PHP 7: The Important Stuff

With <3 from SymfonyCasts

Chapter 1: 1 <3 Speed & Throwable

Hey guys! I hate to start a tutorial off on a depressing note but... look... PHP 5 is *dead*. WAY dead. Like, it's not even supported anymore. PHP 5 is like that old, bad relationship that you just can't get out of. Hey, it's time to move on. You're better than PHP 5.

Also... did you know that PHP 7 is a full *2* numbers higher than PHP 5? I heard they skipped PHP 6 because 7 was just *too* awesome to fit into such a low number. Or... maybe because they messed up PHP 6. Anyways, let's pretend that it's because PHP 7 is so great... because it is.

This tutorial is about learning the *important* stuff... only. Look, you can spend *hours* reading the PHP 7 CHANGELOG. Believe us... we did it. I get it, they made *a lot* of stuff better... cool... but most of it isn't that critical. This means that we will *not* be talking about the spaceship operator... because it is apparently *not* a person who drives a flying saucer. Nope, it's an edge-case way to compare numbers and strings. So disappointing.

Speed Sells

So what *is* important in PHP 7? Honestly... the *biggest* selling point for upgrading is speed. PHP 7 performance is on point. To show it off, Zend made a cute infographic. Summary: PHP 7 equals zoooooom!

And that means you can take this to your manager and say:

> Hey buddy! When we upgrade to PHP 7, our pages will be faster and we can turn off like 10 servers.

And then they'll throw you a parade and promote you to CEO. Enjoy.

Setting up the Project

And now that we know how to *sell* the upgrade to management, let's get into the cool technical stuff.

As always, you should definitely enjoy a snack during the tutorial... and code along with me! Download the course code from this page and unzip it. Inside, you'll find a fancy start/ directory with the same code you see here. Follow the README.md file to get things set up. The last step will be to find your favorite terminal, go to the project directory, and run:

```
$ php bin/console server:run
```

to start the built-in web server. Open up the project in your browser: http://localhost:8000.

Welcome to AquaNote! A project we've been building in our Symfony series. For this tutorial, it'll be a nice skeleton to start with.

Proper Error Handling

Ok, the *first* feature of PHP 7 that gets me excited is... proper error handling. Stay with me: I promise this *is* exciting!

At the root of the project, create a new file called play-exceptions.php. Inside, let's do something fun: like, write some really bad code! We'll call an undefinedFunction(). And below, I'll write "continue processing file", even though we know that's crazy! This script will blow up *way* before that line.

5 lines | play-exceptions.php

```php
<?php
undefinedFunction();

echo "\n\nContinue processing file...";
```

Find your terminal, open up a new tab, and run:

```
$ php play-exceptions.php
```

Fatal error! Woohoo!

Catching Errors

In PHP 5, we could catch *exceptions*... but not errors. Sure, you could try to work with the error handler... but that's confusing stuff! In PHP 7... they fixed things! We can finally catch *errors*.

Start with a normal try-catch block. But instead of catching Exception, catch \Throwable. Yes! In PHP 7, you can write bad code, catch it, and even print out the error message!

9 lines | play-exceptions.php

```php
<?php
try {
    undefinedFunction();
} catch (\Throwable $error) {
    echo 'Now if you write bad code, you can catch it! ' . $error->getMessage();
}

echo "\n\nContinue processing file...";
```

Try the file again:

```
$ php play-exceptions.php
```

Haha! It *actually* runs.

About Throwable

About this Throwable thingy: it's actually a core *interface*. Here's the deal: we still have the core Exception class. And now, there is also an Error class. And both Exception and Error implement the Throwable interface. So if you want to catch both exceptions *and* errors, catch \Throwable. If you want to only catch exceptions, then catch \Exception. And if you want to only catch errors, use catch \Error. It's quite elegant.

And, just like with exceptions, there are different *types* of errors, each with its own class. For example, TypeError is thrown when you're passing an argument of a wrong *type* to a function. And actually, that's our next topic: the new scalar type system and strict mode!

Chapter 2: Scalar Type Hints

Of course the big big features of PHP 7 are scalar type-hinting and return types. Let's play with scalar type-hinting first.

Open up src/AppBundle/Controller/GenusController.php. Let's create a new page to play with:
public function typesExampleAction(). Above that, I'll use @Route("/types") to map a URL to this.

I already have a normal PHP class called Genus... which is a type of animal classification. It has an id, name, speciesCount, funFact and a few other things.

Back in the controller, let's create a new genus: $genus = new Genus(). The genus's name should obviously be a string. But, let's be difficult and set it to an integer: $genus->setName(4). Below, var_dump($genus); and then die;.

154 lines | src/AppBundle/Controller/GenusController.php

```php
<?php
... line 2
namespace AppBundle\Controller;
... lines 4 - 14
class GenusController extends Controller
{
    /**
     * @Route("/types")
     */
    public function typesExampleAction()
    {
        $genus = new Genus();
        $genus->setName(4);

        var_dump($genus);die;
    }
... lines 27 - 152
}
```

Cool! In the browser, head to /types to check it out. It works! Even though *we* think name should be a string, PHP lets us pass whatever we want. Right now, name is actually an int.

