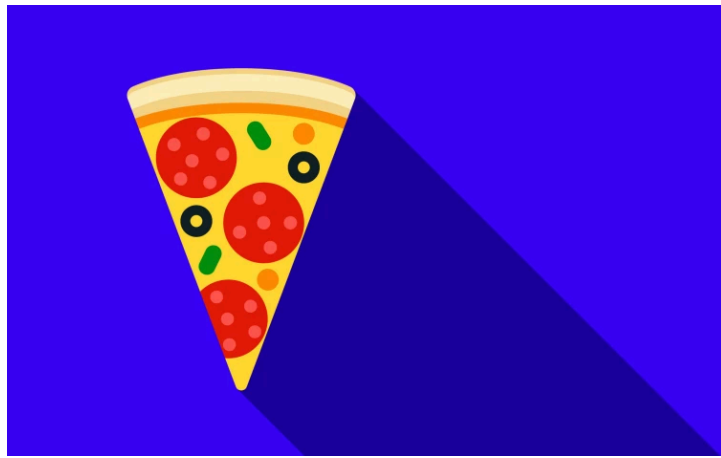


# PHP Namespaces in Under 5 Minutes



# 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:

```
Foo.php
↕ // ... lines 1 - 2
3 class Foo
4 {
5     public function doAwesomeThings()
6     {
7
8     }
9 }
```

Say hi `Foo`! Hilarious.

```
Foo.php
↕ // ... lines 1 - 2
3 class Foo
4 {
5     public function doAwesomeThings()
6     {
7         echo "Hi Foo!\n";
8     }
9 }
```

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

```
some-other-file.php
↕ // ... lines 1 - 2
3 require 'Foo.php';
4
5 $foo = new Foo();
```

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

```
some-other-file.php
↕ // ... lines 1 - 2
3  require 'Foo.php';
4
5  $foo = new Foo();
6
7  $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`:

```
Foo.php
↕ // ... lines 1 - 2
3  namespace Acme\Tools;
4
5  class Foo
6  {
↕ // ... lines 7 - 10
11 }
```

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:
```

```
some-other-file.php
↕ // ... lines 1 - 2
3 require 'Foo.php';
4
5 $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`:

```
some-other-file.php
↕ // ... lines 1 - 2
3 require 'Foo.php';
4
5 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`:

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

```
↕ // ... lines 1 - 2
3 require 'Foo.php';
4
5 use Acme\Tools\Foo as SomeFooClass;
6
7 $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:

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

```
↕ // ... lines 1 - 2
3 require 'Foo.php';
4
5 use Acme\Tools\Foo;
6
7 $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:

```
some-other-file.php
↕ // ... lines 1 - 8
9  $foo->doAwesomeThings();
10
11  $dt = new DateTime();
12  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 `doAwesomeThings` method inside our friend `Foo`:

```
Foo.php
↕ // ... lines 1 - 2
3  namespace Acme\Tools;
4
5  class Foo
6  {
7      public function doAwesomeThings()
8      {
9          echo "Hi Foo!\n";
10
11         $dt = new DateTime();
12         echo $dt->getTimestamp()."\n";
13     }
14 }
```

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`:

```
Foo.php
↕ // ... lines 1 - 2
3 namespace Acme\Tools;
4
5 class Foo
6 {
7     public function doAwesomeThings()
8     {
↕ // ... lines 9 - 10
11         $dt = new \DateTime();
↕ // ... line 12
13     }
14 }
```

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

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

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

Foo.php

```
↕ // ... lines 1 - 2
3 namespace Acme\Tools;
4
5 use DateTime;
6
7 class Foo
8 {
9     public function doAwesomeThings()
10    {
↕ // ... lines 11 - 12
13         $dt = new DateTime();
↕ // ... line 14
15     }
16 }
```

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!



*With <3 from SymphonyCasts*