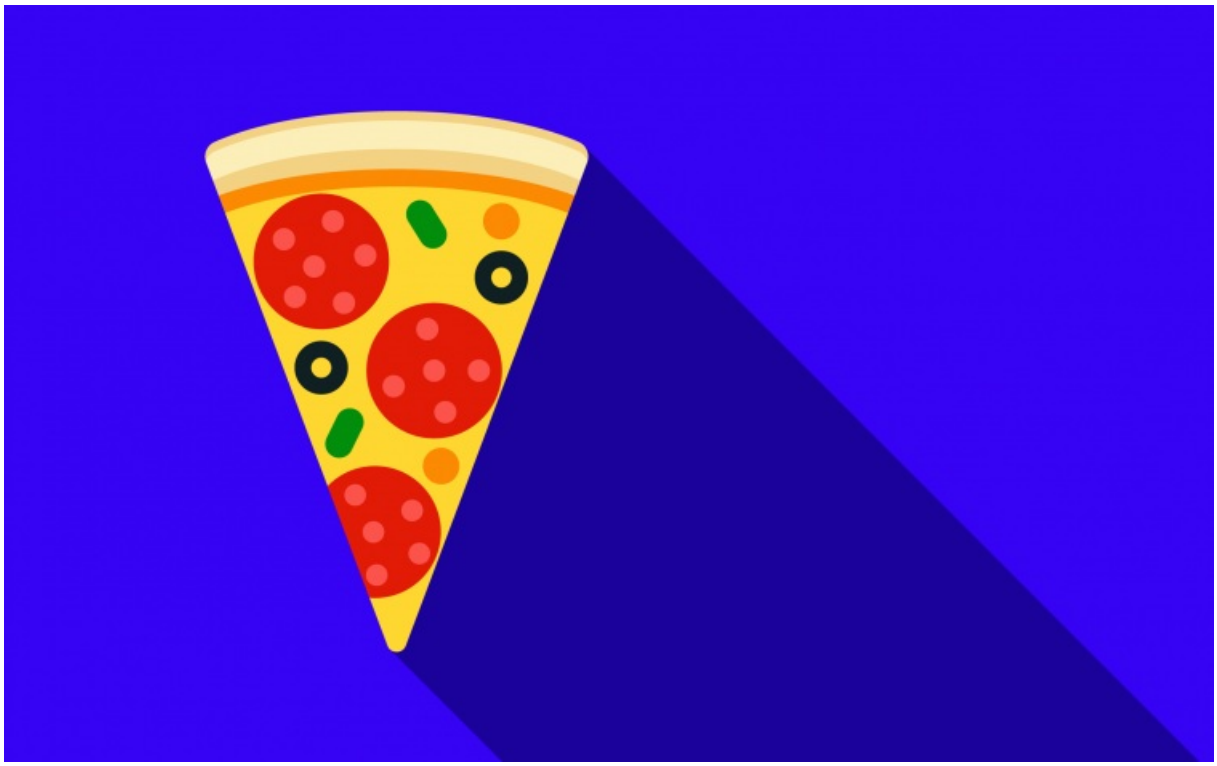
PHP Namespaces in Under 5 Minutes



With <3 from SymfonyCasts

Chapter 1: PHP Namespaces in under 5 Minutes

I've an idea! Let's *master* PHP namespaces... and let's do it in *under* 5 minutes. Sip some coffee... let's go!

Meet Foo

Meet Foo: a *perfectly* boring PHP class:

9 lines | Foo.php

```
... lines 1 - 2
class Foo
{
    public function doAwesomeThings()
    {

    }
}
```

Say hi Foo! Hilarious.

10 lines | Foo.php

```
... lines 1 - 2
class Foo
{
    public function doAwesomeThings()
    {
        echo "Hi Foo!\n";
    }
}
```

To instantiate our favorite new class, I'll move over to a different file and say - drumroll - $foo = new Foo():

6 lines | some-other-file.php

```
... lines 1 - 2
require 'Foo.php';

$foo = new Foo();
```

Tada! We can even call a method on it: $foo->doAwesomeThings():

8 lines | some-other-file.php

```
... lines 1 - 2
require 'Foo.php';

$foo = new Foo();

$foo->doAwesomeThings();
```

Will it work? Of course! I can open a terminal and run:

```
$ php some-other-file.php
```

Namespaces: Making Foo more Hipster

Right now, Foo doesn't have a namespace! To make Foo more hipster, let's fix that. Above the class, add, how about, namespace Acme\Tools:

12 lines | Foo.php

```
... lines 1 - 2
namespace Acme\Tools;

class Foo
{
... lines 7 - 10
}
```

Usually the namespace of a class matches its directory, but that's not *technically* required. I just invented this one!

