setStrength($number)
{
if (!is_numeric($number)) {
throw new Exception('Invalid strength passed '.$number);
}
$this->strength = $number;
}
... lines 53 - 123
}
```

First, we planned ahead. When we created the setStrength() method, we said:

> You know what? This needs to be a number, so if somebody passes something dumb like "banana," then let's check for that and trigger an error.

And second, in order to trigger an error, we threw an *exception*:

125 lines lib/Model/AbstractShip.php

```
... lines 1 - 4
abstract class AbstractShip
{
... lines 7 - 44
public function setStrength($number)
{
if (!is_numeric($number)) {
throw new Exception('Invalid strength passed '.$number);
}
... lines 50 - 51
}
... lines 53 - 123
}
```

And that's actually what I want to talk about: Exceptions are classes, but they're completely special.

But first, Exception is a core PHP class, and when we added a namespace to this file, we forgot to change it to \Exception:

125 lines lib/Model/AbstractShip.php

```
... lines 1 - 4
abstract class AbstractShip
{
... lines 7 - 44
public function setStrength($number)
{
if (!is_numeric($number)) {
throw new \Exception('Invalid strength passed '.$number);
}
... lines 50 - 51
}
... lines 53 - 123
}
```

That's better. Now refresh again. *This* is a much better error:

> Uncaught Exception: Invalid strength passed "banana"

When things go Wrong: Throw an Exception

When things go wrong, we throw exceptions. Why? Well, first: it stops execution of the page and immediately shows us a nice error.

Tip

If you install the XDebug extension, exception messages are more helpful, prettier and will fix your code for you (ok, that last part is a lie).

Catching Exceptions: Much Better than Catching a Cold

Second, exceptions are *catchable*. Here's what that means.

Suppose that I wanted to kill the page right here with an error. I actually have two options: I can throw an exception, *or* I could print some error message and use a die statement to stop execution.

But when you use a die statement, your script is *truly* done: none of your other code executes. But with an exception, you can actually try to *recover* and keep going!

Let's look at how. Open up PdoShipStorage. Inside fetchAllShipsData(), change the table name to fooooo:

35 lines lib/Service/PdoShipStorage.php

```
... lines 1 - 4
class PdoShipStorage implements ShipStorageInterface
{
... lines 7 - 13
public function fetchAllShipsData()
{
$statement = $this->pdo->prepare('SELECT * FROM FOOOOO');
... lines 17 - 19
}
... lines 21 - 33
}
```

That clearly will *not* work. This method is called by ShipLoader, inside getShips():

67 lines lib/Service/ShipLoader.php

```
... lines 1 - 8
class ShipLoader
{
... lines 11 - 20
public function getShips()
{
... lines 23 - 24
$shipsData = $this->queryForShips();
... lines 26 - 31
}
... lines 33 - 60
private function queryForShips()
{
return $this->shipStorage->fetchAllShipsData();
}
}
```

When we try to run this, we get an *exception*:

> Base table or view not found

The error is coming from PdoShipStorage on line 18, but we can also see the line that called this: ShipLoader line 23.

Now, what if we knew that *sometimes*, for some reason, an exception like this might be thrown when we call fetchAllShipsData(). And when that happens, we *don't* want to kill the page or show an error. Instead, we want to - temporarily - render the page with zero ships.

How can we do this? First, surround the line - or lines - that might fail with a try-catch block. In the catch, add \Exception $e:

72 lines lib/Service/ShipLoader.php

```
... lines 1 - 8
class ShipLoader
{
... lines 11 - 60
private function queryForShips()
{
try {
return $this->shipStorage->fetchAllShipsData();
} catch (\Exception $e) {
... lines 66 - 67
}
}
}
```

Now, if the fetchAllShipsData() method throws an exception, the page will *not* die. Instead, the code inside catch will be called and then execution will keep going like normal:

72 lines lib/Service/ShipLoader.php

```
... lines 1 - 8
class ShipLoader
{
... lines 11 - 60
private function queryForShips()
{
try {
return $this->shipStorage->fetchAllShipsData();
} catch (\Exception $e) {
// if all else fails, just return an empty array
return [];
}
}
}
```

That means, we can say $shipData = array().

Using the Exception Object

And just like that, the page works. That's the power of exceptions. When you throw an exception, any code that calls your code has the opportunity to catch the exception and say:

> No no no, I don't want the page to die. Instead, let's do something else.

Of course, we probably also don't want this to fail silently without us knowing, so you might trigger an error and print the message for our logs. Notice, in catch, we have access to the Exception object, and every exception has a getMessage() method on it. Use that to trigger an error to our logs:

73 lines lib/Service/ShipLoader.php

