# Refactoring to Collections

## The Definitive Guide to Curing the Common Loop

*by* **Adam Wathan**

# Contents

**A Bit of Theory** **7**

Imperative vs. Declarative Programming . . . . . 7

Imperative Programming . . . . . 7

Declarative Programming . . . . . 9

Higher Order Functions . . . . . 9

Noticing Patterns . . . . . 10

Functional Building Blocks . . . . . 15

Each . . . . . 15

Map . . . . . 17

Filter . . . . . 19

Reduce . . . . . 23

Transforming Data . . . . . 31

Thinking in Steps . . . . . 33

The Problem with Primitives . . . . . 35

Arrays as Objects . . . . . 37

Introducing Collections . . . . . 38

A Note on Mutability . . . . . 40

Quacking Like... an Array? . . . . . 42

The Golden Rule of Collection Programming . . . . . 49

**A Lot of Practice** 51
Pricing Lamps and Wallets . . . 52
Replace Conditional with Filter . . . 54
Replace || with Contains . . . 55
Reduce to Sum . . . 56
Replace Nested Loop with FlatMap . . . 58
Plucking for Fun and Profit . . . 61
CSV Surgery 101 . . . 62
Everything is Better as a Collection . . . 65
Binary to Decimal . . . 67
A Quick Refresher . . . 67
Using a For Loop . . . 67
Breaking It Down . . . 68
Reversing the Collection . . . 69
Mapping with Keys . . . 70
What's Your GitHub Score? . . . 72
Loops and Conditionals . . . 73
Replace Collecting Loop with Pluck . . . 74
Extract Score Conversion with Map . . . 76
Replace Switch with Lookup Table . . . 78
Associative Collections . . . 79
Extracting Helper Functions . . . 82
Encapsulating in a Class . . . 83
Formatting a Pull Request Comment . . . 86
Concatenating in a Loop . . . 87
Map and Implode . . . 87
Stealing Mail . . . 88
Replace Nested Check with Contains . . . 90

Contains as a Higher Order Function . . . . . 91
Choosing a Syntax Handler . . . . . 94
Looking for a Match . . . . . 95
Getting the Right Checker . . . . . 96
Replace Iteration with First . . . . . 97
A Hidden Rule. . . . . 98
Providing a Default . . . . . 99
The Null Object Pattern . . . . . 100
The Null Checker . . . . . 101
Tagging on the Fly. . . . . 104
Extracting the Loop . . . . . 105
Normalizing with Map . . . . . 107
Nitpicking a Pull Request . . . . . 109
A Fork in the Code . . . . . 110
Learning from Smalltalk . . . . . 111
Collection Macros . . . . . 113
Chainable Conditions . . . . . 114
Comparing Monthly Revenue . . . . . 117
Matching on Index. . . . . 118
Zipping Things Together . . . . . 119
Using Zip to Compare. . . . . 119
Transposing Form Input . . . . . 121
Quick and Dirty . . . . . 125
Identifying a Need . . . . . 126
Introducing Transpose . . . . . 128
Implementing Transpose . . . . . 130
Transpose in Practice . . . . . 130

Ranking a Competition . . . . . 132
Zipping-in the Ranks . . . . . 134
Dealing with Ties . . . . . 135
One Step at a Time . . . . . 136
Grouping by Score . . . . . 137
Adjusting the Ranks . . . . . 139
Collapse and Sort . . . . . 142
Cleaning Up . . . . . 144
Grouping Operations . . . . . 147
Breaking the Chain . . . . . 148
The Pipe Macro . . . . . 149
**Afterword** **152**

# A Bit of Theory

## Imperative vs. Declarative Programming

You've probably heard the terms imperative (or procedural) and declarative programming before, and if you're anything like me, went looking for a precise definition of the two only to find some vague hand wavy descriptions that didn't give you any concrete answers.

Over time I've realized that this is because it really isn't black or white. Code snippet A can be *more* declarative than code snippet B, but maybe code snippet C is even more declarative than code snippet A.

Here's my best shot at explaining how I think about imperative and declarative programming.

### Imperative Programming

Imperative programming is a style of programming that focuses on *how* something gets done. The code is usually overly concerned with building results in intermediate variables and managing control flow with loops and conditional statements.

Say we have a list of users and we want to fetch the email addresses of all users who have an email address on file.

An imperative solution in PHP could look like this:

```
function getUserEmails($users)
{
    $emails = [];

    for ($i = 0; $i < count($users); $i++) {
        $user = $users[$i];

        if ($user->email !== null) {
            $emails[] = $user->email;
        }
    }

    return $emails;
}
```

This probably seems innocent enough, and we've all written code that looks like this. But think about what this code is saying:

1. Create an empty array that we will use to build our result
2. Create a variable to store our counter, starting at 0
3. Check our counter variable to make sure it is still less than the number of users in the array
    - If so:
        1. Create a reference to the item in the array at the location matching our current counter value
        2. Check if the `email` property of the user is equal to `null`
            - If not, add that user's email address to the end of our result array
        3. Increment our counter by one
        4. Return to step 3
    - If not, return our result array

Instead of trying to say *"give me the emails of the users who have emails"*, the solution focuses on implementation details about how many times to repeat chunks of code, accessing indexes on data structures, and managing counters.

### Declarative Programming

Instead of focusing on *how* the computer should do the work, declarative programming focuses on telling the computer *what* we need it to accomplish.

Compare the code above to the same operation in SQL:

```
SELECT email FROM users WHERE email IS NOT NULL
```

We didn't have to write anything about loops, counters, or array indexes. We just told the computer *what* we wanted, not *how* to get it.

Under the hood, I'm sure the SQL engine must be doing some sort of iteration or keeping track of which records it's checked or which records it hasn't, but I don't really know for sure.

And that's the beauty of it: *I don't need to know.*

PHP is a much different beast than SQL of course, and we're not going to be able to recreate that exact syntax.

But can we get any closer? Sure we can, using higher order functions!

## 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:

```
$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:

```
$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.

## Filter

Say we had a list of products and we needed to know which ones were out of stock. Without using any higher order functions, we could write that code like this:

```
$outOfStockProducts = [];

foreach ($products as $product) {
    if ($product->isOutOfStock()) {
        $outOfStockProducts[] = $product;
    }
}

return $outOfStockProducts;
```

Similarly, if we wanted the products that cost more than $100, we could write this:

```
$expensiveProducts = [];

foreach ($products as $product) {
    if ($product->price > 100) {
        $expensiveProducts[] = $product;
    }
}

return $expensiveProducts;
```

The only difference between these two examples is the conditional. We can abstract that difference away by extracting anonymous functions, like we did with `map`:

```
$func = function ($product) {
    return $product->isOutOfStock();
};

$results = [];

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

return $results;
```

```
$func = function ($product) {
    return $product->price > 100;
};

$results = [];

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

return $results;
```

Bundling up what's left gives us an implementation of a higher order function called `filter`:

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

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

    return $result;
}

$outOfStockProducts = filter($products, function ($product) {
    return $product->isOutOfStock();
});

$expensiveProducts = filter($products, function ($product) {
    return $product->price > 100;
});
```

The `filter` operation is used to *filter out* any elements of an array that you don't want. You tell `filter` which elements to keep by passing a callback that returns `true` if you want to keep the element, and `false` if you want it removed.

It's important to understand that `filter` doesn't actually change or transform any of the items in the array; it just strips out the items you don't want. That means that the items that make it into the new array are not only the same *type* as the ones in the old array, they're the *same items*.

This is in stark contrast to `map`, which is used to create *new* items by applying some operation to the existing items. You might `map` *products* into *prices*, but you always `filter` products into a new array of products.

#### Reject

Sometimes it can be more expressive to specify the items we want to *discard* instead of the items we want to *keep*.

*Reject* is a close cousin of `filter` that let's us do just that, simply by flipping the conditional:

