



The PHP Company

Using PHP 5.3 Namespaces for Fame and Fortune

Matthew Weier O'Phinney
Project Lead, Zend Framework

What are “namespaces”?

What are “namespaces”?

"A way in which to group related classes, functions and constants"
- <http://php.net/namespace>

Basics

Namespace Separator

“\”

Get over it!

What can live in namespaces

- Classes
- Constants
- Functions

Declaring namespaces

- Single line declaration

```
namespace Zend;
```

Declaring namespaces

- Block declaration

```
namespace Zend
{
}
namespace Phly
{
}
```

Subnamespaces

- Separate the subnamespaces using the namespace separator

```
namespace Zend\Log;  
namespace Zend\Log\Writer;
```

Referring to namespaced elements

- From non-namespaced code:

```
$class = new My\Registry();  
$class = new My\Log\Logger();  
My\str_split($s); // function call  
$val = My\APIKEY; // constant value
```

Referring to namespaced elements

- From code using the same namespace:

```
namespace My;  
  
$class = new Registry();  
$class = new Log\Logger();  
str_split($s); // function call  
$val = APIKEY; // constant value
```

Referring to namespaced elements

- From code in a different namespace:

```
namespace Your;  
  
$class = new \My\Registry();  
$class = new \My\Log\Logger();  
\My\str_split($s); // function call  
$val = \My\APIKEY; // constant value
```

Using namespaced code:

- Prefixing with a namespace separator resolves as a fully qualified name
- Resolution order:
 - ▶ Globally qualified: known; otherwise:
 - ▶ Current namespace, or relative to it (Classes, functions, constants)
 - ▶ Global namespace (functions, constants)

This means:

- You can override system functions within namespaces!

```
namespace My;

function str_split($string, $splitlen = 2)
{
    return preg_split('/\W/', $string, $splitlen);
}

$a = str_split("Foo, Bar"); // invokes My\str_split()
/* array (
 *     0 => 'Foo',
 *     1 => ' Bar',
 * )
 */
```

Importing namespaces

- A way to bring code from another namespace into the current namespace.
- Import:
 - ▶ Classes
 - ▶ Other namespaces
- Keyword used is *use*

Importing namespaces

```
namespace My;
class Registry {}

namespace Your;
use My;           // imports namespace
use My\Registry; // imports class
```

Using imported code

```
namespace Your;  
use My;          // imports namespace  
$r = new My\Registry();
```

```
namespace Your;  
use My\Registry; // imports class  
$r = new Registry();
```

Aliasing

- *"import namespace or class, but refer to it using this name"*
- PHP utilizes the *as* keyword to alias

Aliasing

```
namespace Your;  
use My as m;  
$r = new m\Registry();
```

```
namespace Your;  
use My\Registry as reg;  
$r = new reg();
```

Make typehinting more semantic!

```
namespace App;
use Zend\EventManager\EventManager as Events;

class Foo
{
    public function events(Events $events = null)
    {
        if ($events instanceof Events) {
            $this->events = $events;
        } elseif (!$this->events instanceof Events) {
            $this->events = new Events(__CLASS__);
        }
        return $this->events;
    }
}
```

Importing multiple namespaces/classes

- Multiple *use* statements

```
namespace Their;  
use My;  
use Your;
```

Importing multiple namespaces/classes

- Or use a comma (“,”) to separate statements

```
namespace Their;  
use My,  
    Your;
```

Importing multiple namespaces/classes

- With aliases

```
namespace Their;  
  
use My as m, // aliased  
    Your;      // not aliased
```

What namespace are you in?

- **`__NAMESPACE__`**

Comprehensive example

- The setup:

```
namespace My\Package;
const APIKEY = 1;
function get($data) {}
class Service {}

namespace My\OtherPackage;
const APIKEY = 2;
function get($data) {}
class Service {}
```

Comprehensive example

```
namespace Test;
use My\Package\Service as PackageService,
    My\OtherPackage;

const APIKEY = 3;
echo APIKEY; // 3
echo OtherPackage\APIKEY; // 2
echo \My\Package\APIKEY; // 1

$s = new PackageService(); // My\Package\Service
$s = new Service(); // E_FATAL (no match)
$s = new OtherPackage\Service();
// My\OtherPackage\Service

get($baz); // E_FATAL (no matching function in
// current namespace)
OtherPackage\get($baz); // My\OtherPackage\get()
```

Pitfalls

Referencing namespaced code in strings

- Classes, constants, and functions referenced in a string ***MUST*** be fully qualified
- No namespace separator *prefix* is necessary

Referencing namespaced code in strings

- Example 1

```
namespace My;
// Assume My\Log\Logger is a defined class

$class = 'Log\Logger';
$o      = new $class; // E_FATAL; can't find
                        // relative names

$class = 'My\Log\Logger';
$o      = new $class; // Success

$class = '\My\Log\Logger';
$o      = new $class; // Also success,
                        // but not necessary
```

Referencing namespaced code in strings

- Example 2

```
namespace My;
// Assume My\Log\Logger is a defined class

$class = 'Log\Logger';
if ($o instanceof $class) { } // Fails; can't
                               // resolve class

$class = 'My\Log\Logger';
if ($o instanceof $class) { } // Success
```

Why use namespaces?