```
... lines 1 - 8
class ShipLoader
{
... lines 11 - 60
private function queryForShips()
{
try {
return $this->shipStorage->fetchAllShipsData();
} catch (\Exception $e) {
trigger_error('Exception! '.$e->getMessage());
// if all else fails, just return an empty array
return [];
}
}
}
```

Ok, refresh! Right now, we see the error on top of the page. But that's just because of our error_reporting settings in php.ini. On production, this wouldn't display, but *would* write a line to our logs.

Chapter 10: Different Exception Classes

Catching an exception is really powerful. But you can get even fancier.

For right now, var_dump() the Exception object:

74 lines lib/Service/ShipLoader.php

```php
... lines 1 - 8
class ShipLoader
{
    ... lines 11 - 60
    private function queryForShips()
    {
        try {
            ... line 64
        } catch (\Exception $e) {
            var_dump($e);
            ... lines 67 - 69
        }
    }
}
```

Ok, this object is actually a PDOException. Two Important things: *all* exceptions are objects, and all exception classes ultimately extend PHP's base Exception class. So if you could look at the source code for PDOException, you'd see that it extends Exception:

And this ends up giving us a lot more flexibility when working with exceptions. Why? Remember, we're pretending that - for some reason - we occasionally have some database problems that cause a PDOException to be thrown. When that happens, we want to recover and just show zero ships. And we've got that.

But what if something *else* goes wrong inside fetchAllShipsData() that has *nothing* to do with talking to the database. Well, that would be truly unexpected, and in those cases, I want to let the exception be thrown like normal so we can see it while we're developing.

So here's the question: how can we catch PDOException objects, but not *any* others? By changing the catch to \PDOException:

73 lines lib/Service/ShipLoader.php

```php
... lines 1 - 8
class ShipLoader
{
    ... lines 11 - 60
    private function queryForShips()
    {
        try {
            ... line 64
        } catch (\PDOException $e) {
            trigger_error('Database Exception! '.$e->getMessage());
            ... lines 67 - 68
        }
    }
}
```

I'll also change the message to "Database Exception".

Refresh! Cool: it still catches that exception. But check this out: go back into PdoShipStorage and - before the query - throw a different exception: there's one called InvalidArgumentException:

36 lines lib/Service/PdoShipStorage.php

```php
... lines 1 - 4
class PdoShipStorage implements ShipStorageInterface
{
    ... lines 7 - 13
    public function fetchAllShipsData()
    {
        throw new \InvalidArgumentException('something else went wrong!');
        ... lines 17 - 20
    }
    ... lines 22 - 34
}
```

There's nothing special about this class: PHP has several built-in exceptions, and you can use whatever one feels right for your scenario.

But, it should *not* be caught by our try-catch. Try it out.

Yes! It totally kills the page.

Exceptions are something that you'll get used to leveraging as you develop more. But here's the key takeaway: when things go wrong, throw an exception.

Don't Get Too Clever

Oftentimes, I see people try to *not* throw exceptions. Instead, they try to recover in some way. Don't do that. I would rather throw an exception, see the error in my error log and fix it than try to render a broken page and never realize that there's a bug in my code.

In fact, most frameworks have a pretty easy way to automatically notify you - like via Slack or by email - whenever an exception is thrown on your site.

Let's fix our code: take out the throw new exception and change the table back to ship. All better.

Chapter 11: Magic Methods: __toString() __get, __set()

If I give you an object, could you print it? What I mean is, in battle.php, after we determine the winners, we echo $ship1->getName(), which is of course a string:

112 lines battle.php

```
... lines 1 - 39
<html>
... lines 41 - 59
<body>
<div class="container">
... lines 62 - 64
<div>
<h2 class="text-center">The Matchup:</h2>
<p class="text-center">
<br>
<?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': ''; ?>
</p>
</div>
... lines 74 - 108
</div>
</body>
</html>
```

But could we just print $ship1 and $ship2?

112 lines battle.php

```
... lines 1 - 39
<html>
... lines 41 - 59
<body>
<div class="container">
... lines 62 - 64
<div>
<h2 class="text-center">The Matchup:</h2>
<p class="text-center">
<br>
<?php echo $ship1Quantity; ?> <?php echo $ship1; ?><?php echo $ship1Quantity > 1 ? 's': ''; ?>
VS.
<?php echo $ship2Quantity; ?> <?php echo $ship2; ?><?php echo $ship2Quantity > 1 ? 's': ''; ?>
</p>
</div>
... lines 74 - 108
</div>
</body>
</html>
```

Does it make sense to print an object? The answer is... no. Try to battle, you get a very clear error that says:

> Object of class Model\RebelShip could not be converted to string in battle.php.

Remember this error: you'll eventually try to print an object on accident and see this!

But you CAN Print an Object

Why am I telling you this seemingly small and obvious fact? Because I'm lying! You *can* print objects! You just have to do a little bit more work.

Here's the big picture: there are ways to give a class super-powers - like the ability to be printed or - as we'll see next - the ability to pretend like it's an array.

Open up AbstractShip. To make objects of this class printable, go to the bottom and create a new public function __toString(). Inside, return $this->getName():

130 lines lib/Model/AbstractShip.php