```
- function filter($items, $func)
+ function reject($items, $func)
  {
      $result = [];

      foreach ($items as $item) {
-         if ($func($item)) {
+         if (! $func($item)) {
              $result[] = $item;
          }
      }

      return $result;
  }
```

We can even implement `reject` *in terms of* `filter` just by negating the callback:

```
function reject($items, $func)
{
    return filter($items, function ($item) use ($func) {
        return ! $func($item);
    });
}
```

Say we needed a list of products that are in stock and our `Product` class only exposes an `outOfStock` method. An implementation using `filter` would look like this:

```
$inStockProducts = filter($products, function ($product) {
    return ! $product->isOutOfStock();
});
```

By using `filter`, our code is saying "*keep* the products that *are not* out of stock." It's technically correct, but sort of awkward to say. Compare that to an implementation that uses `reject`:

```
$inStockProducts = reject($products, function ($product) {
    return $product->isOutOfStock();
});
```

Now our code is saying, "*discard* the products that *are* out of stock". It's a subtle difference, but I think the code is clearer as a result.

## Reduce

Say we had a shopping cart of items and we needed to calculate the total price of the cart. One way to do that would be to loop over all of the items in the cart and keep a running tally of the total price:

```
$totalPrice = 0;

foreach ($cart->items as $item) {
    $totalPrice = $totalPrice + $item->price;
}

return $totalPrice;
```

Now imagine another situation where we wanted to send an email to a group of customers and we needed to generate a comma separated list of their emails for the BCC line. We could build that string by looping over the customers and concatenating their emails together, like so:

```
$bcc = '';

foreach ($customers as $customer) {
    $bcc = $bcc . $customer->email . ', ';
}

return $bcc;
```

These examples are of course very similar, so what would it look like to create an abstraction around this operation? Let's step through it line-by-line and see if we can find a way to extract a higher order function.

First, let's rename both `$totalPrice` and `$bcc` to something more general like `$accumulator` since we are using it to build up our final result:

```
$accumulator = 0;

foreach ($cart->items as $item) {
    $accumulator = $accumulator + $item->price;
}

return $accumulator;
```

```
$accumulator = '';

foreach ($customers as $customer) {
    $accumulator = $accumulator . $customer->email . ', ';
}

return $accumulator;
```

Comparing the first two lines, there's still a small difference. Unlike `filter` and `map` which both always start with an empty array, one of these examples begins with a `0` while the other begins with an empty string:

```
$accumulator = 0;

foreach ($cart->items as $item) {
    $accumulator = $accumulator + $item->price;
}

return $accumulator;
```

```
$accumulator = '';

foreach ($customers as $customer) {
    $accumulator = $accumulator . $customer->email . ', ';
}

return $accumulator;
```

We can get around this by extracting the initial value into a separate variable:

```
$initial = 0;

$accumulator = $initial;

foreach ($cart->items as $item) {
    $accumulator = $accumulator + $item->price;
}

return $accumulator;
```

```
$initial = '';

$accumulator = $initial;

foreach ($customers as $customer) {
    $accumulator = $accumulator . $customer->email . ', ';
}

return $accumulator;
```

Next if we look at the `foreach` loop, one example is looping over the cart items and the other is looping over customers. We can generalize that by pulling out a temporary variable called `$items`:

```
$items = $cart->items;
$initial = 0;

$accumulator = $initial;

foreach ($items as $item) {
    $accumulator = $accumulator + $item->price;
}

return $accumulator;
```

```
$items = $customers;
$initial = '';

$accumulator = $initial;

foreach ($items as $item) {
    $accumulator = $accumulator . $item->email . ', ';
}

return $accumulator;
```

Now the only difference between the two examples is how we are building our `$accumulator`. In the cart example we're *adding* the item's price to the current `$accumulator`, while in the email example we are *concatenating* the customer's email with the `$accumulator`.

```
$items = $cart->items;
$initial = 0;

$accumulator = $initial;

foreach ($items as $item) {
    $accumulator = $accumulator + $item->price;
}

return $accumulator;
```

```
$items = $customers;
$initial = '';

$accumulator = $initial;

foreach ($items as $item) {
    $accumulator = $accumulator . $item->email . ', ';
}

return $accumulator;
```

Normally we would extract these pieces of code into anonymous functions that took the current item as their only parameter.

But in this case, the code we are trying to extract depends on both the current item *and* what we've built up to that point in `$accumulator`.

We can accommodate this by taking *both* values as parameters in our anonymous functions:

```
$items = $cart->items;
$callback = function ($totalPrice, $item) {
    return $totalPrice + $item->price;
};
$initial = 0;

$accumulator = $initial;

foreach ($items as $item) {
    $accumulator = $callback($accumulator, $item);
}

return $accumulator;
```

```
$items = $customers;
$callback = function ($bcc, $customer) {
    return $bcc . $customer->email . ', ';
};
$initial = '';

$accumulator = $initial;

foreach ($items as $item) {
    $accumulator = $callback($accumulator, $item);
}

return $accumulator;
```

Finally we have two blocks of identical code in both examples:

```
$items = $cart->items;
$callback = function ($totalPrice, $item) {
    return $totalPrice + $item->price;
};
$initial = 0;

$accumulator = $initial;

foreach ($items as $item) {
    $accumulator = $callback($accumulator, $item);
}

return $accumulator;
```

```
$items = $customers;
$callback = function ($bcc, $customer) {
    return $bcc . $customer->email . ', ';
};
$initial = '';

$accumulator = $initial;

foreach ($items as $item) {
    $accumulator = $callback($accumulator, $item);
}

return $accumulator;
```

We can extract that code into a higher order function called `reduce`, which takes all of the variables we extracted as parameters:

```
function reduce($items, $callback, $initial)
{
    $accumulator = $initial;

    foreach ($items as $item) {
        $accumulator = $callback($accumulator, $item);
    }

    return $accumulator;
}

$totalPrice = reduce($cart->items, function ($totalPrice, $item) {
    return $totalPrice + $item->price;
}, 0);

$bcc = reduce($customers, function ($bcc, $customer) {
    return $bcc . $customer->email . ', ';
}, '');
```

### With Great Power

The `reduce` operation is used to take some array of items and *reduce* it down to a single value. It has no opinion about what that single value should be; it could be a number, a string, an object, whatever you want, it doesn't matter.

It can even be used to reduce one array of items into *another* array, which means we can even implement `map` and `filter` in terms of `reduce`.

Here's what `map` would look like:

```
function map($items, $func)
{
    return reduce($items, function ($mapped, $item) use ($func) {
        $mapped[] = $func($item);
        return $mapped;
    }, []);
}
```

Since `reduce` is a pretty low level functional operation that can turn an array into just about anything, it's not always very expressive on its own. Sometimes when I find myself using `reduce`, what I really want is a higher level abstraction built *on top of* `reduce` that communicates what I'm trying to do more clearly.

For instance, in our total cart price example we could create a new abstraction on top of `reduce` called `sum` that simply sums the values returned from some callback:

```
function sum($items, $callback)
{
    return reduce($items, function ($total, $item) use ($callback) {
        return $total + $callback($item);
    }, 0);
}

$totalPrice = sum($cart->items, function ($item) {
    return $item->price;
});
```

In our BCC list example, we could use `reduce` to write a function like `join` *(ignoring the existing join function for a minute)* that concatenates all of the values returned from a callback:

```
function join($items, $callback)
{
    return reduce($items, function ($string, $item) use ($callback) {
        return $string . $callback($item);
    }, '');
}

$bcc = join($customers, function ($customer) {
    return $customer->email . ', ';
});
```

Between the four fundamental operations we've covered so far, `reduce` is definitely the trickiest to wrap your head around. For a bit of practice, try and reimplement `filter` in terms of `reduce` and see what you come up with.

## Transforming Data

So back to our original example, how can we get this:

```
function getUserEmails($users)
{
    $emails = [];

    for ($i = 0; $i < count($users); $i++) {
        $user = $users[$i];

        if ($user->email !== null) {
            $emails[] = $user->email;
        }
    }

    return $emails;
}
```

...to look more like this?