In PHP 5, we *could* type-hint arguments... but only with either array, callable or a class name. Well, no more! In PHP 7, we can type-hint with string... or int, float, bool or pizza. Wait, not pizza.

224 lines | src/AppBundle/Entity/Genus.php

```php
<?php
... lines 2 - 3
namespace AppBundle\Entity;
... lines 5 - 17
class Genus
{
... lines 20 - 102
    public function setName(string $name)
    {
        $this->name = $name;
    }
... lines 107 - 222
}
```

But, woh! PhpStorm is *super* mad about this. That's because my PHPStorm is living in the past! It's still configured to parse things as PHP 5. Open the PHP Storm settings and search for PHP. Under "Languages and Frameworks", set the version to 7.1 We'll also be showing off some 7.1 features.

So much happier! With *just* this one small change, go back, refresh and watch closely. Specifically, watch the "name integer 4" part. Woh! Now it's a *string* 4.

Weak Mode versus Strict Mode

You see, PHP 7 types now have two modes: weak mode and strict mode. When you use weak mode - which is the default - PHP will try to "coerce" the argument into whatever the type-hint is. In this case, it turns the integer 4 into a string 4. And we're totally accustomed to this in PHP: if you try to echo an integer or treat it like a string in any way, PHP automatically makes it a string.

Strict Mode (strict_types=1)

But if you use strict mode, things are much different. Instead of changing the value from an integer to a string, it will throw an error: a TypeError. How do we activate strict mode?

At the top of Genus, make the *very* first thing in the file say declare(strict_types=1);. This *must* be the first line in the file.

224 lines | src/AppBundle/Entity/Genus.php

```
<?php
declare(strict_types = 1);
... line 3
namespace AppBundle\Entity;
... lines 5 - 17
class Genus
{
... lines 20 - 222
}
```

But check this out: when we refresh... still no error! What's the deal? Copy that line, open GenusController and paste it here.

155 lines | src/AppBundle/Controller/GenusController.php

```
<?php
declare(strict_types = 1);
... line 3
namespace AppBundle\Controller;
... lines 5 - 15
class GenusController extends Controller
{
... lines 18 - 153
}
```

Now refresh again.

Boom!

This is the error we expected:

> TypeError Argument 1 passed to setName() must be type string, integer given.

This... is a bit confusing. The important thing is that strict_types is added to the file where we *pass* the value... not actually the file where we add the type-hint. When you add strict_types=1, you're saying:

> I want "strict" type-checking to be applied to all function calls I *make* from this file.

It's done this way on purpose: if you download an external library that uses scalar type-hints, *you* get to decide whether or not you want arguments you pass to its functions to be strictly type-checked. *You*, the developer, get to opt into strict mode.

Strict Mode or Weak Mode?

So... which should you use? It's up to you... just remember that you've been using PHP for *years* in "weak mode". If it's never bothered you that PHP automatically changes integers to strings instead of throwing an error, you might choose to keep using weak mode. When you enable strict mode, your code *will* be more predictable... but you'll also see - and need to correct - a lot more errors.

And remember! You can use scalar type-hints in either mode. There's also one more thing to consider when choosing between weak or strict mode: return types.

Chapter 3: Return Types

PHP 7 added scalar type-hinting. But it *also* added - *finally* - return types. These didn't exist at *all* in PHP 5, not even for objects. This gives us the ability to say that this method returns a string and that method returns some object. I love return types.

Start in GenusController: let's fix our code first. Say $genus->setName('Octopus'). Now, dump $genus->getName().

156 lines | src/AppBundle/Controller/GenusController.php

```
<?php
declare(strict_types = 1);
... line 3
namespace AppBundle\Controller;
... lines 5 - 15
class GenusController extends Controller
{
... lines 18 - 20
    public function typesExampleAction()
    {
        $genus = new Genus();
        $genus->setName('Octopus');
... line 25
        var_dump($genus->getName());
... line 27
    }
... lines 29 - 154
}
```

We know that this should return a string... but technically... it could return anything. Like, what if we get crazy and just return 5.

225 lines | src/AppBundle/Entity/Genus.php

```
<?php
... lines 2 - 3
namespace AppBundle\Entity;
... lines 5 - 17
class Genus
{
... lines 20 - 97
    public function getName()
    {
        return 5;
... line 101
    }
... lines 103 - 223
}
```

What happens? Well, no surprise... it returns 5, an *integer*. Yep, we have a method that should return a string, but is instead returning an integer. Lame!

Adding a Return Type

To fix this, add a return type. After the method name, add : string. Of course, this could be *any* type: a class name, int, bool, float, array, callable or even the new iterable. More on that later.