```
... lines 1 - 4
abstract class AbstractShip
{
... lines 7 - 124
public function __toString()
{
return $this->getName();
}
}
```

Go back, refresh, and now it works just fine.

By adding the __toString() method - we gave PHP the ability to convert our object into a string. The __toString() *must* be called exactly like this, and there are other methods that take on special meaning. They all start with __, and we've already seen one: __construct():

130 lines lib/Model/AbstractShip.php

```
... lines 1 - 4
abstract class AbstractShip
{
... lines 7 - 29
public function __construct($name)
{
$this->name = $name;
}
... lines 34 - 128
}
```

These are collectively called Magic Methods.

The Magic __get()

There are actually just a few magic methods: let's look at another common one. In battle.php, scroll down a little bit to where it shows the ship health. Change this: instead of $ship1->getStrength(), say $ship1->strength:

112 lines battle.php
```
... lines 1 - 39
<html>
... lines 41 - 59
<body>
<div class="container">
... lines 62 - 73
<div class="result-box center-block">
... lines 75 - 95
<dl class="dl-horizontal">
... line 97
<dd><?php echo $ship1->strength; ?></dd>
... lines 99 - 100
</dl>
</div>
... lines 103 - 108
</div>
</body>
</html>
```

This should *not* work, and PHPStorm tells us why: the member - meaning property - has private access. We can't access a private property from outside the class.

But once again - via a magic method - you can bend the rules. This time, add a public function __get() with a single argument: $propertyName. For now, just dump that:

135 lines lib/Model/AbstractShip.php
```
... lines 1 - 4
abstract class AbstractShip
{
... lines 7 - 129
public function __get($propertyName)
{
var_dump($propertyName);die;
}
}
```

Refresh to see what happens. Interesting! It dumps the string strength. Here's the magic: if you reference a property on your object that is not accessible - either because it doesn't exist or is private or protected - *and* you have an __get() method, then PHP will call that and pass you the property name.

Then - if you want - you can return its value. Add return $this->$propertyName:

135 lines lib/Model/AbstractShip.php
```
... lines 1 - 4
abstract class AbstractShip
{
... lines 7 - 129
public function __get($propertyName)
{
return $this->$propertyName;
}
}
```

This looks weird: PHP will see $propertyName, evaluate that to strength, and then return $this->strength.

Refresh again. It *works*!

Not surprisingly, there's also a method called __set(), which allows you to *assign* a value to a non-existent property, like $ship->strength = 100.

Don't be Too Clever

Now, *just* because you have all this new power *doesn't* mean you should use it. As soon as you add things like __get(), it starts to break your object oriented rules. All of a sudden, even though it *looks* like strength is private, I actually *can* get it... so it's not really private.

You also won't get reliable auto completions from your editor - it has a hard time figuring out what you're doing in these magic methods.

So my recommendation is: avoid using magic methods, except for __toString() and __construct().

But, you *do* need to know these exist: even if *you* don't use them, other libraries will, which might be confusing if you're not watching for it.

But beyond magic methods, there *are* other super powers you can give your objects that I *do* love. Let's look at those.

Chapter 12: ArrayAccess: Treat your Object like an Array

Let's do something *else* that's not possible. BattleResult is an object:

112 lines battle.php

```
... lines 1 - 39
<html>
... lines 41 - 59
<body>
<div class="container">
... lines 62 - 73
<div class="result-box center-block">
<h3 class="text-center audiowide">
Winner:
<?php if ($battleResult->isThereAWinner()): ?>
<?php echo $battleResult->getWinningShip()->getName(); ?>
<?php else: ?>
Nobody
<?php endif; ?>
</h3>
... lines 83 - 101
</div>
... lines 103 - 108
</div>
</body>
</html>
```

But, use your imagination: its only real job is to hold these three properties, plus it does have one extra method: isThereAWinner():

57 lines lib/Model/BattleResult.php

```
... lines 1 - 4
class BattleResult
{
private $usedJediPowers;
private $winningShip;
private $losingShip;
... lines 10 - 51
public function isThereAWinner()
{
return $this->getWinningShip() !== null;
}
}
```

But for the most part, it's kind of a glorified associative array.

Let's get crazy and *treat* the object like an array: say $battleResults['winningShip']->getName():

112 lines battle.php