```
SELECT email FROM users WHERE email IS NOT NULL
```

For starters, let's switch to a `foreach` loop so there's a little less noise:

```
function getUserEmails($users)
{
    $emails = [];

    foreach ($users as $user) {
        if ($user->email !== null) {
            $emails[] = $user->email;
        }
    }

    return $emails;
}
```

All of a sudden this is starting to look a little more familiar. Doesn't the highlighted code look a lot like what we extracted into `map` previously?

```
function getUserEmails($users)
{
    $emails = [];

    foreach ($users as $user) {
        if ($user->email !== null) {
            $emails[] = $user->email;
        }
    }

    return $emails;
}
```

But if we try to implement `getUserEmails` using `map`, it doesn't quite work.

The point of `map` is to apply a transformation to *every* element in an array, so it always returns an array that is the same size as the original. Our `getUserEmails` function is only meant to return the *subset* of items where the `$email` field isn't `null`.

When we need a subset, we use `filter`, but filtering isn't quite right either.

Filter is meant to give you all of the elements in an array that satisfy some condition, in our case whether or not `$email` is `null`. The problem is `filter` would give us the *users* that have emails, and we want the email addresses themselves.

## Thinking in Steps

The problem we're facing right now is that we're trying to do too many things at the same time. When I'm in a situation like this, the first thing I do is try and turn *"I can't because..."* into *"I could if...".*

For example:

> *I can't use `map` because it would be applied to every user, not just the users with emails.*

...becomes:

> *I could use `map` if I was only working with the users that have emails.*

Well, getting just the users with email addresses sounds like a problem we *can* solve with filter:

```
$usersWithEmails = filter($users, function ($user) {
    return $user->email !== null;
});
```

And now that we are only dealing with users who *do* have emails, we can remove the condition in our loop:

```
function getUserEmails($users)
{
    $usersWithEmails = filter($users, function ($user) {
        return $user->email !== null;
    });

    $emails = [];

    foreach ($usersWithEmails as $user) {
        $emails[] = $user->email;
    }

    return $emails;
}
```

Once the condition is gone, we can replace the loop with a `map`:

```
function getUserEmails($users)
{
    $usersWithEmails = filter($users, function ($user) {
        return $user->email !== null;
    });

    $emails = map($usersWithEmails, function ($user) {
        return $user->email;
    });

    return $emails;
}
```

What we've done here is taken our original solution and split it up into two distinct operations, where each operation has one clear responsibility.

Instead of having one block of code responsible for excluding any users without emails *and* extracting the email from that user, we do the same work in two separate steps.

As your code gets more complex, splitting things up like this starts to pay off in dividends because debugging a sequence of simple, independent operations turns out to be *much* easier than debugging a single complex operation.

## The Problem with Primitives

Even though it might still look a bit foreign, what we have now is simpler in a number of ways:

1. We've eliminated the `if` statement by using `filter` to get rid of the users that have no email address.
2. We've eliminated the `$email` collecting variable by using `map` to *transform* our array of users into an array of emails.
3. We've eliminated the loop, opting to treat it as an implementation detail of `filter` and `map`.

The last improvement I'd like to make is to get rid of the `$usersWithEmails` and `$emails` temporary variables.

Usually the best way to get rid of a temporary variable is to use a refactoring called "Inline Temp", where you replace all occurrences of that variable with the expression assigned to it.

Inlining `$emails` is easy; we just return the result of `map` directly:

```
function getUserEmails($users)
{
    $usersWithEmails = filter($users, function ($user) {
        return $user->email !== null;
    });

    return map($usersWithEmails, function ($user) {
        return $user->email;
    });
}
```

...but the code starts to get a bit cryptic if we inline the `$usersWithEmails` variable:

```
function getUserEmails($users)
{
    return map(filter($users, function ($user) {
        return $user->email !== null;
    }), function ($user) {
        return $user->email;
    });
}
```

Not very easy to read is it? It looks even worse if we use PHP's built-in array functions, since they have an unintuitive parameter order:

```
function getUserEmails($users)
{
    return array_map(function ($user) {
        return $user->email;
    }, array_filter($users, function ($user) {
        return $user->email !== null;
    }));
}
```

The reason this code is difficult to understand is because it has to be read *inside-out.*

This same problem arises when working with strings in PHP. For example, here's some code that converts `snake_case` strings to `camelCase`:

```
$camelString = lcfirst(
    str_replace(' ', '',
        ucwords(str_replace('_', ' ', $snakeString))
    )
);
```

Quick, in what order are things happening here? Takes a bit of effort to parse, doesn't it?

Since strings and arrays are primitive types, we have to operate on them from the outside by passing them as parameters into other functions. This is what leads to "inside-out" code, where you need to count the braces to figure out what's happening first.

Compare the "inside-out" example above to this imaginary syntax:

```
$camelString = $snakeString->replace('_', ' ')
                           ->ucwords()
                           ->replace(' ', '')
                           ->lcfirst();
```

Much easier to understand right?

The difference is that we're treating `$snakeString` as an *object* instead of a primitive type. By calling methods on the object directly instead of passing it around as a parameter, all of a sudden our code reads left to right, with the operations appearing in the order that they're executed.

## Arrays as Objects

Imagine for a second that we could call methods directly on an array. How would that affect our `getUserEmails` function?

```
  function getUserEmails($users)
  {
-     $usersWithEmails = filter($users, function ($user) {
+     $usersWithEmails = $users->filter(function ($user) {
          return $user->email !== null;
      });

-     $emails = map($usersWithEmails, function ($user) {
+     $emails = $usersWithEmails->map(function ($user) {
          return $user->email;
      });

      return $emails;
  }
```

If we inline `$emails` again, we're left with the following:

```
function getUserEmails($users)
{
    $usersWithEmails = $users->filter(function ($user) {
        return $user->email !== null;
    });

    return $usersWithEmails->map(function ($user) {
        return $user->email;
    });
}
```

The difference is that this time `$usersWithEmails` is *outside* of our call to `map` instead of inside. Now when we inline it, `filter` appears before `map`, and our code reads left to right instead of inside-out:

```
function getUserEmails($users)
{
    return $users->filter(function ($user) {
        return $user->email !== null;
    })->map(function ($user) {
        return $user->email;
    });
}
```

Isn't that just delightful? This style of programming is commonly called a *collection pipeline*, and we can totally do it in PHP.

## Introducing Collections

A *collection* is an object that bundles up an array and lets us perform array operations by calling methods on the collection instead of passing the array into functions.

Here's a simple `Collection` class that just supports `map` and `filter`:

```
class Collection
{
    protected $items;

    public function __construct($items)
    {
        $this->items = $items;
    }

    public function map($callback)
    {
        return new static(array_map($callback, $this->items));
    }

    public function filter($callback)
    {
        return new static(array_filter($this->items, $callback));
    }

    public function toArray()
    {
        return $this->items;
    }
}
```

To use this in our `getUserEmails` example, all we need to do is wrap the `$users` parameter in a new `Collection`, and convert the collection back to an array before we return it:

```
function getUserEmails($users)
{
    return (new Collection($users))->filter(function ($user) {
        return $user->email !== null;
    })->map(function ($user) {
        return $user->email;
    })->toArray();
}
```

Chaining methods after a traditional constructor can look a bit cluttered, so I'll often create a *named constructor* to clean things up:

```
class Collection
{
    protected $items;

    public function __construct($items)
    {
        $this->items = $items;
    }

    public static function make($items)
    {
        return new static($items);
    }
    // ...
}
```

Using the named constructor saves us a set of parentheses at the call site and looks a little tidier to my eyes:

```
  function getUserEmails($users)
  {
-     return (new Collection($users))->filter(function ($user) {
+     return Collection::make($users)->filter(function ($user) {
          return $user->email !== null;
      })->map(function ($user) {
          return $user->email;
      })->toArray();
  }
```

## A Note on Mutability

You might have noticed in the examples so far that whenever we apply some operation to an array, we always return a *new* array; we don't actually change the original array.

This is most obvious in our `Collection` implementation above, where we explicitly return a `new static` in both `map` and `filter`:

```
class Collection
{
    // ...

    public function map($callback)
    {
        return new static(array_map($callback, $this->items));
    }

    public function filter($callback)
    {
        return new static(array_filter($this->items, $callback));
    }

    // ...
}
```

Compare that to this implementation where instead of returning a new collection, we just replace the `$items` property:

```
class Collection
{
    // ...

    public function map($callback)
    {
        $this->items = array_map($callback, $this->items);
        return $this;
    }

    public function filter($callback)
    {
        $this->items = array_filter($this->items, $callback);
        return $this;
    }
}
```

This might not seem like a big difference, but it can cause brain-melting, spooky-action-at-a-distance bugs that will take all of the "fun" out of *fun*ctional programming.

Take a look at this code:

```
$employees = new Collection([
    ['name' => 'Mary', 'email' => 'mary@example.com', 'salaried' => true],
    ['name' => 'John', 'email' => 'john@example.com', 'salaried' => false],
    ['name' => 'Kelly', 'email' => 'kelly@example.com', 'salaried' => true],
]);

$employeeEmails = $employees->map(function ($employee) {
    return $employee['email'];
});

$salariedEmployees = $employees->filter(function ($employee) {
    return $employee['salaried'];
});
```

Can you spot the bug?

See that `map` call that gets us the `$employeeEmails`? If we just replace the `$items` property instead of returning a new collection, that `$employees` variable actually becomes a collection of *emails* as soon as `map` is finished. So when we try to `filter` the list of employees, we're actually filtering a list of emails. *Yikes!*

So yeah, don't do this.

## Quacking Like... an Array?

The simple collection we've got so far is pretty neat but it's a bit annoying having to constantly convert our data back and forth between collections and arrays.

It would be nice if we could build a collection that we could use *in place of* an array without our system noticing, and thankfully PHP makes that *(mostly)* possible through a handful of interfaces.

### ArrayAccess

One of the special features of arrays is that you can get the element at a specific offset using square bracket notation.

```
$items = [1, 2, 3];

echo $items[2];
// => 3
```

If we try to do that with our `Collection` class, PHP throws a fit:

```
$items = Collection::make([1, 2, 3]);

echo $items[2];
// => Fatal error: Cannot use object of type Collection as array!
```

We can add support for square bracket notation to our collection by implementing the `ArrayAccess` interface, which consists of four methods:

```
interface ArrayAccess
{
    // Allow the collection to respond to `isset($items['key'])` checks
    abstract public function offsetExists($offset);

    // Allow retrieving an item from the collection using `$items['key']`
    abstract public function offsetGet($offset);

    // Allow adding an item to the end of the collection using `$items[] =
    // $foo` as well as at a specific key using `$items['key'] = $foo`
    abstract public function offsetSet($offset, $value);

    // Allow removing an item from the collection using `unset($items['key'])
    abstract public function offsetUnset($offset);
}
```

To add these methods to our `Collection` object, we just need to delegate the calls to our underlying `$items` property:

```
class Collection implements ArrayAccess
{
    protected $items;

    public function __construct($items)
    {
        $this->items = $items;
    }

    public function offsetExists($offset)
    {
        return array_key_exists($this->items, $offset);
    }

    public function offsetGet($offset)
    {
        return $this->items[$offset];
    }

    public function offsetSet($offset, $value)
    {
        if ($offset === null) {
            $this->items[] = $value;
        } else {
            $this->items[$offset] = $value;
        }
    }

    public function offsetUnset($offset)
    {
        unset($this->items[$offset]);
    }

    // ...
}
```

Now we can work with the offsets in our collection exactly as if our collection was a raw array:

```
$items = Collection::make([1, 2, 3]);

echo $items[2];
// => 3

$items[] = 4;
// => [1, 2, 3, 4]

isset($items[3]);
// => true

unset($items[0]);
// => [2, 3, 4]
```

### Countable

The `Countable` interface allows an object to be passed to PHP's built-in `count` function. Not quite as exciting as what we were able to do with `ArrayAccess`, but I digress.

Here's the interface:

```
interface Countable
{
    abstract public function count();
}
```

Here's what it looks like to implement:

```
class Collection implements ArrayAccess, Countable
{
    protected $items;

    public function __construct($items)
    {
        $this->items = $items;
    }

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

    // ...
}
```

Now we can get the size of our collection just like it was a regular array, which helps us achieve that nice polymorphic effect we're looking for:

```
$items = Collection::make([1, 2, 3]);

count($items);
// => 3
```

As a side benefit, now we have a `count()` method that we can chain with other operations.

For example, we can combine it with `filter` to calculate the number of salaried employees:

```
$employees = new Collection([
    ['name' => 'Mary', 'email' => 'mary@example.com', 'salaried' => true],
    ['name' => 'John', 'email' => 'john@example.com', 'salaried' => false],
    ['name' => 'Kelly', 'email' => 'kelly@example.com', 'salaried' => true],
]);
```

```
$numberOfSalariedEmployees = $employees->filter(function ($employee) {
    return $employee['salaried'];
})->count();
```

If you know you're working with a collection, chaining is pretty much always the better way to use `count()`.

Save the function version for situations where you might have a collection *or* a regular array, because this just looks stupid:

```
// Gross!
$numberOfSalariedEmployees = count($employees->filter(function ($employee) {
    return $employee['salaried'];
}));
```

### IteratorAggregate

There's one other thing we can do with a regular array that we still can't do with our collection, and that's iterate over it using a `foreach` loop.

You might not know this, but you actually *can* use `foreach` to iterate over an object's public properties, no extra programming required:

```
class Foo
{
    public $bar = 'baz';
    public $qux = 'norf';
}

$foo = new Foo;

foreach ($foo as $property => $value) {
    echo $property . ' -> ' . $value;
}
// => bar -> baz
// => qux -> norf
```

If you didn't already know this, it's probably because iterating over an object's public properties isn't really all that useful.

What *would* be useful is if we could tell `foreach` to iterate over our `$items` property, and the `IteratorAggregate` interface lets us do that.

To implement `IteratorAggregate`, we need to add a `getIterator` method to our collection that returns a Traversable.

The easiest way to do that is to return our `$items` property wrapped up in an ArrayIterator:

```
class Collection implements ArrayAccess, Countable, IteratorAggregate
{
    protected $items;

    public function __construct($items)
    {
        $this->items = $items;
    }

    public function getIterator()
    {
        return new ArrayIterator($this->items);
    }

