Refactoring to Collections

The Definitive Guide to Curing the Common Loop

by **Adam Wathan**

Contents

Higher Order Functions

A higher order function is a function that takes another function as a parameter, returns a function, or does both.

For example, here's a higher order function that wraps a block of code in a database transaction:

```
public function transaction($func)
{
    $this->beginTransaction();

    try {
        $result = $func();
        $this->commitTransaction();
    } catch (Exception $e) {
        $this->rollbackTransaction();
        throw $e;
    }

    return $result
}
```

And here's what it would look like to use:

```
try {
    $databaseConnection->transaction(function () use ($comment) {
        $comment->save();
    });
} catch (Exception $e) {
    echo "Something went wrong!";
}
```

Noticing Patterns

Higher order functions are powerful because they let us create abstractions around common programming patterns that couldn't otherwise be reused.

Say we have a list of customers and we need to get a list of their email addresses. We can implement that without any higher order functions like this:

```php
$customerEmails = [];

foreach ($customers as $customer) {
    $customerEmails[] = $customer->email;
}

return $customerEmails;
```

Now say we also have a list of product inventory and we want to know the total value of our stock of each item. We might write something like this:

```php
$stockTotals = [];

foreach ($inventoryItems as $item) {
    $stockTotals[] = [
        'product' => $item->productName,
        'total_value' => $item->quantity * $item->price,
    ];
}

return $stockTotals;
```

At first glance it might not look like there's much to abstract here, but if you look carefully you'll notice there's only one real difference between these two examples.

In both cases, all we're doing is building a new array of items by applying some operation to every item in the existing list. The only difference between the two examples is the actual operation that we apply.

In the first example we're just extracting the `email` field from the item:

```
$customerEmails = [];

foreach ($customers as $customer) {
    $email = $customer->email;
    $customerEmails[] = $email;
}

return $customerEmails;
```

In the second example, we create a new associative array from several of the item's fields:

```
$stockTotals = [];

foreach ($inventoryItems as $item) {
    $stockTotal = [
        'product' => $item->productName,
        'total_value' => $item->quantity * $item->price,
    ];
    $stockTotals[] = $stockTotal;
}

return $stockTotals;
```

If we generalize the names of everything except the two chunks of code that are different, we get this:

```
$results = [];

foreach ($items as $item) {
    $result = $item->email;
    $results[] = $result;
}

return $results;
```

```
$results = [];

foreach ($items as $item) {
    $result = [
        'product' => $item->productName,
        'total_value' => $item->quantity * $item->price,
    ];
    $results[] = $result;
}

return $results;
```

We're close to an abstraction here, but those two pesky chunks of code in the middle are preventing us from getting there. We need to get those pieces out and replace them with something that can stay the same for both examples.

We can do that by extracting those chunks of code into anonymous functions. Each anonymous function just takes the current item as its parameter, applies the operation to that item, and returns it.

Here's the email example after extracting an anonymous function:

```
$func = function ($customer) {
    return $customer->email;
};

$results = [];

foreach ($items as $item) {
    $result = $func($item);
    $results[] = $result;
}

return $results;
```

...and here's the inventory example after the same extraction:

```
$func = function ($item) {
    return [
        'product' => $item->productName,
        'total_value' => $item->quantity * $item->price,
    ];
};

$results = [];

foreach ($items as $item) {
    $result = $func($item);
    $results[] = $result;
}

return $results;
```

Now there's a big block of identical code in both examples that we can extract into something reusable. If we bundle that up into its own function, we've implemented a higher order function called `map`!

```
function map($items, $func)
{
    $results = [];

    foreach ($items as $item) {
        $results[] = $func($item);
    }

    return $results;
}

$customerEmails = map($customers, function ($customer) {
    return $customer->email;
});
```

```
$stockTotals = map($inventoryItems, function ($item) {
    return [
        'product' => $item->productName,
        'total_value' => $item->quantity * $item->price,
    ];
});
```

Functional Building Blocks

Map is just one of dozens of powerful higher order functions for working with arrays. We'll talk about a lot of them in later examples, but let's cover some of the fundamental ones in depth first.