Code Organization: Filesystem

- Namespace separator has an affinity for the directory separator
- Suggests a 1:1 relationship with file system

```
namespace My\Log;  
class Logger {}  
/* My/Log/Logger.php (*nix)  
   My\Log\Logger.php (Windows) */
```

Code Organization: By Responsibility

- Interfaces

- ▶ Use cases:

- `instanceof: $class instanceof Adapter`
 - `implements: SomeClass implements Adapter`

- ▶ Natural language is easiest to understand

- ▶ Natural language suggests a hierarchy

Code Organization: By Responsibility

```
namespace Translator\Adapter;  
use Translator\Adapter;  
  
class ConcreteAdapter implements Adapter  
{ }
```

- **interface** Translator\Adapter **in** Translator/Adapter.php
- **class** Translator\Adapter\ConcreteAdapter **in** Translator/Adapter/ConcreteAdapter.php

Code Organization: By Responsibility

- Takeaway: we're referencing the *capabilities*, not the *language type*

Readability

- When within a namespace, reference classes directly, keeping usage succinct

```
namespace Zend\Http;  
$client = new Client();
```

Readability

- Importing and aliasing make names more succinct

```
namespace Application;
use Zend\Http\Client as HttpClient,
    Zend\Logger\Logger;

$client = new HttpClient();
$log    = new Logger();
```

Readability

- Aliasing allows giving names context

```
namespace Application\Controller;  
  
use Zend\Controller\Action  
    as ActionController;  
  
class FooController  
    extends ActionController {}
```

Dependency declarations

- **Suggestion:** import *every* class outside the current namespace that you consume

```
namespace Application\Controller;  
  
use Zend\Controller\Action  
    as ActionController,  
    My\ORM\Mapper as DataMapper,  
    My\Entity\BlogEntry,  
    Zend\EventManager\EventManager;
```

Dependency declarations

- Imports are hints to the interpreter, and don't cost anything
- Helps to document dependencies up front
- Allows static analysis to determine dependencies

Resources

-
- PHP Manual: <http://php.net/namespaces>

Basically, a language-supported mechanism for grouping these language features, as well as a language-supported mechanism for avoiding naming conflicts.

Namespace Separator

“\”

Get over it!

© All rights reserved. Zend Technologies, Inc.



There are a ton of reasons why the backslash was used, ranging from length of token to position of the token on most keyboards to previous use of other tokens to understanding the scope of a given operator/operation visually. Just use it, and move along.

What can live in namespaces

- Classes
- Constants
- Functions

Declaring namespaces

- Single line declaration

```
namespace Zend;
```

You can declare multiple namespaces in the same file in this way, but it's not recommended

Declaring namespaces

- Block declaration

```
namespace Zend
{
}
namespace Phly
{
}
```

This is the recommended way to declare multiple namespaces when in a single file.

Subnamespaces

- Separate the subnamespaces using the namespace separator

```
namespace Zend\Log;  
namespace Zend\Log\Writer;
```

Referring to namespaced elements

- From non-namespaced code:

```
$class = new My\Registry();  
$class = new My\Log\Logger();  
My\str_split($s); // function call  
$val = My\APIKEY; // constant value
```

Referring to namespaced elements

- From code using the same namespace:

```
namespace My;  
  
$class = new Registry();  
$class = new Log\Logger();  
str_split($s); // function call  
$val = APIKEY; // constant value
```

Referring to namespaced elements

- From code in a different namespace:

```
namespace Your;  
  
$class = new \My\Registry();  
$class = new \My\Log\Logger();  
\My\str_split($s); // function call  
$val = \My\APIKEY; // constant value
```

Using namespaced code:

- Prefixing with a namespace separator resolves as a fully qualified name
- Resolution order:
 - ▶ Globally qualified: known; otherwise:
 - ▶ Current namespace, or relative to it (Classes, functions, constants)
 - ▶ Global namespace (functions, constants)

classes outside the current namespace must be referenced using a fully (globally) qualified name, or `_imported_`.

This means:

- You can override system functions within namespaces!

```
namespace My;

function str_split($string, $splitlen = 2)
{
    return preg_split('/\W/', $string, $splitlen);
}

$a = str_split("Foo, Bar"); // invokes My\str_split()
/* array (
 *   0 => 'Foo',
 *   1 => ' Bar',
 * )
 */
```

Importing namespaces

- A way to bring code from another namespace into the current namespace.
- Import:
 - ▶ Classes
 - ▶ Other namespaces
- Keyword used is *use*

Importing namespaces

```
namespace My;
class Registry {}

namespace Your;
use My;           // imports namespace
use My\Registry; // imports class
```

Note: importing top-level namespaces in code with no namespace has no effect and results in a warning.