    // ...
}
```

Now we can pass our collection into `foreach` and iterate over the `$items` just like we were directly iterating over a regular array:

```
$collection = Collection::make([1, 2, 3]);

foreach ($collection as $item) {
    echo $item;
}
// => 1
// => 2
// => 3
```

This is pretty cool, but with any luck, when you're done reading this book you'll never want to do it.

Which leads us to...

## The Golden Rule of Collection Programming

*Never use a `foreach` loop outside of a collection!*

Every time you use a `foreach` loop, you're doing it to accomplish something else, and I promise you that "something else" already has a name.

Need to loop over an array to perform some operation on each item and stuff the result into another array? You don't need to loop, you need to *map*.

Need to loop over an array to strip out the items that don't match some criteria? You don't need to loop, you need to *filter*.

Pipeline programming is about operating at a higher level of abstraction. Instead of doing things with the items in a collection, you do things *to the collection itself.*

Map it, filter it, reduce it, sum it, zip it, reverse it, transpose it, flatten it, group it, count it, chunk it, sort it, slice it, search it; if you can do it with a `foreach` loop, you can do it with a collection method.

As soon as you elevate arrays from primitive types to objects that can have their own behavior, there's no reason to ever use a `foreach` loop outside of the collection itself, and I'm going to prove it to you.

From this point forward, you won't see a single `foreach` anywhere other than encapsulated inside a collection method.

Let the games begin!

# A Lot of Practice

Now that you have a pretty good grasp of the fundamentals, let's put them to use and learn some new tricks along the way.

We've seen how we could write our own `Collection`, but for these examples we're going to use an off-the-shelf implementation.

If you're familiar with any of my work outside of this book, you probably know that I'm a big fan of the Laravel framework. Laravel ships with a very flexible and feature-rich `Collection` class, and since it's pretty portable and easy to pull into other projects, we'll use that implementation going forward.

You can create a Laravel Collection in three ways:

1. Passing an array to the traditional constructor:

   ```
   $collection = new Collection($items);
   ```

2. Using the `make` named constructor:

   ```
   $collection = Collection::make($items);
   ```

3. Using the `collect()` helper function:

   ```
   $collection = collect($items);
   ```

Personally, I love the terseness of the helper function and since it also saves us some space in the code samples, so we'll roll with that approach.

If you'd like to follow along and play with these examples yourself, you can pull in the library we'll be using via Composer:

```
composer require illuminate/support
```

Remember, even though we're using a specific library for these examples, these ideas are completely portable and can be applied with any decent collection implementation across many programming languages.

Let's get started!

## Pricing Lamps and Wallets

A while back I came across a little programming challenge that Shopify were posing to potential student interns.

> *Given a JSON feed of products from a store, figure out how much it would cost to buy every variant of every single lamp and wallet that store has for sale.*

Here's an example of what the JSON feed looks like, simplified a bit for brevity:

```
{
  "products": [
    {
      "title": "Small Rubber Wallet",
      "product_type": "Wallet",
      "variants": [
        { "title": "Blue", "price": 29.33 },
        { "title": "Turquoise", "price": 18.50 }
      ]
    },
    {
      "title": "Sleek Cotton Shoes",
      "product_type": "Shoes",
      "variants": [
        { "title": "Sky Blue", "price": 20.00 }
      ]
    },
```

```
    {
      "title": "Intelligent Cotton Wallet",
      "product_type": "Wallet",
      "variants": [
        { "title": "White", "price": 17.97 }
      ]
    },
    {
      "title": "Enormous Leather Lamp",
      "product_type": "Lamp",
      "variants": [
        { "title": "Azure", "price": 65.99 },
        { "title": "Salmon", "price": 1.66 }
      ]
    },
    // ...
  ]
}
```

So we know we're going to have a bunch of different products in there, some of which have a `product_type` of "Lamp" or "Wallet", and some of which don't. Each product also has a number of `variants`, and the variants are what actually have prices.

First things first, let's grab the products out of the JSON and wrap them in a new collection:

```
$url = 'http://shopicruit.myshopify.com/products.json';

$productJson = json_decode(file_get_contents($url), true);

$products = collect($productJson['products']);
```

We'll use this imperative solution as a starting point:

```
$totalCost = 0;

// Loop over every product
foreach ($products as $product) {
    $productType = $product['product_type'];

    // If the product is a lamp or wallet...
    if ($productType == 'Lamp' || $productType == 'Wallet') {

        // Loop over the variants and add up their prices
        foreach ($product['variants'] as $productVariant) {
            $totalCost += $productVariant['price'];
        }
    }
}

return $totalCost;
```

So where do we start?

## Replace Conditional with Filter

Our goal is to take this one big `foreach` loop and figure out how we can break it down into a series of simple, independent, chainable steps.

The first thing that sticks out to me is this `if` statement:

```
$totalCost = 0;

foreach ($products as $product) {
    $productType = $product['product_type'];
```

```
    if ($productType == 'Lamp' || $productType == 'Wallet') {
        foreach ($product['variants'] as $productVariant) {
            $totalCost += $productVariant['price'];
        }
    }
}

return $totalCost;
```

We're looping over *all* of the products, but we only do any work if the product is a lamp or a wallet.

If we filter out the other products in advance, we can totally eliminate that conditional:

```
$lampsAndWallets = $products->filter(function ($product) {
    $productType = $product['product_type'];
    return $productType == 'Lamp' || $productType == 'Wallet';
});

$totalCost = 0;

foreach ($lampsAndWallets as $product) {
    foreach ($product['variants'] as $productVariant) {
        $totalCost += $productVariant['price'];
    }
}

return $totalCost;
```

## Replace || with Contains

See this line where we check if the `$productType` is a "Lamp" or a "Wallet"?

```
$lampsAndWallets = $products->filter(function ($product) {
    $productType = $product['product_type'];
    return $productType == 'Lamp' || $productType == 'Wallet';
});
```

A little trick I use to simplify comparisons like this is to use an `in_array` check:

```
$lampsAndWallets = $products->filter(function ($product) {
    return in_array($product['product_type'], ['Lamp', 'Wallet']);
});
```

Instead of checking if the product type is an $x$ or a $y$ or a $z$, `in_array` let's us say "here's a list of the product types we want, is this product in that list?"

The collection equivalent of `in_array` is `contains`, and it's a nice improvement because it removes any ambiguity about parameter order:

```
$lampsAndWallets = $products->filter(function ($product) {
    return collect(['Lamp', 'Wallet'])->contains($product['product_type']);
});
```

## Reduce to Sum

Looking at what we have now, what's the next thing we can break out into it's own simple step?

```
$lampsAndWallets = $products->filter(function ($product) {
    return collect(['Lamp', 'Wallet'])->contains($product['product_type']);
});

$totalCost = 0;

foreach ($lampsAndWallets as $product) {
    foreach ($product['variants'] as $productVariant) {
        $totalCost += $productVariant['price'];
    }
}

return $totalCost;
```

In highlighted code above, I can see at least two separate concerns:

1. Getting the price of each product variant.
2. Summing the prices to get a total cost.

If we split these up, we can use `reduce` to replace step 2:

```
$lampsAndWallets = $products->filter(function ($product) {
    return collect(['Lamp', 'Wallet'])->contains($product['product_type']);
});

// Get all of the product variant prices
$prices = collect();

foreach ($lampsAndWallets as $product) {
    foreach ($product['variants'] as $productVariant) {
        $prices[] = $productVariant['price'];
    }
}

// Sum the prices to get a total cost
$totalCost = $prices->reduce(function ($total, $price) {
    return $total + $price;
}, 0);

return $totalCost;
```

Remember what I said earlier about how you can often replace `reduce` with a more expressive operation? Here we can just use `sum` which turns our `reduce` call into one simple line:

```
$lampsAndWallets = $products->filter(function ($product) {
    return collect(['Lamp', 'Wallet'])->contains($product['product_type']);
});
```

```
// Get all of the product variant prices
$prices = collect();

foreach ($lampsAndWallets as $product) {
    foreach ($product['variants'] as $productVariant) {
        $prices[] = $productVariant['price'];
    }
}
return $prices->sum();
```

## Replace Nested Loop with FlatMap

So what about this chunk in the middle?

