Method Chaining with PHP

Image
by

Colin O'Dell

on
December 19, 2009

Method Chaining with PHP

Method chaining is a fluent interface design pattern used to simplyify your code. If you've used frameworks like Zend or jQuery, you probably have some experience with chaining. Essentially your objects return themselves, allowing you to "chain" multiple actions together.

Before I dive into the DIY aspects, here's an example of some cluttered code we can simplify with chaining. Let's say we want to fetch some data using Zend_Db. Assume we have a company database and want to know how many hours our employees this month. We *could* use a SQL query like this:

```
SELECT E.LastName, E.FirstName, E.Department, SUM(H.HoursLogged) AS
HoursThisMonth FROM Employees AS E INNER JOIN Hours AS H ON H.EmpID = E.EmpID
WHERE H.DateLogged >= '12/1/2009' ORDER BY LastName, FirstName
```

This is perfectly acceptable to use, but hard to read. Using Zend_Db_Select to build the query, we could write it like this:

```
$startDate = '12/1/2009';
$db = Zend_Db::factory( /*options*/ );
$select = $db->select();
$select->from(
array('E' => 'Employees'),
array('LastName, FirstName, Department')
);
$select->joinInnerUsing(
array('H' => 'Hours'),
'EmpID',
array('HoursThisMonth' => 'SUM(HoursLogged)')
);
$select->where('H.DateLogged > ?', $startDate);
$select->order(array('LastName', 'FirstName'));
```

What Chaining Looks Like

That certainly looks better, but see how every line starts with "$select->"? We can eliminate that repetition and clean up the code through chaining. Since Zend_Db_Select already implements chaining, we can do this:

```
$startDate = '12/1/2009';
$db = Zend_Db::factory( /*options*/ );
$select = $db
->select()
->from(
array('E' => 'Employees'),
array('LastName, FirstName, Department')
)
->joinInnerUsing(
array('H' => 'Hours'),
'EmpID',
array('HoursThisMonth' => 'SUM(HoursLogged)')
)
->where('H.DateLogged > ?', $startDate)
->order(array('LastName', 'FirstName'))
;
```

The whole query is built with a single line of code! This sample is much easier to read and maintain.

When Do I Implement Chaining?

Method chaining is best implemented on functions/methods that modify their parent object. They usually set a variable or perform some action without returning a result. Here are some examples of good candidates for chaining:

```
$log->write("Something just happened.");
$log->write("And it happened again.");
$mail->to('myself@domain.com');
$mail->subject('Test');
$mail->body('Hello World!');
$mail->send();
$cache->clear();
$cache->add("somekey", "andavalue");
```

How To Implement It

Implementation is extremely simple - simply return the object instance once the method has completed:

```
function someMethod (params)
{
/* do stuff */
return $this;
}
```

That's all you need to do! Since some of the original methods didn't return a value, returning "$this" shouldn't affect your scripts at all. Now your code looks like this:

```
$log->write("Something just happened.")
->write("And it happened again.")
;
$mail->to('myself@domain.com')
->subject('Test')
```

```
->body('Hello World!')
->send()
;
$cache->clear()
->add("somekey", "andavalue")
;
```

Not all your methods need to (or should) chain. For example, the "$mail->send()" function above should probably return false if the message wasn't sent:

```
if($mail->to('myself@domain.com')->subject('Test')->body('Hello World!')-
>send())
echo "Message Sent!";
else
echo "Oops, something went wrong!";
```

You certainly don't have to use chaining in everything you do, but I'm sure you'll find some good uses for it.

https://www.unleashed-technologies.com/blog/method-chaining-php

```
<?php
class fakeString
{
    private $str;
    function __construct()
    {
        $this->str = "";
    }

    function addA()
    {
        $this->str .= "a";
        return $this;
    }

    function addB()
    {
        $this->str .= "b";
        return $this;
    }

    function getStr()
    {
        return $this->str;
    }
}


$a = new fakeString();


echo $a->addA()->addB()->getStr();
```

- This is also sometimes referred to as Fluent Interface

– Nithesh Chandra
Sep 16 '10 at 6:31

- 19

@Nitesh that is incorrect. Fluent Interfaces use Method Chaining as their primary mechanism, but it's not the same. Method chaining simply returns the host object, while a Fluent Interface is aimed at creating a DSL. Ex: `$foo->setBar(1)->setBaz(2)` vs `$table->select()->from('foo')->where('bar = 1')->order('ASC)`. The latter spans multiple objects.

– Gordon
Sep 16 '10 at 7:32

- 3

public function __toString() { return $this->str; } This will not require the last method "getStr()" if you're echoing out the chain already.

– tfont
Apr 22 '14 at 7:01

- 9

@tfont True, but then we're introducing magic methods. One concept at a time should be sufficient.