```
... lines 1 - 39
<html>
... lines 41 - 59
<body>
<div class="container">
... lines 62 - 73
<div class="result-box center-block">
<h3 class="text-center audiowide">
Winner:
<?php if ($battleResult->isThereAWinner()): ?>
<?php echo $battleResult['winningShip']->getName(); ?>
... lines 79 - 80
<?php endif; ?>
</h3>
... lines 83 - 101
</div>
... lines 103 - 108
</div>
</body>
</html>
```

That shouldn't work, but let's refresh and try it. Ah yes:

> Cannot use object of type Model\BattleResult as array in battle.php.

It's right - we're breaking the rules.

The ArrayAccess Interface

After the last chapter, you might expect me to go into BattleResults and add some new magic method down at the bottom that would make this legal. But nope!

There is actually a *second* way to add special behavior to a class, and this method involves interfaces. Basically, PHP has a group of built-in interfaces and each gives your class a different super-power if you implement it.

The most famous is probably \ArrayAccess.

Of course as soon as you implement any interface, it will require you to add some methods. In this case, PhpStorm is telling me that I needed offsetGet(), offsetUnset(), offsetExist() and offsetSet():

77 lines lib/Model/BattleResult.php

```
... lines 1 - 4
class BattleResult implements \ArrayAccess
{
... lines 7 - 75
}
```

Ok, let's do that, but with a little help from my editor. In PhpStorm, I can go to the "Code"->"Generate" menu and select "Implement Methods". Select these 4:

77 lines lib/Model/BattleResult.php

```
... lines 1 - 4
class BattleResult implements \ArrayAccess
{
... lines 7 - 56
public function offsetExists($offset)
{
... line 59
}
public function offsetGet($offset)
{
... line 64
}
public function offsetSet($offset, $value)
{
... line 69
}
public function offsetUnset($offset)
{
... line 74
}
}
```

Cool!

And just by doing this, it's *legal* to treat our object like an array. And when someone tries to access some array key - like winningShip - we'll just return that property instead.

So, for offsetExists(), use a function called property_exists() and pass it $this and $offset: that will be whatever key the user is trying to access:

77 lines lib/Model/BattleResult.php

```
... lines 1 - 4
class BattleResult implements \ArrayAccess
{
... lines 7 - 56
public function offsetExists($offset)
{
return property_exists($this, $offset);
}
... lines 61 - 75
}
```

For offsetGet(), return $this->$offset and in offsetSet(), say $this->$offset = $value:

77 lines lib/Model/BattleResult.php

```
... lines 1 - 4
class BattleResult implements \ArrayAccess
{
... lines 7 - 61
public function offsetGet($offset)
{
return $this->$offset;
}
public function offsetSet($offset, $value)
{
$this->$offset = $value;
}
... lines 71 - 75
}
```

And finally - even though it would be weird from someone to unset one of our keys, let's make that legal by removing the property: unset($this->$offset):

77 lines lib/Model/BattleResult.php

```
... lines 1 - 4
class BattleResult implements \ArrayAccess
{
... lines 7 - 71
public function offsetUnset($offset)
{
unset($this->$offset);
}
}
```

Ok, this is a little weird, but it works. Now, just like with magic methods, don't run and use this everywhere for no reason. But occasionally, it might come in handy. And more importantly, you *will* see this sometimes in outside libraries. This means that even though something *looks* like an array, it might actually be an object.

Let me show you just *one* other really cool, magic thing - this is my favorite. Right now, in ShipLoader, the getShips() method return an array:

73 lines lib/Service/ShipLoader.php

```php
... lines 1 - 8
class ShipLoader
{
    ... lines 11 - 20
    public function getShips()
    {
        $ships = array();
        $shipsData = $this->queryForShips();
        foreach ($shipsData as $shipData) {
            $ships[] = $this->createShipFromData($shipData);
        }
        return $ships;
    }
    ... lines 33 - 70
}
```

Instead of doing that, I'm going to return an object - a ShipCollection object. Don't ask why yet. I'll show you some reasons in a minute.

Creating ShipCollection

First create a new PHP class called ShipCollection:

17 lines lib/Model/ShipCollection.php

```php
... lines 1 - 2
namespace Model;
class ShipCollection
{
    ... lines 7 - 15
}
```

Hey, check it out: PhpStorm already correctly-guessed that this should have the Model namespace: it understands our PSR-0 naming convention.

Inside, add a private $ships property: this will be an array of Ship objects. Then add a public function __construct() method, give it a $ships argument, and set that property inside:

17 lines lib/Model/ShipCollection.php

```php
... lines 1 - 4
class ShipCollection
{
    ... lines 7 - 9
    private $ships;
    public function __construct(array $ships)
    {
        $this->ships = $ships;
    }
}
```

Above the $ships *just* to help our editor with autocompletion later, add some PHP Doc that says that this is an array of AbstractShip:

17 lines lib/Model/ShipCollection.php