```
$lampsAndWallets = $products->filter(function ($product) {
    return collect(['Lamp', 'Wallet'])->contains($product['product_type']);
});

$prices = collect();

foreach ($lampsAndWallets as $product) {
    foreach ($product['variants'] as $productVariant) {
        $prices[] = $productVariant['price'];
    }
}

return $prices->sum();
```

It looks like we're trying to *map* product variants into their prices, but we're starting with a collection of products, not a collection of variants. So how can we build one big collection of variants to map into their prices?

One thing that gets us a little bit closer is to map each *product* into just its variants:

```
$variants = $lampsAndWallets->map(function ($product) {
    return $product['variants'];
});
```

The issue we have now is that we're stuck with a collection of *arrays* of variants, not just one big list of variants:

```
[
  //  ...
  [
    { "title": "Blue", "price": 29.33 },
    { "title": "Turquoise", "price": 18.50 }
  ],
  [
    { "title": "Sky Blue", "price": 20.00 }
  ],
  [
    { "title": "White", "price": 17.97 }
  ],
    { "title": "Azure", "price": 65.99 },
    { "title": "Salmon", "price": 1.66 }
  ],
  // ...
]
```

Fortunately for us, there's a method for this!

*Flatten* is a collection operation that flattens an arbitrarily deep collection to a single level. It takes a `$depth` parameter (defaulting to infinity) that's used to control how many levels it should flatten.

Since we only need to flatten one level, so we can flatten our product variant collection like so:

```
$variants = $lampsAndWallets->map(function ($product) {
    return $product['variants'];
})->flatten(1);
```

Using `map` and `flatten` together like this is so common that a lot of collection implementations offer single method called `flatMap` that combines them:

```
$variants = $lampsAndWallets->flatMap(function ($product) {
    return $product['variants'];
});
```

This gives us the flat collection of product variants we've been looking for:

```
[
  //  ...
  { "title": "Blue", "price": 29.33 },
  { "title": "Turquoise", "price": 18.50 },
  { "title": "Sky Blue", "price": 20.00 },
  { "title": "White", "price": 17.97 },
  { "title": "Azure", "price": 65.99 },
  { "title": "Salmon", "price": 1.66 },
  // ...
]
```

Now that we have all of the product variants in a single collection, we can use `map` to get their prices and get rid of both `foreach` loops:

```
$lampsAndWallets = $products->filter(function ($product) {
    return collect(['Lamp', 'Wallet'])->contains($product['product_type']);
});

$variants = $lampsAndWallets->flatMap(function ($product) {
    return $product['variants'];
});

$prices = $variants->map(function ($productVariant) {
    return $productVariant['price'];
});

return $prices->sum();
```

At this point, we can collapse this whole thing into a single pipeline:

```
return $products->filter(function ($product) {
    return collect(['Lamp', 'Wallet'])->contains($product['product_type']);
})->flatMap(function ($product) {
    return $product['variants'];
})->map(function ($productVariant) {
    return $productVariant['price'];
})->sum();
```

This is already pretty fantastic, but believe it or not we can still make this shorter.

## Plucking for Fun and Profit

Much like `flatMap` is a shortcut for mapping a collection and then flattening it by one, `pluck` is a shortcut for mapping a single field out of each element in a collection.

For example, if we had a collection of users and we needed to get their email addresses, we could write it like this using `map`:

```
$emails = $users->map(function ($user) {
    return $user['email'];
});
```

...or we could write it like this using `pluck`:

```
$emails = $users->pluck('email');
```

In our case, we can use `pluck` to get the prices of the product variants:

```
return $products->filter(function ($product) {
    return collect(['Lamp', 'Wallet'])->contains($product['product_type']);
})->flatMap(function ($product) {
    return $product['variants'];
})->pluck('price')->sum();
```

Let me also tell you a little secret about `sum`: It takes an optional parameter that works just like `pluck`!

So we can replace `->pluck('price')->sum()` with just `->sum('price')`, leaving us with this as our final solution:

```
return $products->filter(function ($product) {
    return collect(['Lamp', 'Wallet'])->contains($product['product_type']);
})->flatMap(function ($product) {
    return $product['variants'];
})->sum('price');
```

Not a single loop, conditional, or temporary variables to be found. Pretty elegant if you ask me!

## CSV Surgery 101

That last example was a bit of a monster, so here's one that's more of a quick tip.

Recently I was working on a project where I needed to import some data from a spreadsheet, and the customer had some data mushed together in a single column that I needed to extract.

The annoying part was that the data was a little inconsistent, with any given entry being in one of three formats:

1. `{DepartmentName}_{SupervisorName}_{ShiftId}`
2. `{DepartmentName}_{ShiftId}`
3. `{ShiftId}`

Thankfully all I needed was the `ShiftId` from each entry. So given a list of these strings, how can we get the `ShiftId` from each one?

Let's assume we're starting with an array that contains just the data from this column:

```
$shifts = [
    'Shipping_Steve_A7',
    'Sales_B9',
    'Support_Tara_K11',
    'J15',
    'Warehouse_B2',
    'Shipping_Dave_A6',
];
```

...and we want to end up with an array that looks like this:

```
$shiftIds = [
    'A7',
    'B9',
    'K11',
    'J15',
    'B2',
    'A6',
];
```

Since we're *transforming* every element in the first array into a corresponding element in the second array, I think we can safely say this is going to be some kind of `map` operation:

```
$shiftIds = collect($shifts)->map(function ($shift) {
    // How do we get the shift ID?
});
```

So what transformation do we need to apply to get that `ShiftId` piece?

The `ShiftId` always comes after the *last* underscore, so one approach would be to find the position of the last underscore, then grab the substring from that position onwards.

We can do that with `strrpos` to find the last underscore and `substr` to extract the substring:

```
$shiftIds = collect($shifts)->map(function ($shift) {
    $underscorePosition = strrpos($shift, '_');
    $substringOffset = $underscorePosition + 1;
    return substr($shift, $substringOffset);
});
```

If we run this against our data set above, we get this result:

```
$shifts = [
    'A7',
    'B9',
    'K11',
    '15',
    'B2',
    'A6',
];
```

Can you spot the error?