Using imported code

```
namespace Your;
use My; // imports namespace
$r = new My\Registry();

namespace Your;
use My\Registry; // imports class
$r = new Registry();
```

Aliasing

- *"import namespace or class, but refer to it using this name"*
- PHP utilizes the *as* keyword to alias

Aliasing

```
namespace Your;  
use My as m;  
$r = new m\Registry();  
  
namespace Your;  
use My\Registry as reg;  
$r = new reg();
```

Make typehinting more semantic!

```
namespace App;
use Zend\EventManager\EventManager as Events;

class Foo
{
    public function events(Events $events = null)
    {
        if ($events instanceof Events) {
            $this->events = $events;
        } elseif (!$this->events instanceof Events) {
            $this->events = new Events(__CLASS__);
        }
        return $this->events;
    }
}
```

Importing multiple namespaces/classes

- Multiple *use* statements

```
namespace Their;  
use My;  
use Your;
```

Importing multiple namespaces/classes

- Or use a comma (“,”) to separate statements

```
namespace Their;  
use My,  
    Your;
```

Importing multiple namespaces/classes

- With aliases

```
namespace Their;  
  
use My as m, // aliased  
    Your;    // not aliased
```

What namespace are you in?

- `__NAMESPACE__`

Comprehensive example

- The setup:

```
namespace My\Package;
const APIKEY = 1;
function get($data) {}
class Service {}

namespace My\OtherPackage;
const APIKEY = 2;
function get($data) {}
class Service {}
```

Comprehensive example

```
namespace Test;
use My\Package\Service as PackageService,
    My\OtherPackage;

const APIKEY = 3;
echo APIKEY; // 3
echo OtherPackage\APIKEY; // 2
echo \My\Package\APIKEY; // 1

$s = new PackageService(); // My\Package\Service
$s = new Service(); // E_FATAL (no match)
$s = new OtherPackage\Service();
// My\OtherPackage\Service

get($baz); // E_FATAL (no matching function in
// current namespace)
OtherPackage\get($baz); // My\OtherPackage\get()
```


Referencing namespaced code in strings

- Classes, constants, and functions referenced in a string ***MUST*** be fully qualified
- No namespace separator *prefix* is necessary

Referencing namespaced code in strings

- Example 1

```
namespace My;
// Assume My\Log\Logger is a defined class

$class = 'Log\Logger';
$o      = new $class; // E_FATAL; can't find
                // relative names

$class = 'My\Log\Logger';
$o      = new $class; // Success

$class = '\My\Log\Logger';
$o      = new $class; // Also success,
                // but not necessary
```

Referencing namespaced code in strings

- Example 2

```
namespace My;
// Assume My\Log\Logger is a defined class

$class = 'Log\Logger';
if ($o instanceof $class) { } // Fails; can't
                                // resolve class

$class = 'My\Log\Logger';
if ($o instanceof $class) { } // Success
```


Code Organization: Filesystem

- Namespace separator has an affinity for the directory separator
- Suggests a 1:1 relationship with file system

```
namespace My\Log;  
class Logger {}  
/* My/Log/Logger.php (*nix)  
   My\Log\Logger.php (Windows) */
```

Code Organization: By Responsibility

- Interfaces
 - ▶ Use cases:
 - `instanceof: $class instanceof Adapter`
 - `implements: SomeClass implements Adapter`
 - ▶ Natural language is easiest to understand
 - ▶ Natural language suggests a hierarchy

Code Organization: By Responsibility

```
namespace Translator\Adapter;  
use Translator\Adapter;  
  
class ConcreteAdapter implements Adapter  
{  
}
```

- **interface** `Translator\Adapter` in `Translator/Adapter.php`
- **class** `Translator\Adapter\ConcreteAdapter` in `Translator/Adapter/ConcreteAdapter.php`

Code Organization: By Responsibility

- Takeaway: we're referencing the *capabilities*, not the *language type*

Readability

- When within a namespace, reference classes directly, keeping usage succinct

```
namespace Zend\Http;  
$client = new Client();
```

Readability

- Importing and aliasing make names more succinct

```
namespace Application;
use Zend\Http\Client as HttpClient,
    Zend\Logger\Logger;

$client = new HttpClient();
$log    = new Logger();
```

Readability

- Aliasing allows giving names context

```
namespace Application\Controller;  
  
use Zend\Controller\Action  
    as ActionController;  
  
class FooController  
    extends ActionController {}
```

Dependency declarations

- **Suggestion:** import *every* class outside the current namespace that you consume

```
namespace Application\Controller;  
  
use Zend\Controller\Action  
    as ActionController,  
    My\ORM\Mapper as DataMapper,  
    My\Entity\BlogEntry,  
    Zend\EventManager\EventManager;
```

Dependency declarations

- Imports are hints to the interpreter, and don't cost anything
- Helps to document dependencies up front
- Allows static analysis to determine dependencies

Click to add title

- PHP Manual: <http://php.net/namespace>