225 lines | src/AppBundle/Entity/Genus.php

```
<?php
declare(strict_types = 1);
... line 3
namespace AppBundle\Entity;
... lines 5 - 17
class Genus
{
... lines 20 - 97
    public function getName(): string
    {
... lines 100 - 101
    }
... lines 103 - 223
}
```

As *soon* as we do this, PhpStorm is furious all over again! And so is PHP: refresh!

> TypeError: Return value of getName() must be of the type string, integer returned

Return Types & Weak Versus Strict Mode

Yes! There is one *really* important thing happening behind the scenes: this throws an error *only* because this class is in *strict* mode.

What I mean is, if we changed Genus back to weak mode, then instead of throwing an error, PHP would try to turn the integer 5 into a string. The strict or weak mode affects argument type-hints *and* return types.

But here's the tricky part: in this case, the strict_types, that's important is the one in Genus. If you remove the declare(strict_types=1) on Genus and then refresh the page... it works!

Wait, wait, wait. When we type-hinted the *argument* in setName(), that caused an error when we put the *controller* in strict mode. But when we added the *return type*, suddenly it was important to use strict mode in the Genus class.

Here's the real, full explanation. When you add strict_types=1, it says:

> I want "strict" type-checking to be applied to all function calls I make *from* this file and all return types *in* this file.

Or, to be even more brief:

> I want "strict" type-checking to be applied to all values I control in this file.

The return value of getName() is something that *we* control and calculate in Genus. Thanks to the strict_types in Genus, PHP forces us to write good code and return the correct type.

But, with setName(), the argument value is being created *outside* of this class in GenusController. For that, the strict_types needs to be added there.

Actually, scalar type hinting and return types are *really* easy... except for the strict_types part. My advice is this: start adding a few strict_types to your code and see how you like it. It's a great feature, but your code will work fine with or without it.

In Genus, let's fix our code by returning $this->name. Now, life is good.

224 lines | src/AppBundle/Entity/Genus.php

```
<?php
... lines 2 - 17
class Genus
{
... lines 20 - 97
    public function getName(): string
    {
        return $this->name;
    }
... lines 102 - 222
}
```

But what if Genus did *not* have a name yet? In that case, getGenus() would return null. Is that allowed? Nope! null is *not* a string... and this can be really annoying. Fortunately, PHP 7.1 gives us nullable types.

Chapter 4: Nullable Types

Let's try an experiment: In GenusController, change our var_dump() to $genus->getFunFact(): another property on Genus that *should* be a string. If you refresh now... it's null! No surprise: we haven't set this yet and that method doesn't have a return type.

156 lines | src/AppBundle/Controller/GenusController.php

```php
<?php
declare(strict_types = 1);
... line 3
namespace AppBundle\Controller;
... lines 5 - 15
class GenusController extends Controller
{
... lines 18 - 20
    public function typesExampleAction()
    {
... lines 23 - 25
        var_dump($genus->getFunFact());
... line 27
    }
... lines 29 - 154
}
```

Now, add one: : string. Refresh again.

224 lines | src/AppBundle/Entity/Genus.php

```php
<?php
declare(strict_types = 1);
... line 3
namespace AppBundle\Entity;
... lines 5 - 17
class Genus
{
... lines 20 - 130
    public function getFunFact(): string
    {
        return $this->funFact;
    }
... lines 135 - 222
}
```

Explosion! This method returns *null*... which apparently is *not* a string. This actually made return types a pain in PHP 7... so in PHP 7.1, they fixed it! With "nullable" types. It works like this: if a return type can be null, add a ? in front of the type.

224 lines | src/AppBundle/Entity/Genus.php

```
... lines 1 - 17
class Genus
{
... lines 20 - 130
    public function getFunFact(): ?string
    {
        return $this->funFact;
    }
... lines 135 - 222
}
```

Yep, this method can return a string *or* null. And once again, life is good!

Nullable Type Arguments

Let's go further! In the controller, add $genus->setFunFact('This is fun') then var_dump($genus->getFunFact()). After, do $genus->setFunFact(null)... because null *should* be allowed.

160 lines | src/AppBundle/Controller/GenusController.php

```
<?php
declare(strict_types = 1);
... line 3
namespace AppBundle\Controller;
... lines 5 - 15
class GenusController extends Controller
{
... lines 18 - 20
    public function typesExampleAction()
    {
... lines 23 - 25
        $genus->setFunFact('This is fun');
        var_dump($genus->getFunFact());
... line 28
        $genus->setFunFact(null);
... lines 30 - 31
    }
... lines 33 - 158
}
```

Will this work? Totally! It prints the string, then it prints null. Unless... you type-hint the argument. Right now the argument to setFunFact() can be anything. Add the string type-hint.