```php
... lines 1 - 4
class ShipCollection
{
    /**
     * @var AbstractShip[]
     */
    private $ships;
    ... lines 11 - 15
}
```

Obviously, ShipCollection is a class... but its *only* purpose is to be a small wrapper around an array. In ShipLoader, instead of returning the array, return a new ShipCollection() object and pass it $ships:

74 lines lib/Service/ShipLoader.php

```php
... lines 1 - 7
use Model\ShipCollection;
class ShipLoader
{
    ... lines 12 - 21
    public function getShips()
    {
        ... lines 24 - 31
        return new ShipCollection($ships);
    }
    ... lines 34 - 71
}
```

Now, stop: we're referencing ShipCollection inside of ShipLoader, so we need a use statement for it. Go to the top to add it. But wait! It's already there! Thank you PhpStorm: it added it automatically for

me when I auto-completed the class name. Whether your editor does this or not, just make sure to *not* forget those use statements!

Finally, above the method, we're *not* returning an array of AbstractShip objects anymore: we're now returning a ShipCollection:

74 lines lib/Service/ShipLoader.php

```
... lines 1 - 7
use Model\ShipCollection;
class ShipLoader
{
    ... lines 12 - 18
    /**
    * @return ShipCollection
    */
    public function getShips()
    {
        ... lines 24 - 31
        return new ShipCollection($ships);
    }
    ... lines 34 - 71
}
```

Cool Now again, don't worry about *why* we're doing this yet. For now, let's try to fix our app.

Implementing ArrayAccess First

First, go to index.php. Boom!

> Cannot use object of type ShipCollection as array in index.php on line 13.

No surprise. After creating the $brokenShip, we're trying to add it to the ShipCollection as if it were an array!

145 lines index.php

```
... lines 1 - 11
$ships = $shipLoader->getShips();
$brokenShip = new BrokenShip('Just a hunk of metal');
$ships[] = $brokenShip;
... lines 16 - 145
```

That's not allowed... oh wait it is! Open ShipCollection and make it implement \ArrayAccess:

37 lines lib/Model/ShipCollection.php

```
... lines 1 - 4
class ShipCollection implements \ArrayAccess
{
    ... lines 7 - 35
}
```

Now, at the bottom, I'll open the "Code"->"Generate" menu and implement the same 4 methods as before. This is even easier now: in offsetExists(), use array_key_exists($offset, $this->ships). The other methods are even easier: I'll fill each in by acting on the $ships array property:

37 lines lib/Model/ShipCollection.php

```
... lines 1 - 4
class ShipCollection implements \ArrayAccess
{
    ... lines 7 - 16
    public function offsetExists($offset)
    {
        return array_key_exists($offset, $this->ships);
    }
    public function offsetGet($offset)
    {
        return $this->ships[$offset];
    }
    public function offsetSet($offset, $value)
    {
        $this->ships[$offset] = $value;
    }
    public function offsetUnset($offset)
    {
        unset($this->ships[$offset]);
    }
}
```

Perfect! The ShipCollection object can now act like an array.

So refresh again! It works!

You can't Loop Over an Object :(

Ok, let's start a battle. Woh: check this out - there are *no* ships. What's going on here?

Look back at index.php:

145 lines index.php

```
... lines 1 - 36
<html>
... lines 38 - 62
<body>
<div class="container">
<div class="page-header">
<h1>OO Battleships of Space</h1>
</div>
<table class="table table-hover">
... lines 69 - 79
<tbody>
<?php foreach ($ships as $ship): ?>
... lines 82 - 95
<?php endforeach; ?>
</tbody>
</table>
... lines 99 - 141
</div>
</body>
</html>
```

Eventually we try to *loop* over the $ships variable but this is a ShipCollection object! It turns out that after implementing ArrayAccess, we can use the array syntax with an object, but we still *cannot* loop over it like an array.

The IteratorAggregate Interface

Can we teach PHP *how* to loop over our object? Absolutely: and the answer is another interface. To implement a second interface, add a comma and then use \IteratorAggregate:

42 lines lib/Model/ShipCollection.php
```
... lines 1 - 4
class ShipCollection implements \ArrayAccess, \IteratorAggregate
... lines 6 - 42
```

Repeat our trick from before: "Code"->"Generate" and then "Implement Methods". This time we only need to add *one* method: getIterator(). The easiest way to make this work is to return another core helper class: return new \ArrayIterator() and pass that $this->ships:

42 lines lib/Model/ShipCollection.php
```
... lines 1 - 4
class ShipCollection implements \ArrayAccess, \IteratorAggregate
{
... lines 7 - 36
public function getIterator()
{
return new \ArrayIterator($this->ships);
}
}
```

This tells PHP that when we try to loop over this object, it should actually loop over the $ships array property.

Ok, give it a try. Hey guys, we have ships! By adding 2 interfaces, we've made our ShipCollection object look and act almost *exactly* like an array.

Why did we Do this?

Ok, let's *finally* answer the question: why did we do this? Because sometimes, it might be useful to add some helpful *methods* to an array. Well, of course you can't do that, but you *can* add methods to a class.

For example, add a new method called public function removeAllBrokenShips(), because maybe we want a collection of *only* working ships. By adding this method, that would be really easy:

51 lines lib/Model/ShipCollection.php
```
... lines 1 - 4
class ShipCollection implements \ArrayAccess, \IteratorAggregate
{
... lines 7 - 41
public function removeAllBrokenShips()
{
... lines 44 - 48
}
}
```

Inside, loop over $this->ships as $key => $ship. Then, if !$ship->isFunctional(), unset($this->ships[$key]):

51 lines lib/Model/ShipCollection.php
```
... lines 1 - 4
class ShipCollection implements \ArrayAccess, \IteratorAggregate
{
... lines 7 - 41
public function removeAllBrokenShips()
{
foreach ($this->ships as $key => $ship) {
if (!$ship->isFunctional()) {
unset($this->ships[$key]);
}
}
}
}
```

Let's test this fancy new method out. In index.php, call $ships->removeAllBrokenShips():

147 lines index.php