Each

Each is no more than a `foreach` loop wrapped inside of a higher order function:

```
function each($items, $func)
{
    foreach ($items as $item) {
        $func($item);
    }
}
```

You're probably asking yourself, *"why would anyone bother to do this?"* Well for one, it hides the implementation details of the loop *(and we hate loops.)*

Imagine a world where PHP didn't have a `foreach` loop. Our implementation of `each` would look something like this:

```
function each($items, $func)
{
    for ($i = 0; $i < count($items); $i++) {
        $func($items[$i]);
    }
}
```

In that world, having an abstraction around "do this with every item in the array" seems pretty reasonable. It would let us take code that looks like this:

```
for ($i = 0; $i < count($productsToDelete); $i++) {
    $productsToDelete[$i]->delete();
}
```

...and rewrite it like this, which is a bit more expressive:

```
each($productsToDelete, function ($product) {
    $product->delete();
});
```

Each also becomes an obvious improvement over using `foreach` directly as soon as you get into chaining functional operations, which we'll cover later in the book.

A couple things to remember about `each`:

- If you're tempted to use any sort of collecting variable, `each` is not the function you should be using.

  ```
  // Bad! Use `map` instead.
  each($customers, function ($customer) use (&$emails) {
      $emails[] = $customer->email;
  });

  // Good!
  $emails = map($customers, function ($customer) {
      return $customer->email;
  });
  ```

- Unlike the other basic array operations, `each` doesn't return anything. That's a clue that it should be reserved for *performing actions*, like deleting products, shipping orders, sending emails, etc.

```
each($orders, function ($order) {
    $order->markAsShipped();
});
```

Map

We've talked about map a bit already, but it's an important one and deserves its own reference.

Map is used to *transform* each item in an array into something else. Given some array of items and a function, map will apply that function to every item and spit out a new array of the same size.

Here's what `map` looks like as a loop:

```
function map($items, $func)
{
    $result = [];

    foreach ($items as $item) {
        $result[] = $func($item);
    }

    return $result;
}
```

Remember, every item in the new array has a relationship with the corresponding item in the original array. A good way to remember how `map` works is to think of there being a *mapping* between each item in the old array and the new array.

Map is a great tool for jobs like:

- Extracting a field from an array of objects, such as mapping customers into their email addresses:

```
$emails = map($customers, function ($customer) {
    return $customer->email;
});
```

- Populating an array of objects from raw data, like mapping an array of JSON results into an array of domain objects:

```
$products = map($productJson, function ($productData) {
    return new Product($productData);
});
```

- Converting items into a new format, for example mapping an array of prices in cents into a displayable format:

```
$displayPrices = map($prices, function ($price) {
    return '$' . number_format($price / 100, 2);
});
```

Map vs. Each

A common mistake I see people make is using `map` when they should have used `each`.

Consider our `each` example from before where we were deleting products. You could implement the same thing using `map` and it would technically have the same effect:

```
map($productsToDelete, function ($product) {
    $product->delete();
});
```

Although this code works, it's semantically incorrect. We didn't `map` anything here. This code is going to go through all the trouble of creating a new array for us where every element is `null` and we aren't going to do anything with it.

Map is about transforming one array into another array. If you aren't transforming anything, you shouldn't be using `map`.

As a general rule, you should be using `each` instead of `map` if any of the following are true:

1. Your callback doesn't return anything.
2. You don't do anything with the return value of `map`.
3. You're just trying to perform some action with every element in an array.

What's Your GitHub Score?

Here's one that originally came out of an interview question someone shared on Reddit.

GitHub provides a public API endpoint that returns all of a user's recent public activity. The JSON response it gives you is an array of objects shaped generally like this (simplified a bit for brevity):

```
[
    {
      "id": "3898913063",
      "type": "PushEvent",
      "public": true,
      "actor": "adamwathan",
      "repo": "tightenco/jigsaw",
      "payload": { /* ... */ }
    },
    // ...
]
```

Check it out for yourself by making a `GET` request to this URL:

```
https://api.github.com/users/{your-username}/events
```

The interview task was to take these events and determine a user's "GitHub Score", based on the following rules:

1. Each `PushEvent` is worth 5 points.
2. Each `CreateEvent` is worth 4 points.
3. Each `IssuesEvent` is worth 3 points.
4. Each `CommitCommentEvent` is worth 2 points.
5. All other events are worth 1 point.