224 lines | src/AppBundle/Entity/Genus.php

```
<?php
... lines 2 - 3
namespace AppBundle\Entity;
... lines 5 - 17
class Genus
{
... lines 20 - 135
    public function setFunFact(string $funFact)
    {
        $this->funFact = $funFact;
    }
... lines 140 - 222
}
```

No problem, right? Refresh! Ah! The first dump works, but setFunFact(null) *fails*. Duh, null is *not* a string.

With scalar type-hints, we suddenly need to think about things that were never a problem before. That's mostly good, but it's a bit more work. To make this argument nullable, add that same ? before the type. Without this, passing `null` as an argument is illegal... in both strict *and* weak modes.

226 lines | src/AppBundle/Entity/Genus.php

```
<?php
declare(strict_types = 1);
... line 3
namespace AppBundle\Entity;
... lines 5 - 17
class Genus
{
... lines 20 - 135
    public function setFunFact(?string $funFact): void
    {
... lines 138 - 140
    }
... lines 142 - 224
}
```

Refresh again. Beautiful!

?string versus string = null

Now, you might be thinking:

> Wait, wait wait. How is ?string different than string $funFact = null?

Hmm, good question! Because if I say string $funFact = null, that *does* allow a null value to be passed. In reality, these two syntaxes are *almost* the same. The difference is that when you default the argument to null, I'm allowed to call setFunFact() without *any* arguments: the argument is optional.

But with the nullable, ?string syntax, the argument *is* still required... it's simply that null is a valid value. That makes ?string better... unless you *actually* want the argument to be optional.

And by the way, the nullable ? works for *any* type, like classes. We'll see that in action next!

Chapter 5: Void Types & Refactoring an Entire Class

There's *one* last feature with return types: void return types. setFunFact() is a function... but it doesn't return anything. You can advertise that with : void. This literally means that the method returns... nothing. And not surprisingly, when we refresh, it works great... because we are in fact *not* returning anything.

226 lines | src/AppBundle/Entity/Genus.php

```
<?php
declare(strict_types = 1);
... line 3
namespace AppBundle\Entity;
... lines 5 - 17
class Genus
{
... lines 20 - 135
    public function setFunFact(?string $funFact): void
    {
... lines 138 - 140
    }
... lines 142 - 224
}
```

But now, try to return null. This will *not* work! When your return type is void, it literally means you do not return *anything*. Even null is not allowed.

226 lines | src/AppBundle/Entity/Genus.php

```
<?php
... lines 2 - 17
class Genus
{
... lines 20 - 135
    public function setFunFact(?string $funFact): void
    {
        $this->funFact = $funFact;

        return null;
    }
... lines 142 - 224
}
```

The void return type isn't that important, but it's useful because (A), it documents that this method does not return anything and (B) it guarantees that we don't get crazy and accidentally return something.

Oh, but you *can* use the return statement - as long as it it's just return with nothing after.

```
226 lines | src/AppBundle/Entity/Genus.php
<?php
... lines 2 - 17
class Genus
{
... lines 20 - 135
    public function setFunFact(?string $funFact): void
    {
        $this->funFact = $funFact;

        return;
    }
... lines 142 - 224
}
```

Updating all of Genus

Great news! We now know *everything* about scalar type hints and return types. And we can use our new super powers to update *everything* in Genus.

Let's do it! Start with getId(), this will return an int, but it should be nullable: there is no id at first. For getName(), the same thing, ?string. For setName(), it's up to you: this accepts a string, but I *am* going to allow it to be null. But if you never want that to happen, don't allow it! And of course, the method should return void.

```
224 lines | src/AppBundle/Entity/Genus.php
<?php
... lines 2 - 18
class Genus
{
... lines 21 - 93
    public function getId(): ?int
... lines 95 - 98
    public function getName(): ?string
... lines 100 - 103
    public function setName(?string $name): void
... lines 105 - 222
}
```

For getSubFamily(), this is easy: it will return a SubFamily object or null. The *cool* part is that we don't need the PHP doc anymore! We have a return type! Amazing!

```
224 lines | src/AppBundle/Entity/Genus.php
<?php
... lines 2 - 18
class Genus
{
... lines 21 - 108
    public function getSubFamily(): ?SubFamily
... lines 110 - 222
}
```

For setSubFamily(), mark this to return void. Notice that the argument is SubFamily $subFamily = null. If we want that argument to be required, we could change that to ?SubFamily. Your call!

```
224 lines | src/AppBundle/Entity/Genus.php

<?php
... lines 2 - 18
class Genus
{
... lines 21 - 113
    public function setSubFamily(SubFamily $subFamily = null): void
... lines 115 - 222
}
```

Let's keep going! getSpeciesCount() returns a nullable int, though we could give that a default value of 0 if we want, and remove the question mark. setSpeciesCount() accepts a nullable int and returns void. Again, the nullable part is up to you.

```
224 lines | src/AppBundle/Entity/Genus.php

<?php
... lines 2 - 18
class Genus
{
... lines 21 - 118
    public function getSpeciesCount(): ?int
... lines 120 - 123
    public function setSpeciesCount(?int $speciesCount): void
... lines 125 - 222
}
```