```
... lines 1 - 11
$ships = $shipLoader->getShips();
$brokenShip = new BrokenShip('Just a hunk of metal');
$ships[] = $brokenShip;
... lines 16 - 18
$ships->removeAllBrokenShips();
... lines 20 - 147
```

This looks and acts like an array, but with the super-power to have methods on it. ooOOOooo.

Refresh and check this out: no more broken ships, ever.

There are more of these interfaces that have special powers, but these are the most common ones. And the most important thing is just to understand that they exist and how they work.

Chapter 14: Traits: "Horizontal" Reuse

Ok team: we need a new ship class - a BountyHunterShip. Start simple: in the model directory, add a new class: BountyHunterShip. Once again, PhpStorm already added the correct namespace for us:

22 lines lib/Model/BountyHunterShip.php

```
... lines 1 - 2
namespace Model;

class BountyHunterShip extends AbstractShip
{
    ... lines 7 - 20
}
```

Like every other ship, extend AbstractShip. Ah, but we do *not* need a use statement for this: that class lives in the same namespace as us.

Just like with an interface, when you extend an abstract class, you usually need to implement some methods. Go back to "Code"->"Generate"->"Implement Methods". Select the 3 that this class needs:

22 lines lib/Model/BountyHunterShip.php

```
... lines 1 - 4
class BountyHunterShip extends AbstractShip
{
    public function getJediFactor()
    {
        // TODO: Implement getJediFactor() method.
    }

    public function getType()
    {
        // TODO: Implement getType() method.
    }

    public function isFunctional()
    {
        // TODO: Implement isFunctional() method.
    }
}
```

Great!

Now, bounty hunter ships are interesting for a few reasons. First, they're never broken: those scrappy bounty hunters can always get the ship started. For isFunctional(), return true:

29 lines lib/Model/BountyHunterShip.php

```
... lines 1 - 4
class BountyHunterShip extends AbstractShip
{
    ... lines 7 - 18
    public function isFunctional()
    {
        return true;
    }
    ... lines 23 - 27
}
```

For getType(), return Bounty Hunter:

29 lines lib/Model/BountyHunterShip.php

```
... lines 1 - 4
class BountyHunterShip extends AbstractShip
{
    ... lines 7 - 13
    public function getType()
    {
        return 'Bounty Hunter';
    }
    ... lines 18 - 27
}
```

Simple. But the jediFactor will vary ship-by-ship. Add a JediFactor property and return that from inside getJediFactor():

29 lines lib/Model/BountyHunterShip.php

```
... lines 1 - 4
class BountyHunterShip extends AbstractShip
{
    private $jediFactor;

    public function getJediFactor()
    {
        return $this->jediFactor;
    }
    ... lines 13 - 27
}
```

At the bottom of the class add a public function setJediFactor() so that we can change this property: $this->jediFactor = $jediFactor:

29 lines lib/Model/BountyHunterShip.php

```
... lines 1 - 4
class BountyHunterShip extends AbstractShip
{
... lines 7 - 23
public function setJediFactor($jediFactor)
{
$this->jediFactor = $jediFactor;
}
}
```

Cool!

To get one of these into our system, let's do something simple. Open ShipLoader. At the bottom of getShips(), add a new ship to the collection: $ships[] = new BountyHunterShip() called 'Slave I' - Boba Fett's famous ship:

77 lines lib/Service/ShipLoader.php

