

Anonymous Functions in PHP 5.3

Matthew Weier O'Phinney

26 April 2011

But first, some vocabulary

Lambdas

- From “**lambda** calculus”

Lambdas

- From “**lambda** calculus”
- Names of functions are merely a “convenience”, and hence all functions are considered anonymous

Lambdas

- From “**lambda** calculus”
- Names of functions are merely a “convenience”, and hence all functions are considered anonymous
- Unless you’re using a true functional programming language with first-class functions, most likely you’re simply using anonymous functions, or *lambda expressions*.

Closures

- A function which includes a referencing environment

Closures

- A function which includes a referencing environment
 - Each function has its own scope
 - Allows hiding *state*

Closures

- A function which includes a referencing environment
 - Each function has its own scope
 - Allows hiding *state*
- Differentiating factor is that closures allow binding references that exist at the time of creation, to use when called.

Anonymous Functions

- Any function defined and/or called without being bound to an identifier.
 - You can assign the function to a variable, but you're not giving it its own name.

PHP's Offering

Functors

- Any object defining a `__invoke()` method.
- Object instances can now be called as if they were functions.

```
class Command
{
    public function __invoke($name)
    {
        echo "Hello, $name";
    }
}
$c = new Command();
$c('Matthew'); // "Hello, Matthew"
```

Anonymous Functions

- `new Closure` class, which defines `__invoke()`.

Anonymous Functions

- new Closure class, which defines `__invoke()`.
- `__invoke()` body is “replaced” with the defined function.

Anonymous Functions

- new `Closure` class, which defines `__invoke()`.
- `__invoke()` body is “replaced” with the defined function.
- Ability to bind variables via *imports*, allowing creation of *closures*.

Important to know:

- Just like normal PHP functions, anonymous functions exist in their own scope.

Important to know:

- Just like normal PHP functions, anonymous functions exist in their own scope.
- You cannot import `$this`.

Important to know:

- Just like normal PHP functions, anonymous functions exist in their own scope.
- You cannot import `$this`.
- You cannot alias imported variables.

Anonymous Function and Closure Syntax

Basics

- Simply like normal function declaration, except no name:

```
function($value1[, $value2[, ... $valueN]]) { };
```

Assign to variables

- Assign functions to variables; don't forget the semicolon terminator!

```
$greet = function($name) {  
    echo "Hello, $name";  
};  
  
$greet('Matthew'); // "Hello, Matthew"
```

Pass as arguments to other callables

- Allow other functionality to call the function.

```
function say($value, $callback)
{
    echo $callback($value);
}

say('Matthew', function($name) {
    return "Hello, $name";
}); // "Hello, Matthew"
```

Create closures

- Bind variables at creation, and use them at call-time.

```
$log = Zend_Log::factory($config);

$logger = function() use($log) {
    $args = func_get_args();
    $log->info(json_encode($args));
};

$logger('foo', 'bar'); // ["foo", "bar"]
```

Things to try

Array operations

- **Sorting** (`usort`, `uasort`, **etc.**)
- **Walking, mapping, reducing**
- **Filtering**

Sorting

```
$stuff = array('apple', 'Anise', 'Applesauce', 'appleseed');  
usort($stuff, function($a, $b) {  
    return strcasecmp($a, $b);  
});  
// 'Anise', 'apple', 'Applesauce', 'appleseed'
```

Walking

- *Walking* allows you to change the values of an array.

Walking

- *Walking* allows you to change the values of an array.
- If not using objects, then you need to pass by reference in order to alter values.

```
$stuff = array('apple', 'Anise', 'Applesauce', 'appleseed');  
array_walk($stuff, function(&$value) {  
    $value = strtoupper($value);  
});  
// 'APPLE', 'ANISE', 'APPLESAUCE', 'APPLESEED'
```

Mapping

- *Mapping* performs work on each element, resulting in a new array with the values returned.