Loops and Conditionals

First let's take a look at an imperative approach to solving this problem:

```php
function githubScore($username)
{
    // Grab the events from the API, in the real world you'd probably use
    // Guzzle or similar here, but keeping it simple for the sake of brevity.
    $url = "https://api.github.com/users/{$username}/events";
    $events = json_decode(file_get_contents($url), true);

    // Get all of the event types
    $eventTypes = [];

    foreach ($events as $event) {
        $eventTypes[] = $event['type'];
    }

    // Loop over the event types and add up the corresponding scores
    $score = 0;

    foreach ($eventTypes as $eventType) {
        switch ($eventType) {
            case 'PushEvent':
                $score += 5;
                break;
            case 'CreateEvent':
                $score += 4;
                break;
            case 'IssuesEvent':
                $score += 3;
                break;
            case 'CommitCommentEvent':
                $score += 2;
                break;
            default:
                $score += 1;
                break;
        }
    }

    return $score;
}
```

Let's start cleaning!

Replace Collecting Loop with Pluck

First things first, let's wrap the GitHub events in a collection:

```
  function githubScore($username)
  {
      $url = "https://api.github.com/users/{$username}/events";
-     $events = json_decode(file_get_contents($url), true);
+     $events = collect(json_decode(file_get_contents($url), true));

      // ...
  }
```

Now let's take a look at this first loop:

```
function githubScore($username)
{
    $url = "https://api.github.com/users/{$username}/events";
    $events = collect(json_decode(file_get_contents($url), true));

    $eventTypes = [];

    foreach ($events as $event) {
        $eventTypes[] = $event['type'];
    }
    $score = 0;

    foreach ($eventTypes as $eventType) {
        switch ($eventType) {
            case 'PushEvent':
                $score += 5;
                break;
            // ...
        }
    }

    return $score;
}
```

We know by know that any time we're *transforming* each item in an array into something new we can use `map` right? In this case, the transformation is so simple that we can even use `pluck`, so let's swap that out:

```
function githubScore($username)
{
    $url = "https://api.github.com/users/{$username}/events";
    $events = collect(json_decode(file_get_contents($url), true));

    $eventTypes = $events->pluck('type');

    $score = 0;

    foreach ($eventTypes as $eventType) {
        switch ($eventType) {
            case 'PushEvent':
                $score += 5;
                break;
            // ...
        }
    }

    return $score;
}
```

Already four lines gone and a lot more expressive, nice!

Extract Score Conversion with Map

How about this second big loop with the `switch` statement?

```
function githubScore($username)
{
    $url = "https://api.github.com/users/{$username}/events";
    $events = collect(json_decode(file_get_contents($url), true));

    $eventTypes = $events->pluck('type');

    $score = 0;

    foreach ($eventTypes as $eventType) {
        switch ($eventType) {
            case 'PushEvent':
                $score += 5;
                break;
            case 'CreateEvent':
                $score += 4;
                break;
            case 'IssuesEvent':
                $score += 3;
                break;
            case 'CommitCommentEvent':
                $score += 2;
                break;
            default:
                $score += 1;
                break;
        }
    }

    return $score;
}
```

We're trying to sum up a bunch of scores here, but we're doing it using a collection of event types.

Maybe this would be simpler if we could just sum a collection of scores instead? Let's convert the event types to scores using `map`, then just return the `sum` of that collection:

```php
function githubScore($username)
{
    $url = "https://api.github.com/users/{$username}/events";
    $events = collect(json_decode(file_get_contents($url), true));

    $eventTypes = $events->pluck('type');

    $scores = $eventTypes->map(function ($eventType) {
        switch ($eventType) {
            case 'PushEvent':
                return 5;
            case 'CreateEvent':
                return 4;
            case 'IssuesEvent':
                return 3;
            case 'CommitCommentEvent':
                return 2;
            default:
                return 1;
        }
    });

    return $scores->sum();
}
```

This is a little bit better, but that nasty `switch` statement is really cramping our style. Let's tackle that next.