For getUpdatedAt(), set its return type to a nullable DateTimeInterface, because this starts as null. But notice... I'm returning a DateTime object... so why make the return-type DateTimeInterface? Well, it's up to you. With this type-hint, I could update my code later to return a DateTimeImmutable object. But more importantly, the return type is what you're "advertising" to outsiders. Choose whatever makes the most sense.

```
224 lines | src/AppBundle/Entity/Genus.php

<?php
... lines 2 - 18
class Genus
{
... lines 21 - 140
    public function getUpdatedAt(): \DateTimeInterface
... lines 142 - 222
}
```

Ok, let's get this done! setIsPublished takes a bool that is *not* nullable, and it returns void. getIsPublished will definitely return a bool - we initialized it to a bool when we defined the property.

```
224 lines | src/AppBundle/Entity/Genus.php

<?php
... lines 2 - 18
class Genus
{
... lines 21 - 145
    public function setIsPublished(bool $isPublished): void
... lines 147 - 150
    public function getIsPublished(): bool
... lines 152 - 222
}
```

For getNotes(), return a Collection and then update the PHPDoc to match. There are two interesting things happening. First, ArrayCollection implements this Collection interface, so using the interface is a bit more flexible. Normally, that's just a choice you can make: set your return type to the class you *know* you're returning... or use the more flexible interface. But actually, for Doctrine collections, you *must* use Collection. Depending on the situation, this property might be an ArrayCollection *or* a PersistentCollection... both of which implement the Collection interface. In other words, the *only* guarantee we can make is that this returns the Collection interface.

224 lines | src/AppBundle/Entity/Genus.php

```
<?php
... lines 2 - 18
class Genus
{
... lines 21 - 155
    /**
     * @return Collection|GenusNote[]
     */
    public function getNotes(): Collection
... lines 160 - 222
}
```

Second, this returns a collection of GenusNote objects. In other words, if you call getNotes() and then loop over the results, each item will be a GenusNote. But there's no way to denote that with return types. That's why we're keeping the |GenusNote[]. That helps my editor when looping.

getFirstDiscoveredAt() returns a nullable DateTimeInterface and setFirstDiscoveredAt() returns void. getSlug() will be a nullable string, setSlug() will accept a nullable string argument and return void. addGenusScientists() will return void and removeGenusScientist() the same. For getGenusScientists(), like before, I'll set the return type to Collection and update the PHP doc. Do the same for getExpertScientists(): return Collection. This PHPDoc is already correct... but I'll shorten it.

224 lines | src/AppBundle/Entity/Genus.php

```
<?php
... lines 2 - 18
class Genus
{
... lines 21 - 163
    public function getFirstDiscoveredAt(): ?\DateTimeInterface
... lines 165 - 168
    public function setFirstDiscoveredAt(\DateTime $firstDiscoveredAt = null): void
... lines 170 - 173
    public function getSlug(): ?string
... lines 175 - 178
    public function setSlug(?string $slug): void
... lines 180 - 183
    public function addGenusScientist(GenusScientist $genusScientist): void
... lines 185 - 194
    public function removeGenusScientist(GenusScientist $genusScientist): void
... lines 196 - 205
    /**
     * @return Collection|GenusScientist[]
     */
    public function getGenusScientists(): Collection
... lines 210 - 213
    /**
     * @return \Doctrine\Common\Collections\Collection|GenusScientist[]
     */
    public function getExpertScientists(): Collection
... lines 218 - 222
}
```

Phew! That makes our class a lot *tighter*: it's now more readable and more difficult to make mistakes. But it also took some work! The cool thing is that you have the power to add return types and type-hint arguments wherever you want. But you don't need to do it *everywhere*.

After *all* those changes... did we break anything? Find your browser and go to /genus/new. This is a "dummy" URL that creates and saves a Genus behind the scenes. So apparently, that still works! Click the Genus to go to its show page. Then, login using weaverryan+1@gmail.com and password iliketurtles. Once you do that, click to edit the genus.

Let's see... change the species to 5000, keep the fun fact empty and change the name. Hit enter!

Yay! Everything still works! And our Genus class is awesome!

Chapter 6: Private Constants

Okay enough with type hints and return types! PHP 7.1 added another cool feature for class constants: we can finally make them private!

In GenusController find, getNotesAction(). This is used by an AJAX call to load notes that appear at the bottom of the genus show page. For example, go to /genus and click on one of them. Bam! At the bottom, an AJAX call loads a list of notes, complete with an avatar. That comes from the avatarUri field that's returned.

Look at the /images/ part: that looks funny to me. All of our images are stored in that directory, and that's fine. But I hate having random strings like this in my code. This is a perfect place to use... drum roll... a constant!

Open GenusNote, which is the object we're rendering on this page. Add a new constant: const AVATAR_FILE_PREFIX = '/images';. Then, in the controller, use this: GenusNote::AVATAR_FILE_PREFIX.