Mapping

- *Mapping* performs work on each element, resulting in a new array with the values returned.

```
$stuff = array('apple', 'Anise', 'Applesauce', 'appleseed');
$mapped = array_map(function($value) {
    $value = strtoupper($value);
    return $value;
}, $stuff);
// $stuff: array('apple', 'Anise', 'Applesauce', 'appleseed')
// $mapped: array('APPLE', 'ANISE', 'APPLESAUCE', 'APPLESEED')
```

Reducing

- “Combine” elements and return a value or data set.

Reducing

- “Combine” elements and return a value or data set.
- Return value is passed as first argument of next call.

Reducing

- “Combine” elements and return a value or data set.
- Return value is passed as first argument of next call.
- Seed the return value by passing a third argument to `array_reduce()`.

```
$stuff = array('apple', 'Anise', 'Applesauce', 'appleseed');
$reduce = array_reduce($stuff, function($count, $input) {
    $count += substr_count($input, 'a');
    return $count;
}, 0);
// $stuff:  array('apple', 'Anise', 'Applesauce', 'appleseed')
// $reduce: 3
```

Filtering

- Return only the elements that evaluate to true.

Filtering

- Return only the elements that evaluate to true.
- Often, this is a form of *mapping*, and used to trim a dataset to only those of interest prior to reducing.

```
$stuff = array('apple', 'Anise', 'Applesauce', 'appleseed');
$reduce = array_reduce($stuff, function($count, $input) {
    $count += substr_count($input, 'a');
    return $count;
}, 0);
// $stuff: array('apple', 'Anise', 'Applesauce', 'appleseed')
// $reduce: 3
```

String operations

- Regular expressions (`preg_replace_callback`)

String operations

- Regular expressions (`preg_replace_callback`)
- Currying arguments

preg_replace_callback()

- Allows you to transform captured matches.

```
$string = "Today's date next month is " . date('Y-m-d');  
$fixed  = preg_replace_callback('/(\d{4}-\d{2}-\d{2})/',  
function($matches) {  
    $date = new DateTime($matches[1]);  
    $date->add(new DateInterval('P1M'));  
    return $date->format('Y-m-d');  
}, $string);  
// "Today's date next month is 2011-05-26"
```

Currying arguments

- In some cases, you may want to provide default arguments:

Currying arguments

- In some cases, you may want to provide default arguments:
 - To reduce the number of arguments needed.

Currying arguments

- In some cases, you may want to provide default arguments:
 - To reduce the number of arguments needed.
 - To supply values for optional arguments.

Currying arguments

- In some cases, you may want to provide default arguments:
 - To reduce the number of arguments needed.
 - To supply values for optional arguments.
 - To provide a unified signature for callbacks.

Currying arguments

- In some cases, you may want to provide default arguments:
 - To reduce the number of arguments needed.
 - To supply values for optional arguments.
 - To provide a unified signature for callbacks.

```
$hs = function ($value) {  
    return htmlspecialchars($value, ENT_QUOTES, "UTF-8", false);  
};  
$filtered = $hs("<span>Matthew Weier O'Phinney</span>");  
// "&lt;span&gt;Matthew Weier O&#039;Phinney&lt;/span&gt;"
```

Gotchas

References

- Variables passed to callbacks, either as arguments or imports, are not passed by reference.

References

- Variables passed to callbacks, either as arguments or imports, are not passed by reference.
 - Use objects, or

References

- Variables passed to callbacks, either as arguments or imports, are not passed by reference.
 - Use objects, or
 - Pass by reference

References

- Variables passed to callbacks, either as arguments or imports, are not passed by reference.
 - Use objects, or
 - Pass by reference

```
$count    = 0;
$counter  = function (&$value) use (&$count) {
    if (is_int($value)) {
        $count += $value;
        $value  = 0;
    }
};
$stuff = array('foo', 1, 3, 'bar');
array_walk($stuff, $counter);
// $stuff: array('foo', 0, 0, 'bar')
// $count: 4
```

Mixing with other callables

Problems and considerations:

- `Closure` is an “implementation detail”; typehinting on it excludes other callback types.