```
$shifts = [
    'A7',
    'B9',
    'K11',
    '15',
    'B2',
    'A6',
];
```

That line should be `J15`, but we lost the *J*!

The problem is that `strrpos` returns `false` if no underscore is found, and since we have to add 1 to the underscore position to grab the substring from the right offset, we end up truncating the first character of any shift that has no underscores:

```
$shift = 'J15';

$underscorePosition = strrpos($shift, '_');
// => false

$substringOffset = $underscorePosition + 1;
// => false + 1 == 1

$shiftId = substr($shift, $substringOffset);
// => 'J15' from position 1 to end == '15'
```

So if we're going to use this approach, we need to check for underscores first:

```
$shiftIds = collect($shifts)->map(function ($shift) {
    if (strrpos($shift, '_') !== false) {
        $underscorePosition = strrpos($shift, '_');
        $substringOffset = $underscorePosition + 1;
        return substr($shift, $substringOffset);
    } else {
        return $shift;
    }
});
```

I'm sorry but this code is gross. We can do better!

## Everything is Better as a Collection

It's really easy to make mistakes when you're keeping track of character offsets and dealing with substrings. Let's shift our thinking a little bit and solve this problem in a better way.

Remember the three different formats we started with?

1. `{DepartmentName}_{SupervisorName}_{ShiftId}`
2. `{DepartmentName}_{ShiftId}`
3. `{ShiftId}`

Each one of these formats is made up of some number of *parts*, separated by an underscore. We just want the *last part*.

Instead of looking for the last underscore, let's just split the string into its parts using `explode`:

```
$shiftIds = collect($shifts)->map(function ($shift) {
    // $shift => 'Shipping_Steve_A7'
    $parts = explode('_', $shift);
    // $parts => ['Shipping', 'Steve', 'A7']
});
```

The nice thing about `explode` is if you give it a string that doesn't contain the delimiter you specify, it just gives you your string back:

```
explode('_', ['J15']);
// => 'J15'
```

Since all we need now is the last element in the array, we *could* grab it using PHP's `end` function:

```
$shiftIds = collect($shifts)->map(function ($shift) {
    $parts = explode('_', $shift);
    return end($parts);
});
```

...or we could store the parts in a collection and use the `last` method:

```
$shiftIds = collect($shifts)->map(function ($shift) {
    return collect(explode('_', $shift))->last();
});
```

I like the collection version because we can do it one line and `last` is a bit more expressive than `end`.

Collections are a *great* tool for a lot of string processing situations. We'll cover another use case in the next example!

# Binary to Decimal

I've been learning Elixir *(the functional programming language)* through the exercises at exercism.io and one of the challenges was to take a string of binary and convert it to its decimal counterpart.

So given a string like `"100110101"`, we need to write a function that spits out `309`.

## A Quick Refresher

If you haven't done a ton of work with binary in the past, here's the 30 second crash course.

In decimal, or base 10, every column represents a power of 10, so in the number 3716, the 6 is in the ones column *($10^0$)*, the 1 is in the tens column *($10^1$)*, the 7 is in the hundreds column *($10^2$)*, and the 3 is in the thousands column *($10^3$)*.

That means 3716 is equivalent to $(3 \times 10^3) + (7 \times 10^2) + (1 \times 10^1) + (6 \times 10^0)$.

Binary works the same way except its a base 2 system, so every column represents a power of 2 instead of a power of 10.

So the binary number 11010 is the same as $(1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)$, or 26 in decimal.

## Using a For Loop

One way we could solve this would be to iterate over the characters in the binary string, convert each character to decimal, and add them together.

That solution might look something like this:

```
function binaryToDecimal($binary)
{
    $total = 0;
    $exponent = strlen($binary) - 1;

    for ($i = 0; $i < strlen($binary); $i++) {
        $decimal = $binary[$i] * (2 ** $exponent);
        $total += $decimal;
        $exponent--;
    }

    return $total;
}
```

This sort of code look familiar? Just the potential for off-by-one errors here is enough to make me run for the hills!

## Breaking It Down

The most important thing I want you to learn from this book is how to stop trying to do so many things at once and instead solve problems in small, simple steps.

Let's imagine for a second that we weren't allowed to use temporary variables to keep track of things off to the side. How can we solve this problem if we're only allowed to perform operations on the entire data set as a whole?

We can't do much useful work with the binary string directly, but what if we split the string into columns first? We can do that using `str_split`, and then wrap the result in a collection:

```
function binaryToDecimal($binary)
{
    // $binary => "11010"
    $columns = collect(str_split($binary));
    // $columns => ["1", "1", "0", "1", "0"]
}
```

Ok, so we've got a collection of the columns in our string. What can we do next that will get us a step closer to the solution?

Well in our original solution, we talked about converting or *transforming* the binary values to their decimal values, which sounds like something we could do with `map` right?

The problem is we need to know which exponent is associated with each column, but it looks like all we have is the column value itself.

There *is* one other piece of data we have hidden out of site here though: the column *keys*.

```
$columns = [
    0 => "1",
    1 => "1",
    2 => "0",
    3 => "1",
    4 => "0",
];
```

Those column keys do look like the exponents we need, but they're backwards. The first column has a key of `0` but we need it to have a key of `4`...

## Reversing the Collection

So how do we fix a backwards collection? We *reverse* it!

The Laravel Collection object has a `reverse` method, but it reverses the *keys* as well:

```
$columns->reverse();
// => [
//   4 => "0",
//   3 => "1",
//   2 => "0",
//   1 => "1",
//   0 => "1",
// ]
```

Thankfully we can get around this by using the `values` method on the reversed collection, which re-keys the collection back to a normal 0-based index:

```
function binaryToDecimal($binary)
{
    // $binary => "11010"
    $columns = collect(str_split($binary))
                ->reverse()
                ->values();
                // => [
                //   0 => "0",
                //   1 => "1",
                //   2 => "0",
                //   3 => "1",
                //   4 => "1",
                // ]
}
```

Now that we've got our columns matched up with their exponents, we're ready to convert each column to decimal.

## Mapping with Keys

So far any time we've used `map` we've only taken a single parameter in the callback, but the Laravel Collection actually gives us the *key* as the second parameter as well.

This makes it really easy for us to map the binary values to their corresponding decimal value:

```
function binaryToDecimal($binary)
{
    // $binary => "11010"
    $columns = collect(str_split($binary))
                ->reverse()
```

```
                ->values()
                ->map(function ($column, $exponent) {
                    return $column * (2 ** $exponent);
                });
                // => [0, 2, 0, 8, 16]
}
```

Once we've got all of the decimal values, we can calculate the total using `sum` like we did in the Shopify example:

```
function binaryToDecimal($binary)
{
    return collect(str_split($binary))
                ->reverse()
                ->values()
                ->map(function ($column, $exponent) {
                    return $column * (2 ** $exponent);
                })->sum();
}
```

The nicest thing about this refactoring to me is that there's no more temporary state. No taking some data and storing it off to the side to keep track of something while the code goes to do some other work.

The biggest problem with temporary variables is that they force you to hold the entire function in your head at all times to reason about how the function works.

Contrast that with our pipeline solution. Every single operation is entirely standalone. I don't need to understand the value of some temporary variable 6 lines up to know what `reverse` is really doing; it only depends on the output of the previous operation and nothing else. To me, that's elegance.

Alternatively, you could've just used PHP's built-in `bindec` function ;)

# 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:

```
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:

```
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:

```
$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:

```
$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:

```
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!

## Formatting a Pull Request Comment

That last example was pretty heavy, so here's another quick one.

I run a small SaaS application called Nitpick CI that looks for PSR-2 violations in GitHub pull requests and points them out by commenting on the offending lines.

A single line of code can have multiple PSR-2 violations, so at one point in the review process I need to take the violations for a given line and turn them into a single comment that I can post to GitHub.

The violation messages for one line start as a simple collection like this:

```
[
    'Opening brace must be the last content on the line',
    'Closing brace must be on a line by itself',
    'Each PHP statement must be on a line by itself',
];
```

GitHub comments support Markdown, and I'd like to display these messages as an unordered list, so from that array I need to generate this string:

```
"- Opening brace must be the last content on the line\n- Closing brace must
be on a line by itself\n- Each PHP statement must be on a line by itself\n"
```

Let's get to work!