– Kristoffer Sall-Storgaard
Apr 22 '14 at 8:49

- 4

Since PHP 5.4 it's even possible to *everything* in one line: `$a = (new fakeString())->addA()->addB()->getStr();`
– Philzen
Jun 1 '15 at 18:49

Basically, you take an object:

```
$obj = new ObjectWithChainableMethods();
```

Call a method that effectively does a `return $this;` at the end:

```
$obj->doSomething();
```

Since it returns the same object, or rather, a *reference* to the same object, you can continue calling methods of the same class off the return value, like so:

```
$obj->doSomething()->doSomethingElse();
```

That's it, really. Two important things:

1. As you note, it's PHP 5 only. It won't work properly in PHP 4 because it returns objects by value and that means you're calling methods on different copies of an object, which would break your code.
2. Again, you need to return the object in your chainable methods:

   ```
   public function doSomething() {
       // Do stuff
       return $this;
   }

   public function doSomethingElse() {
       // Do more stuff
       return $this;
   }
   ```

Try this code:

```
<?php
class DBManager
{
    private $selectables = array();
    private $table;
    private $whereClause;
    private $limit;

    public function select() {
        $this->selectables = func_get_args();
        return $this;
    }

    public function from($table) {
        $this->table = $table;
        return $this;
    }

    public function where($where) {
        $this->whereClause = $where;
```

```
        return $this;
    }

    public function limit($limit) {
        $this->limit = $limit;
        return $this;
    }

    public function result() {
        $query[] = "SELECT";
        // if the selectables array is empty, select all
        if (empty($this->selectables)) {
            $query[] = "*";
        }
        // else select according to selectables
        else {
            $query[] = join(', ', $this->selectables);
        }

        $query[] = "FROM";
        $query[] = $this->table;

        if (!empty($this->whereClause)) {
            $query[] = "WHERE";
            $query[] = $this->whereClause;
        }

        if (!empty($this->limit)) {
            $query[] = "LIMIT";
            $query[] = $this->limit;
        }

        return join(' ', $query);
    }
}

// Now to use the class and see how METHOD CHAINING works
// let us instantiate the class DBManager
$testOne = new DBManager();
$testOne->select()->from('users');
echo $testOne->result();
// OR
echo $testOne->select()->from('users')->result();
// both displays: 'SELECT * FROM users'

$testTwo = new DBManager();
$testTwo->select()->from('posts')->where('id > 200')->limit(10);
echo $testTwo->result();
// this displays: 'SELECT * FROM posts WHERE id > 200 LIMIT 10'

$testThree = new DBManager();
$testThree->select(
    'firstname',
    'email',
    'country',
    'city'
)->from('users')->where('id = 2399');
echo $testThree->result();
// this will display:
// 'SELECT firstname, email, country, city FROM users WHERE id = 2399'

?>
```

Another Way for static method chaining :

```
class Maker
{
    private static $result      = null;
    private static $delimiter   = '.';
    private static $data        = [];

    public static function words($words)
    {
        if( !empty($words) && count($words) )
        {
            foreach ($words as $w)
            {
                self::$data[] = $w;
            }
        }
        return new static;
    }

    public static function concate($delimiter)
    {
        self::$delimiter = $delimiter;
        foreach (self::$data as $d)
        {
            self::$result .= $d.$delimiter;
        }
        return new static;
    }

    public static function get()
    {
        return rtrim(self::$result, self::$delimiter);
    }
}
```

Calling

```
echo Maker::words(['foo', 'bob', 'bar'])->concate('-')->get();

echo "<br />";

echo Maker::words(['foo', 'bob', 'bar'])->concate('>')->get();
```

A fluent interface allows you to chain method calls, which results in less typed characters when applying multiple operations on the same object.

```
class Bill {

    public $dinner    = 20;

    public $desserts  = 5;

    public $bill;

    public function dinner( $person ) {
        $this->bill += $this->dinner * $person;
        return $this;
```

```
    }
    public function dessert( $person ) {
        $this->bill += $this->desserts * $person;
        return $this;
    }
}

$bill = new Bill();

echo $bill->dinner( 2 )->dessert( 3 )->bill;
```

0

I think this is the most relevant answer.

```
<?php

class Calculator
{
  protected $result = 0;

  public function sum($num)
  {
    $this->result += $num;
    return $this;
  }

  public function sub($num)
  {
    $this->result -= $num;
    return $this;
  }

  public function result()
  {
    return $this->result;
  }
}

$calculator = new Calculator;
echo $calculator->sum(10)->sub(5)->sum(3)->result(); // 8
```

https://stackoverflow.com/questions/3724112/php-method-chaining-or-fluent-interface