Mixing with other callables

Problems and considerations:

- `Closure` is an “implementation detail”; typehinting on it excludes other callback types.
- `is_callable()` tells us only that it can be called, not how.

Mixing with other callables

Problems and considerations:

- `Closure` is an “implementation detail”; typehinting on it excludes other callback types.
- `is_callable()` tells us only that it can be called, not how.
- `is_callable() && is_object()` tells us we have a functor, but omits other callback types.

Mixing with other callables

Three ways to call (1/3):

- `$o($arg1, $arg2)`

Mixing with other callables

Three ways to call (1/3):

- `$o($arg1, $arg2)`
 - Benefits: speed.

Mixing with other callables

Three ways to call (1/3):

- `$o($arg1, $arg2)`
 - Benefits: speed.
 - Problems: won't work unless we have a functor, closure, or static method call.

Mixing with other callables

Three ways to call (2/3):

- `call_user_func($o, $arg1, $arg2)`

Mixing with other callables

Three ways to call (2/3):

- `call_user_func($o, $arg1, $arg2)`
 - Benefits: speed, works with all callables.

Mixing with other callables

Three ways to call (2/3):

- `call_user_func($o, $arg1, $arg2)`
 - Benefits: speed, works with all callables.
 - Problems: if number of arguments are unknown until runtime, this gets difficult.

Mixing with other callables

Three ways to call (3/3):

- `call_user_func_array($o, $argv)`

Mixing with other callables

Three ways to call (3/3):

- `call_user_func_array($o, $argv)`
 - Benefits: works with all callables., works with variable argument counts.

Mixing with other callables

Three ways to call (3/3):

- `call_user_func_array($o, $argv)`
 - Benefits: works with all callables., works with variable argument counts.
 - Problems: speed (takes up to 6x longer to execute than straight call).

You cannot import `$this`

- Creative developers will want to use closures to monkey-patch objects.

You cannot import `$this`

- Creative developers will want to use closures to monkey-patch objects.
- You can. You just can't use `$this`, which means you're limited to public methods.

You cannot import `$this`

- Creative developers will want to use closures to monkey-patch objects.
- You can. You just can't use `$this`, which means you're limited to public methods.
- Also, you can't auto-dereference closures assigned to properties.

Example: Monkey-Patching

```
class Foo
{
    public function __construct()
    {
        $self = $this;
        $this->bar = function () use ($self) {
            return get_object_vars($self);
        };
    }

    public function addMethod($name, Closure $c)
    {
        $this->$name = $c;
    }

    public function __call($method, $args)
    {
        if (property_exists($this, $method) &&
            is_callable($this->$method)) {
            return call_user_func_array($this->$method, $args);
        }
    }
}
```

Serialization

- You can't serialize anonymous functions.

Some Use Cases

Aspect Oriented Programming

- Code defines “aspects,” or code that cuts across boundaries of many components.

Aspect Oriented Programming

- Code defines “aspects,” or code that cuts across boundaries of many components.
- AOP formalizes a way to *join* aspects to other code.

Aspect Oriented Programming

- Code defines “aspects,” or code that cuts across boundaries of many components.
- AOP formalizes a way to *join* aspects to other code.
- Often, you will need to *curry* arguments in order to maintain signatures.

Event Management example

```
$front->events()->attach('dispatch.router.post', function($e) use
($cache) {
    $request = $e->getParam('request');
    if (!$request instanceof Zend\Http\Request || !$request->isGet())
    {
        return;
    }

    $metadata = json_encode($request->getMetadata());
    $key       = hash('md5', $metadata);
    $backend   = $cache->getCache('default');
    if (false !== ($content = $backend->load($key))) {
        $response = $e->getParam('response');
        $response->setContent($content);
        return $response;
    }
    return;
});
```

That's all, folks!

References

- PHP Manual:

`http://php.net/functions.anonymous`

Thank You!