103 lines | src/AppBundle/Entity/GenusNote.php

```
<?php
... line 2
namespace AppBundle\Entity;
... lines 4 - 10
class GenusNote
{
    const AVATAR_FILE_PREFIX = '/images';
... lines 14 - 101
}
```

160 lines | src/AppBundle/Controller/GenusController.php

```
... lines 1 - 15
class GenusController extends Controller
{
... lines 18 - 118
    public function getNotesAction(Genus $genus)
    {
... lines 121 - 122
        foreach ($genus->getNotes() as $note) {
            $notes[] = [
... lines 125 - 126
                'avatarUri' => GenusNote::AVATAR_FILE_PREFIX.'/'.$note->getUserAvatarFilename(),
... lines 128 - 129
            ];
        }
... lines 132 - 137
    }
... lines 139 - 158
}
```

So far, this is *all* stuff we've seen before. And when we refresh... yep! Everything still loads.

Private Constants

This is an improvement... but it would be even *better* if I could call a method on GenusNote to get the complete avatarUri string, instead of calculating it here in my controller. In other words, I *don't* want anyone to use our constant anymore: we're going to add a public *method* instead.

In PHP 7.1, we can say private const.

```
103 lines | src/AppBundle/Entity/GenusNote.php
... lines 1 - 10
class GenusNote
{
    private const AVATAR_FILE_PREFIX = '/images';
... lines 14 - 101
}
```

Now, accessing that constant from outside this class is illegal! That means, if we refresh, our AJAX calls are failing! On the web debug toolbar, yep! You can see the 500 errors! If I open the profiler and click "Exception", we see

> Cannot access private const from the controller

Awesome! So now that I can't use the constant anymore, I'll be looking for a public function to use instead. In GenusNote, add a public function getUserAvatarUri(). Hey! We're PHP 7 pros now, so add a string return type.

```
116 lines | src/AppBundle/Entity/GenusNote.php
... lines 1 - 10
class GenusNote
{
... lines 13 - 64
    public function getUserAvatarUri(): string
    {
... lines 67 - 73
    }
... lines 75 - 114
}
```

Before we add the logic, let's make things fancier. Suppose that *sometimes* there is *not* a userAvatarFilename value for a note. If that's true, let's show a default avatar image.

Back at the top, add another private const BLANK_AVATAR_FILENAME = 'blank.jpg'. We'll pretend that we have a blank.jpg file that should be used when there's no avatar.

```
116 lines | src/AppBundle/Entity/GenusNote.php
... lines 1 - 10
class GenusNote
{
... lines 13 - 14
    private const BLANK_AVATAR_FILENAME = 'blank.jpg';
... lines 16 - 114
}
```

Back in the new method, add $filename = $this->getUserAvatarFilename();. And, if (!$filename), then $filename = self::BLANK_AVATAR_FILENAME... because we *can* access the private constant from inside the class. Finish the method with return self::AVATAR_FILE_PREFIX.'/'.$filename;.

116 lines | src/AppBundle/Entity/GenusNote.php

```
... lines 1 - 10
class GenusNote
{
... lines 13 - 64
    public function getUserAvatarUri(): string
    {
        $filename = $this->getUserAvatarFilename();

        if (!$filename) {
            $filename = self::BLANK_AVATAR_FILENAME;
        }

        return self::AVATAR_FILE_PREFIX.'/'.$filename;
    }
... lines 75 - 114
}
```

Nice! Back in the controller, we're still accessing the private constant, which is *super* obvious. That'll push me to use the public function getUserAvatarUri().

160 lines | src/AppBundle/Controller/GenusController.php

```
... lines 1 - 15
class GenusController extends Controller
{
... lines 18 - 118
    public function getNotesAction(Genus $genus)
    {
... lines 121 - 122
        foreach ($genus->getNotes() as $note) {
            $notes[] = [
... lines 125 - 126
                'avatarUri' => $note->getUserAvatarUri(),
... lines 128 - 129
            ];
        }
... lines 132 - 137
    }
... lines 139 - 158
}
```

Refresh one more time! Love it!

Chapter 7: The iterable Pseudo-Type

I want to talk about yet *another* PHP 7.1 feature. I know, most of this tutorial is actually about PHP 7.1 features... not PHP 7.0. What can I say? They killed it with 7.1.

To show off this feature, open your Genus class and find the bottom. Add a fun new function called feed() with an array $food argument. Set the return type to a string.

I'm going to paste in some code. Basically, we pass an array with some food, and this returns a message. And if we pass *no* food, our genus looks at us funny...

233 lines | src/AppBundle/Entity/Genus.php

```
... lines 1 - 18
class Genus
{
... lines 21 - 223
    public function feed(array $food): string
    {
        if (count($food) === 0) {
            return sprintf('%s is looking at you in a funny way', $this->getName());
        }

        return sprintf('%s recently ate: %s', $this->getName(), implode(', ', $food));
    }
}
```

Let's go use this! In showAction(), create a $food array set to, how about, shrimp, clams, lobsters, and a shark! Pass a new recentlyAte variable into the template set to $genus->feed($food). Then, open the genus/show.html.twig template, add a new "Diet" key, and print recentlyAte.