```
... lines 1 - 10
class ShipLoader
{
... lines 13 - 22
public function getShips()
{
... lines 25 - 28
foreach ($shipsData as $shipData) {
$ships[] = $this->createShipFromData($shipData);
}
// Boba Fett's ship
$ships[] = new BountyHunterShip('Slave I');
... lines 34 - 35
}
... lines 37 - 74
}
```

Ok, head back and refresh! Yes! Slave I - Bounty Hunter, and it's not broken. That was easy.

Code Duplication

So, what's the problem? Look at BountyHunterShip and also look at Ship: there's some duplication. Both classes have a jediFactor property, a getJediFactor() method that returns this, and a setJediFactor that changes it.

Duplication is a bummer. How can we fix this? Well, we could use inheritance. But in this case, it's weird.

For example, we could make BountyHunterShip extend Ship, but then it would inherit this extra stuff that we don't really want or need. We could make it work, but I just don't like it.

Ok, what about making Ship extend BountyHunterShip? That just completely *feels* wrong: philosophically, not all Ships are BountyHunterShips - it's just not the right way to model these classes.

Are we stuck? What we want is a way to *just* share these 3 things: the jediFactor property, getJediFactor() and setJediFactor(). When you only need to share a few things, the right answer might be a *trait*.

Hello Mr Trait

Let's see what this trait thing is. In the Model directory, create a new PHP class called SettableJediFactorTrait. Now, change the class keyword to trait. Traits look and feel *exactly* like a normal class:

19 lines lib/Model/SettableJediFactorTrait.php

```
... lines 1 - 2
namespace Model;
trait SettableJediFactorTrait
{
... lines 7 - 17
}
```

In fact, open up BountyHunterShip and move the property and first method into the trait. Also grab setJediFactor() and put that in the trait too:

19 lines lib/Model/SettableJediFactorTrait.php

```
... lines 1 - 4
trait SettableJediFactorTrait
{
private $jediFactor;
public function getJediFactor()
{
return $this->jediFactor;
}
public function setJediFactor($jediFactor)
{
$this->jediFactor = $jediFactor;
}
}
```

The only difference between classes and traits is that traits can't be instantiated directly. Their purpose is for sharing code.

In BountyHunterShip, we can effectively copy and paste the contents of that trait into this class by going inside the class and adding use SettableJediFactorTrait:

19 lines lib/Model/BountyHunterShip.php

```
... lines 1 - 4
class BountyHunterShip extends AbstractShip
{
use SettableJediFactorTrait;
... lines 8 - 17
}
```

That use statement has *nothing* to do with the namespace use statements: it's just a coincidence. As soon as we do this, when PHP runs, it will copy the contents of the trait and pastes them into this class right before it executes our code. It's as if all the code from the trait actually lives inside this class.

And now, we can do the same thing inside of Ship: remove the jediFactor property and the two methods. At the top, use SettableJediFactorTrait:

29 lines lib/Model/Ship.php

```
... lines 1 - 4
class Ship extends AbstractShip
{
use SettableJediFactorTrait;
... lines 8 - 27
}
```

Give it a try! Refresh. No errors! In fact, nothing changes at all. This is called horizontal reuse: because you're not extending a parent class, you're just using methods and properties from other classes.

This is perfect for when you have a couple of classes that really don't have that much in common, but *do* have some shared functionality. Traits are *also* cool because you *cannot* extend multiple classes, but you *can* use multiple traits.

Chapter 15: Object Composition FTW!

In modern PHP, you're going to spend a lot of time working with *other* people's classes: via external libraries that you bring into your project to get things done faster. Of course, when you do that: you can't actually *edit* their code if you need to change or add some behavior.

Fortunately, OO code gives us some really neat ways to deal with this limitation.

Modifying a Class without Modifying it?

For the next few minutes, I want you to pretend like our PDOShipStorage is actually from a third-party library. In other words, we *can't* modify it.

Now, let's say whenever we call fetchAllShipsData(), it's *really* important for us to log to a file, how many ships were found. But if we can't edit this file, how can we do that?

Using Inheritance

There's actually two ways to do this, and both are pretty awesome. The first way is to create a new class that *extends* PDOShipStorage, like LoggablePDOShipStorage, and override some methods to add logging.

Nah, Use Composition

But forget that, let's skip to a better method called composition. First, create a new class in the Service directory called LoggableShipStorage, but do *not* extend PDOShipStorage:

34 lines lib/Service/LoggableShipStorage.php

```
... lines 1 - 2
namespace Service;
class LoggableShipStorage implements ShipStorageInterface
{
    ... lines 7 - 32
}
```

Now, the only rule for any ship storage object is that it needs to implement the ShipStorageInterface. Add that, and then go to our handy "Code"->"Generate" method to implement the 2 methods we need:

34 lines lib/Service/LoggableShipStorage.php