Replace Switch with Lookup Table

Almost any time you have a `switch` statement like this, you can replace it with an associative array lookup, where the `case` becomes the array key:

```
function githubScore($username)
{
    $url = "https://api.github.com/users/{$username}/events";
    $events = collect(json_decode(file_get_contents($url), true));

    $eventTypes = $events->pluck('type');

    $scores = $eventTypes->map(function ($eventType) {
        $eventScores = [
            'PushEvent' => 5,
            'CreateEvent' => 4,
            'IssuesEvent' => 3,
            'CommitCommentEvent' => 2,
        ];

        return $eventScores[$eventType];
    });

    return $scores->sum();
}
```

This feels cleaner to me because *looking up* the score for an event seems like a much more natural model of what we're trying to do vs. a conditional structure like `switch`.

The problem is we've lost the default case, where all other events are given a score of 1.

To get that behavior back, we need to check if our event exists in the lookup table before trying to access it:

```
function githubScore($username)
{
    // ...

    $scores = $eventTypes->map(function ($eventType) {
        $eventScores = [
            'PushEvent' => 5,
            'CreateEvent' => 4,
            'IssuesEvent' => 3,
            'CommitCommentEvent' => 2,
        ];

        if (! isset($eventScores[$eventType])) {
            return 1;
        }

        return $eventScores[$eventType];
    });

    return $scores->sum();
}
```

All of a sudden this doesn't really seem better than the `switch` statement, but fear not, there's still hope!

Associative Collections

Everything is better as a collection, remember?

So far we've only used collections for traditional numeric arrays, but collections offer us a lot of power when working with associative arrays as well.

Have you ever heard of the "Tell, Don't Ask" principle? The general idea is that you should avoid asking an object a question about itself to make another decision about something you are going to do with that object. Instead, you should push that responsibility *into* that object, so you can just tell it what you need without asking questions first.

How is that relevant in this example? I'm glad you asked! Let's take a look at that `if` statement again:

```php
$eventScores = [
    'PushEvent' => 5,
    'CreateEvent' => 4,
    'IssuesEvent' => 3,
    'CommitCommentEvent' => 2,
];

if (! isset($eventScores[$eventType])) {
    return 1;
}

return $eventScores[$eventType];
```

Here we are asking the lookup table if it has a value for a certain key, and if it doesn't we return a default value.

Collections let us apply the "Tell, Don't Ask" principle in this situation with the `get` method, which takes a key to look up *and a default value to return if that key doesn't exist!*

If we wrap `$eventScores` in a collection, we can refactor the above code like so:

```php
$eventScores = collect([
    'PushEvent' => 5,
    'CreateEvent' => 4,
    'IssuesEvent' => 3,
    'CommitCommentEvent' => 2,
]);

return $eventScores->get($eventType, 1);
```

Collapsing that down and putting it back into context of the entire function gives us this:

```
function githubScore($username)
{
    $url = "https://api.github.com/users/{$username}/events";
    $events = collect(json_decode(file_get_contents($url), true));

    $eventTypes = $events->pluck('type');

    $scores = $eventTypes->map(function ($eventType) {
        return collect([
            'PushEvent' => 5,
            'CreateEvent' => 4,
            'IssuesEvent' => 3,
            'CommitCommentEvent' => 2,
        ])->get($eventType, 1);
    });

    return $scores->sum();
}
```

Now we can collapse that entire thing down into a single pipeline:

```
function githubScore($username)
{
    $url = "https://api.github.com/users/{$username}/events";
    $events = collect(json_decode(file_get_contents($url), true));

    return $events->pluck('type')->map(function ($eventType) {
        return collect([
            'PushEvent' => 5,
            'CreateEvent' => 4,
            'IssuesEvent' => 3,
            'CommitCommentEvent' => 2,
        ])->get($eventType, 1);
    })->sum();
}
```

Extracting Helper Functions

Sometimes the bodies of operations like `map` can grow to several lines, like looking up the event score has here.