163 lines | src/AppBundle/Controller/GenusController.php

```
... lines 1 - 15
class GenusController extends Controller
{
... lines 18 - 94
    public function showAction(Genus $genus)
    {
... lines 97 - 107
        $food = ['shrimp', 'clams', 'lobsters', 'shark'];
... line 109
        return $this->render('genus/show.html.twig', array(
... lines 111 - 112
            'recentNoteCount' => count($recentNotes),
            'recentlyAte' => $genus->feed($food),
        ));
    }
... lines 117 - 161
}
```

94 lines | app/Resources/views/genus/show.html.twig

```
... lines 1 - 4
{% block body %}
... lines 6 - 7
    <div class="sea-creature-container">
... line 9
        <div class="genus-details">
            <dl class="genus-details-list">
... lines 12 - 19
                <dt>Diet</dt>
                <dd>{{ recentlyAte }}</dd>
... lines 22 - 47
            </dl>
        </div>
    </div>
... lines 51 - 92
{% endblock %}
```

Nice! Go back and refresh the show page. There it is! Aurelia recently ate shrimp, clams, lobsters, shark.

Creating an Iterable Object

Now, here's the challenge. Imagine that this feed() function is part of a re-usable library that we're creating, and we want to make it as flexible as possible. Right now, we are *requiring* the $foods argument be an array. But... is that necessary? Really, if you passed me *anything* that I could loop over, we could make this function work.

Let's try it! In GenusController, rename the $food variable to $foodArray. Then, add $food = new \ArrayObject() and pass it $foodArray.

165 lines | src/AppBundle/Controller/GenusController.php

```
... lines 1 - 15
class GenusController extends Controller
{
... lines 18 - 94
    public function showAction(Genus $genus)
    {
... lines 97 - 107
        $foodArray = ['shrimp', 'clams', 'lobsters', 'shark'];
        $foodObject = new \ArrayObject($foodArray);
... lines 110 - 117
    }
... lines 119 - 163
}
```

If you're not familiar with ArrayObject, it's a PHP core object, but it looks and acts like an array. Most importantly, you can foreach over it. So in theory, our feed() function should be able to use this, right?

If you refresh... huge error!

The iterable Pseudo-type

> Argument one passed to feed() must be an array, object given

Of course: we're requiring an array with the type-hint. Well... that's kind of lame. Change this to iterable: a new pseudo-type - like array - from PHP 7.1. This is *perfect* when *all* you care about is that an argument can be used in foreach.

```
233 lines | src/AppBundle/Entity/Genus.php
... lines 1 - 18
class Genus
{
... lines 21 - 223
    public function feed(iterable $food): string
... lines 225 - 231
}
```

Notice, PHPStorm doesn't like this at all. My version of PHPStorm still doesn't think that iterable exists. But, it *is* valid, and this will probably, hopefully be fixed soon.

Of course, if you refresh now, a new error!

> Warning, implode, invalid arguments passed

Our function now allows an array or any iterable object. But... the second argument to implode() *must* be an array. Remember, when you type-hint with iterable, the *only* thing you know is that you can foreach over that value. It's not even legal to use count() like this!

If I want this to be more flexible, we need to do some refactoring. Create a new variable called $foodItems set to an empty array. Then, foreach over $food as $foodItem. This is legal! Inside, put each item into the $foodItems array.

```
239 lines | src/AppBundle/Entity/Genus.php
... lines 1 - 18
class Genus
{
... lines 21 - 223
    public function feed(iterable $food): string
    {
        $foodItems = [];
... line 227
        foreach ($food as $foodItem) {
            $foodItems[] = $foodItem;
        }
... lines 231 - 236
    }
}
```

Finally, update the count to use $foodItems and the same for implode().

```
239 lines | src/AppBundle/Entity/Genus.php
... lines 1 - 18
class Genus
{
... lines 21 - 223
    public function feed(iterable $food): string
    {
... lines 226 - 231
        if (count($foodItems) === 0) {
... line 233
        }
... line 235
        return sprintf('%s recently ate: %s', $this->getName(), implode(', ', $foodItems));
    }
}
```

And just like that, this function can accept *any* value that we can loop over.

Now, I wouldn't necessarily do this in *my* code unless I needed it. If you're always going to pass an array, just type-hint with array! But, you *will* start seeing this more and more in libraries that you use.

Chapter 8: The Multi Exception Catch

Ok, let's talk about *one* more fun feature! And, surprise! This is also new in PHP 7.1.

For this example, let's have some fun. Imagine there's a guy at your office - Crazy Dave who always brings in amazing smelling cookies. But... he *never* shares them. Basically, Crazy Dave is a jerk.