```
... lines 1 - 4
class LoggableShipStorage implements ShipStorageInterface
{
    ... lines 7 - 13
    public function fetchAllShipsData()
    {
        ... lines 16 - 20
    }
    public function fetchSingleShipData($id)
    {
        ... line 25
    }
    ... lines 27 - 32
}
```

So far, this is how every ship storage starts.

But LoggableShipStorage will *not* actually do any of the ship-loading work - it'll offload all that hard work to some *other* ship storage object, like PDOShipStorage. To do that, add a new private $shipStorage property and a public function __construct() method that accepts one ShipStorageInterface argument. Then, set that value onto the $shipStorage property:

34 lines lib/Service/LoggableShipStorage.php

```
... lines 1 - 4
class LoggableShipStorage implements ShipStorageInterface
{
    private $shipStorage;
    public function __construct(ShipStorageInterface $shipStorage)
    {
        $this->shipStorage = $shipStorage;
    }
    ... lines 13 - 32
}
```

For fetchAllShipData(), just return $this->shipStorage->fetchAllShipsData(). Repeat for the other method: return $this->shipStorage->fetchSingleShipData():

34 lines lib/Service/LoggableShipStorage.php

```
... lines 1 - 4
class LoggableShipStorage implements ShipStorageInterface
{
    ... lines 7 - 22
    public function fetchSingleShipData($id)
    {
        return $this->shipStorage->fetchSingleShipData($id);
    }
    ... lines 27 - 32
}
```

We've now created a *wrapper* object that offloads all of the work to an internal ship storage object. This is composition: you put one object *inside* of another.

To use the new class, open up Container. Inside getShipStorage(), add $this->shipStorage = new LoggableShipStorage() and pass it $this->shipStorage, which is the PDOShipStorage object:

77 lines lib/Service/Container.php

```
... lines 1 - 4
class Container
{
... lines 7 - 51
public function getShipStorage()
{
if ($this->shipStorage === null) {
$this->shipStorage = new PdoShipStorage($this->getPDO());
//$this->shipStorage = new JsonFileShipStorage(__DIR__.'/../../resources/ships.json');
// use "composition": put the PdoShipStorage inside the LoggableShipStorage
$this->shipStorage = new LoggableShipStorage($this->shipStorage);
}
... lines 61 - 62
}
... lines 64 - 75
}
```

We've just pulled a "fast one" on our application: our entire app thinks we're using PDOShipStorage, but we just changed that! If you refresh now, nothing is different: everything still eventually goes through the PDOShipStorage object.

But now, we have the opportunity to add *more* functionality - or to change functionality - in either of these methods.

Add some Logging!

To give a really simple example, replace the return statement with $ships = and add return $ships below that:

34 lines lib/Service/LoggableShipStorage.php

```
... lines 1 - 4
class LoggableShipStorage implements ShipStorageInterface
{
... lines 7 - 13
public function fetchAllShipsData()
{
$ships = $this->shipStorage->fetchAllShipsData();
... lines 17 - 19
return $ships;
}
... lines 22 - 32
}
```

Between, we could call some new log() method, passing it a string like: just fetched %s ships - passing that a count() of $ships:

34 lines lib/Service/LoggableShipStorage.php

```
... lines 1 - 4
class LoggableShipStorage implements ShipStorageInterface
{
... lines 7 - 13
public function fetchAllShipsData()
{
$ships = $this->shipStorage->fetchAllShipsData();
$this->log(sprintf('Just fetched %s ships', count($ships)));
return $ships;
}
... lines 22 - 32
}
```

Below, add a new private function log() with a $message argument:

34 lines lib/Service/LoggableShipStorage.php

```
... lines 1 - 4
class LoggableShipStorage implements ShipStorageInterface
{
... lines 7 - 27
private function log($message)
{
// todo - actually log this somewhere, instead of printing!
echo $message;
}
}
```

You should do something more intelligent in a real app, but to prove it's working, echo that message.

Let's refresh! There's our message!

Why is Composition Cool?

Wrapping one object inside of another like this is called composition. You see, when you want to change the behavior of an existing class, the first thing we always think of is

Oh, just extend that class and override some methods

But composition is another option, and it *does* have some subtle advantages. If we had *extended* PDOShipStorage and then later wanted to change back to our JsonFileShipStorage, then all of a sudden we would need to change our LoggableShipStorage to extend JsonFileShipStorage. But with composition, our wrapper class can work with *any* ShipStorageInterface. We could change just one line to go back to loading files from JSON and not lose our logging.

This isn't always a ground-breaking difference, but this is what people mean when they talk about "composition over inheritance".

Alright guys! I have tried to think of all the weird stuff that we haven't talked about with object oriented coding, and I've run out! You are now *super* qualified with this stuff - so get out there, find some classes, find some interfaces, make some traits, do some good, and just keep practicing. It's going to sink in more and more over time, and serve you for *years* to come, in many different languages.

See you next time!