Using a Namespaced Class

Congratulations! Our friend Foo now lives in a namespace. Putting a class in a namespace is a lot like putting a file in a directory. To reference it, use the full, long path to the class: Acme\Tools\Foo:

8 lines | some-other-file.php

```
... lines 1 - 2
require 'Foo.php';

$foo = new \Acme\Tools\Foo();
... lines 6 - 8
```

just like you can use the absolute path to reference a file in your filesystem:

```
$ ls /acme/tools/foo
```

When we try the script now:

```
$ php some-other-file.php
```

It still works!

The Magical & Optional use Statement

And... that's really! Namespaces are basically a way to... make your class names longer! Add the namespace... then refer to the class using the namespace *plus* the class name. That's it.

But... having these *long* class names right in the middle of your code is a bummer! To fix that, PHP namespaces have *one* more special thing: the use statement. At the top of the file, add use Acme\Tools\Foo as SomeFooClass:

10 lines | some-other-file.php

```
... lines 1 - 2
require 'Foo.php';

use Acme\Tools\Foo as SomeFooClass;
... lines 6 - 10
```

This creates a... sort of... "shortcut". Anywhere else in this file, we can now just type SomeClassFoo:

10 lines | some-other-file.php

```
... lines 1 - 2
require 'Foo.php';

use Acme\Tools\Foo as SomeFooClass;

$foo = new SomeFooClass();
... lines 8 - 10
```

and PHP will know that we're *really* referring to the long class name: Acme\Tools\Foo.

```
$ php some-other-file.php
```

Or... if you leave off the as part, PHP will assume you want this alias to be Foo. That's usually how code looks:

10 lines | some-other-file.php

```
... lines 1 - 2
require 'Foo.php';

use Acme\Tools\Foo;

$foo = new Foo();
... lines 8 - 10
```

So, namespaces make class names longer... and use statements allow us to create shortcuts so we can use the "short" name in our code.

Core PHP Classes

In modern PHP code, pretty much *all* classes you deal with will live in a namespace... except for *core* PHP classes. Yep, core PHP classes do *not* live in a namespace... which kinda means that they live at the "root" namespace - like a file at the root of your filesystem:

```
$ ls /some-root-file
```

Let's play with the core DateTime object: $dt = new DateTime() and then echo $dt->getTimestamp() with a line break:

13 lines | some-other-file.php

```
... lines 1 - 8
$foo->doAwesomeThings();

$dt = new DateTime();
echo $dt->getTimestamp()."\n";
```

When we run the script:

```
$ php some-other-file.php
```

It works perfectly! But... now move that *same* code into the doAwsomeThings method inside our friend Foo:

15 lines | Foo.php

```
... lines 1 - 2
namespace Acme\Tools;

class Foo
{
    public function doAwesomeThings()
    {
        echo "Hi Foo!\n";

        $dt = new DateTime();
        echo $dt->getTimestamp()."\n";
    }
}
```

Now try the code:

```
$ php some-other-file.php
```

Ah! It explodes! And check out that error!

> Class Acme\Tools\DateTime not found

The *real* class name should just be DateTime. So, why does PHP think it's Acme\Tools\DateTime? Because namespaces work like directories! Foo lives in Acme\Tools. When we just say DateTime, it's the same as looking for a DateTime file inside of an Acme/Tools directory:

```
$ cd /acme/tools
$ ls DateTime    # /acme/tools/DateTime
```

There are two ways to fix this. The first is to use the "fully qualified" class name. So, \DateTime:

15 lines | Foo.php

```
... lines 1 - 2
namespace Acme\Tools;

class Foo
{
    public function doAwesomeThings()
    {
... lines 9 - 10
        $dt = new \DateTime();
... line 12
    }
}
```

Yep... that works *just* like a filesystem.

```
$ php some-other-file.php
```

Or... you can *use* DateTime... then remove the \ below:

17 lines | Foo.php

```
... lines 1 - 2
namespace Acme\Tools;

use DateTime;

class Foo
{
    public function doAwesomeThings()
    {
... lines 11 - 12
        $dt = new DateTime();
... line 14
    }
}
```

That's really the same thing: there's no \ at the beginning of a use statement, but you should pretend there is. This aliases DateTime to \DateTime.

And... we're done! Namespaces make your class names longer, use statements allow you to create "shortcuts" so you can use short names in your code and the *whole* system works *exactly* like files inside directories.

Have fun!