We haven't talked about this much so far, but just because we're working with collection pipelines doesn't mean we should throw out other good practices like extracting logic into small functions.

In this case, I would extract both the API call and the score lookup into separate functions, giving a solution like this:

```
function githubScore($username)
{
    return fetchEvents($username)->pluck('type')->map(function ($eventType) {
        return lookupEventScore($eventType);
    })->sum();
}

function fetchEvents($username)
{
    $url = "https://api.github.com/users/{$username}/events";
    return collect(json_decode(file_get_contents($url), true));
}

function lookupEventScore($eventType)
{
    return collect([
        'PushEvent' => 5,
        'CreateEvent' => 4,
        'IssuesEvent' => 3,
        'CommitCommentEvent' => 2,
    ])->get($eventType, 1);
}
```

Encapsulating in a Class

What would it look like to fetch someone's GitHub score in a typical modern PHP web app? Surely we wouldn't just have a bunch of global functions floating around calling each other right?

One approach is to create a class that works kind of like a namespace and exposes static functions so you can control their visibility:

```
class GitHubScore
{
    public static function forUser($username)
    {
        return self::fetchEvents($username)
            ->pluck('type')
            ->map(function ($eventType) {
                return self::lookupScore($eventType);
            })->sum();
    }

    private static function fetchEvents($username)
    {
        $url = "https://api.github.com/users/{$this->username}/events";
        return collect(json_decode(file_get_contents($url), true));
    }

    private static function lookupScore($eventType)
    {
        return collect([
            'PushEvent' => 5,
            'CreateEvent' => 4,
            'IssuesEvent' => 3,
            'CommitCommentEvent' => 2,
        ])->get($eventType, 1);
    }
}
```

With this class, I could make a call like `GitHubScore::forUser('adamwathan')` and get a score back.

One of the issues with this approach is that because we're not working with actual objects, we can't keep track of any state anymore. Instead, you end up passing the same parameter around in a bunch of places because you don't really have anywhere to store that data.

It's not too bad in this example, but you can see here we have to pass `$username` into `fetchEvents` since otherwise the method has no way of knowing which user's activity to fetch:

```
class GitHubScore
{
    public static function forUser($username)
    {
        return self::fetchEvents($username)
            ->pluck('type')
            ->map(function ($eventType) {
                return self::lookupScore($event['type']);
            })->sum();
    }

    private static function fetchEvents($username)
    {
        $url = "https://api.github.com/users/{$this->username}/events";
        return collect(json_decode(file_get_contents($url), true));
    }

    // ...
}
```

This can get ugly pretty fast when you've extracted a handful of small methods that need access to the same data.

A neat trick I use in situations like this is to create what I've been calling *private instances.*

Instead of doing all of the work with static methods, I create an instance of the class in the first static method, then delegate all of the work to that instance.

Here's what it looks like:

```php
class GitHubScore
{
    private $username;

    private function __construct($username)
    {
        $this->username = $username;
    }

    public static function forUser($username)
    {
        return (new self($username))->score();
    }

    private function score()
    {
        $this->events()->pluck('type')->map(function ($eventType) {
            return $this->lookupScore($eventType);
        })->sum();
    }

    private function events()
    {
        $url = "https://api.github.com/users/{$this->username}/events";
        return collect(json_decode(file_get_contents($url), true));
    }
    private function lookupScore($eventType)
    {
        return collect([
            'PushEvent' => 5,
            'CreateEvent' => 4,
            'IssuesEvent' => 3,
            'CommitCommentEvent' => 2,
        ])->get($eventType, 1);
    }
}
```

You get the same convenient static API, but internally you get to work with an object that has it's own state, which keeps your method signatures short and simple. Pretty neat stuff!

This is a sample of "Refactoring to Collections" by Adam Wathan. To learn more about the book and read the rest, head over to http://adamwathan.me/refactoring-to-collections.