So we're going to write a really annoying function that acts just like Crazy Dave. Open MainController and add a new public function cookiesAction(). Above, add @Route("/crazy-dave").

29 lines | src/AppBundle/Controller/MainController.php

```
... lines 1 - 9
class MainController extends Controller
... lines 11 - 16
    /**
     * @Route("/crazy-dave")
     */
    public function cookiesAction()
    {
... lines 22 - 26
    }
}
```

If you downloaded the code for this project, you should have a tutorial/ directory with an Exception directory inside. Copy that Exception directory into src/AppBundle.

This contains two new classes for the two reasons that Crazy Dave always gives us for *not* sharing his cookies: the NoCookiesLeft exception and the NoCookieForYou exception. That last one is particularly rude.

15 lines | src/AppBundle/Exception/NoCookiesLeft.php

```
... lines 1 - 7
final class NoCookiesLeft extends \Exception
{
    public function __construct($message = 'There are no more cookies :(', $code = 0, Exception $previous = null)
    {
        parent::__construct($message, $code, $previous);
    }
}
```

15 lines | src/AppBundle/Exception/NoCookieForYou.php

```
... lines 1 - 7
final class NoCookieForYou extends \Exception
{
    public function __construct($message = 'No cookie for you!', $code = 0, Exception $previous = null)
    {
        parent::__construct($message, $code, $previous);
    }
}
```

See, Crazy Dave is such a jerk that he denies me reasonable cookie requests with random reasons. So, if random_int(0, 1), then he says throw new NoCookieForYou. Otherwise, he throws a new NoCookiesLeft exception. So disappointing.

```
29 lines | src/AppBundle/Controller/MainController.php
... lines 1 - 9
class MainController extends Controller
{
... lines 12 - 19
    public function cookiesAction()
    {
        if (random_int(0, 1)) {
            throw new NoCookieForYou();
        }

        throw new NoCookiesLeft();
    }
}
```

Find your browser and change the URL to /crazy-dave. Yep, random awful errors. And *no* cookies.

Catching Exceptions the Old Way

Ya know, we don't appreciate Dave shouting or throwing random things at us. So, let's at least ask Dave to whisper his denials. To do that, wrap all of the logic in a try-catch block. We need to catch *both* errors. No problem! First, catch NoCookieForYou. If that happens, set a new $whisper variable to Crazy Dave whispered and then $e->getMessage().

```
38 lines | src/AppBundle/Controller/MainController.php
... lines 1 - 9

class MainController extends Controller
... lines 12 - 19
     */
    public function cookiesAction()
    {
        try {
... lines 24 - 28
        } catch (NoCookieForYou $e) {
            $whisper = sprintf('Crazy Dave whispered "%s"', $e->getMessage());
        } catch (NoCookiesLeft $e) {
            $whisper = sprintf('Crazy Dave whispered "%s"', $e->getMessage());
        }

        return new Response('<html><body>'.$whisper.'</body></html>');
    }
}
```

Here's the problem: the two exception classes do *not* extend a common exception class or interface... except for the *base* Exception class... and I don't want to catch *all* exceptions.

Yep, to responsibly catch both errors, we need to also catch NoCookiesLeft. And that's a bummer, because we need to duplicate the entire $whisper line. But, on the bright side, we can finally return a new Response() with the message inside.

When we refresh, this *does* fix the problem: Dave is now whispering. Still no cookies though... because Dave is still a jerk.

Catching Multiple Exceptions

If you find yourself in this situation, you should just go buy your own cookies. And if you find yourself catching multiple exceptions, there are two solutions. First, if you are able to modify these exceptions, you should make them extend a common base class or implement a common interface. Then, you can catch *that* exception instance. That's the proper solution.

But if you can't update the classes, in PHP 7, you can catch them both at once. Delete the second catch and instead, say

NoCookieForYou | NoCookiesLeft $e. Say hello to the multi-catch syntax.

36 lines | src/AppBundle/Controller/MainController.php

```
... lines 1 - 10
class MainController extends Controller
{
... lines 13 - 20
    public function cookiesAction()
    {
        try {
... lines 24 - 28
        } catch (NoCookieForYou | NoCookiesLeft $e) {
            $whisper = sprintf('Crazy Dave whispered "%s"', $e->getMessage());
        }
... lines 32 - 33
    }
}
```

Thanks to that, I can refresh and it still works.

Alright guys, that is everything I want to show you from PHP 7. Yes yes, there *are* other things, like the spaceship operator... but they're not that important. And other improvements happen automatically. For example, the rand() function is now an alias to mt_rand(), which is a more secure way of generating random numbers. You get that for free. Thanks PHPeople!

And of course, free performance! Deploy it to your production server and watch your New Relic graphs get awesome! After all, isn't seeing performance graphs suddenly improve basically the best thing ever? Well, the best thing after cookies at least?

Alright guys, that's it for me. I'll seeya next time.