## Concatenating in a Loop

Here's how we might've solved this problem using a procedural style:

```php
private function buildComment($messages)
{
    $comment = '';

    foreach ($messages as $message) {
        $comment .= "- {$message}\n";
    }

    return $comment;
}
```

## Map and Implode

...and here's how we can solve it using collections:

```php
private function buildComment($messages)
{
    return $messages->map(function ($message) {
        return "- {$message}\n";
    })->implode('');
}
```

Both solutions are pretty short and easy to follow, but I love that the collection solution is made up of single, independent operations.

In the concatenation-based solution, the function only makes sense as a whole, but with the pipeline approach, every step is a complete and encapsulated transformation that can stand on its own.

It's subtle, but as problems grow in complexity, being able to slice them up into discrete operations that each work on the entire data set at once really helps make code easier to understand.

## Stealing Mail

In a recent project, I needed to write some tests that made sure certain emails were being sent to the right email addresses.

To do this, I wrote my own fake in-memory mail implementation that would intercept and store emails, allowing me to inspect them and make assertions against them.

To make sure that emails were being sent to the correct people, I needed a method that took an email address as a parameter and checked to see if any of the sent messages contained that address in their recipients list.

```
public function test_new_users_are_sent_a_welcome_email()
{
    $mailer = new InMemoryMailer;
    Mail::swap($mailer);

    $this->post('register', [
        'name' => 'John Doe',
        'email' => 'john@example.com',
        'password' => 'secret',
    ]);

    $this->assertTrue($mailer->hasMessageFor('john@example.com'));
}
```

Here's the relevant bits of the fake mailer:

```
class InMemoryMailer
{
    private $messages = [];

    public function hasMessageFor($email)
    {
        // ???
    }

    // ...
}
```

For our purposes we'll say each message in the `$messages` array is just an associative array shaped like this:

```
$message = [
    'subject' => 'An example email subject!',
    'recipients' => ['jane@example.com', 'john@example.com', 'mary@example.co
    'body' => 'An example email body.',
];
```

So how could we implement the `hasMessageFor` method? Here's an imperative solution for us to use as a starting point:

```
public function hasMessageFor($email)
{
    foreach ($this->messages as $message) {
        foreach ($message['recipients'] as $recipient) {
            if ($recipient == $email) {
                return true;
            }
        }
    }

    return false;
}
```

## Replace Nested Check with Contains

The nested control structures in this solution have got to go. Let's tackle this nested `foreach` loop first:

```
public function hasMessageFor($email)
{
    foreach ($this->messages as $message) {
        foreach ($message['recipients'] as $recipient) {
            if ($recipient == $email) {
                return true;
            }
        }
    }

    return false;
}
```

All we're really trying to do here is check if `$email` is in the recipients list of the current message. Or put another way, does the recipients list *contain* the email address we're looking for?

Remember the `contains` method we touched on way back in the first example? If we wrap the recipients list in a collection, we can use `contains` here to get rid of the second loop:

```
public function hasMessageFor($email)
{
    foreach ($this->messages as $message) {
        if (collect($message['recipients'])->contains($email)) {
            return true;
        }
    }

    return false;
}
```

That's one loop gone, now what about the first one?

## Contains as a Higher Order Function

If I had to describe what we're really trying to do here, I'd say something like:

> *Check to see if our messages list contains a message with this email address in its recipients list.*

Up to this point we've used `contains` to look for a specific *value*, but checking to see if something contains "a message with this email address in its recipients list" is a bit more sophisticated than that.

Thankfully, `contains` is a lot more powerful than you might think! Not only can you pass `contains` a value to check for, you can also pass a *closure*.

When we pass a closure, we're saying *"check if the collection contains any items where this expression is true."* The closure gets called for every item in the collection, and takes the item *key* as its first parameter and the item *value* as its second parameter.

For example, we could check if a collection of names contained any names longer than 8 characters like so:

```
$names = collect(['Adam', 'Katharine', 'Jane', 'Steven']);

$names->contains(function ($i, $name) {
    return strlen($name) > 8;
});
// => true
```

Using this new super power, we can replace the loop that iterates over our `$messages` with a `contains` call that checks if any messages were sent to the specified email address:

```
public function hasMessageFor($email)
{
    return collect($this->messages)->contains(
        function ($i, $message) use ($email) {
            if (collect($message['recipients'])->contains($email)) {
                return true;
            }
            return false;
        }
    );
}
```

We can simplify the closure too, since we're just returning `true` or `false` based on the result of some condition:

```
  public function hasMessageFor($email)
  {
      return collect($this->messages)->contains(
          function ($i, $message) use ($email) {
-             if (collect($message['recipients'])->contains($email)) {
-                 return true;
-             }
-             return false;
+             return (collect($message['recipients'])->contains($email);
          }
      );
  }
```

Whenever possible, I try to make sure I'm working with a collection rather than an array as early in the process as possible, so since we control the `InMemoryMailer` class, I would just make `$messages` a collection from the beginning:

```
class InMemoryMailer
{
    private $messages;
```

```
    public function __construct()
    {
        $this->messages = new Collection;
    }

    // ...
}
```

That saves us a handful of characters in `hasMessageFor` and leaves us with this solution, completely free of loops and conditionals:

```
public function hasMessageFor($email)
{
    return $this->messages->contains(function ($i, $message) use ($email) {
        return collect($message['recipients'])->contains($email);
    });
}
```

For the sake of keeping the example short, we just used a simple associative array for each `$message`, but in the actual implementation I had a dedicated `Message` object that let me simplify the code a bit further:

```
public function hasMessageFor($email)
{
    return $this->messages->contains(function ($i, $message) use ($email) {
        return $message->hasRecipient($email);
    });
}
```

Inside the `Message` object, `hasRecipient` just contains the code we previously had inline:

```
class Message
{
    // ...
```

```
    public function hasRecipient($email)
    {
        return $this->recipients->contains($email);
    }
}
```

Much more declarative than our original loop-based solution, and much more expressive as a result!

## Choosing a Syntax Handler

Here's another example from Nitpick CI.

Nitpick focuses on checking PHP files for PSR-2 violations, but I built it to be extensible so that I could easily check other languages down the road.

To support this, the `Nitpicker` object that co-ordinates the bulk of the analysis holds on to a collection of *style checkers*. Any time a file needs to be checked for style violations, the `Nitpicker` looks up the appropriate style checker and asks it for any violations it finds in the file.

A style checker needs to provide two methods:

1. A `canCheck` method that takes a `ChangedFile` object, and returns true or false depending on whether that style checker can check that particular file.
2. A `check` method that takes a `ChangedFile` object, and returns a collection of `Violation` objects, representing style violations inside the file.

After a file is checked, I create a new `CheckedFile` object, passing the original file and the detected violations into the constructor.

If I don't happen to have a style checker that can check that file, I just create a new `CheckedFile` with an empty collection of violations.

Here's what checking a file looks like at a high level:

```
class Nitpicker
{
    // ...

    private function checkFile($file)
    {
        if ($this->hasCheckerFor($file)) {
            $checker = $this->getCheckerFor($file);
            $violations = $checker->check($file);
            return new CheckedFile($file, $violations);
        } else {
            return new CheckedFile($file, []);
        }
    }

    // ...
}
```

Let's look at how `hasCheckerFor` and `getCheckerFor` could be implemented.

## Looking for a Match

Before we can check a file, we need to make sure we actually have a checker capable of doing the checking, right? Here's how we might implement `hasCheckerFor` using an imperative approach:

```
private function hasCheckerFor($file)
{
    foreach ($this->checkers as $checker) {
        if ($checker->canCheck($file)) {
            return true;
        }
    }

    return false;
}
```

This is the same pattern we ran into in the "Finding an Email" example, and we can refactor it to use `contains` in the same way:

```
private function hasCheckerFor($file)
{
    return $this->checkers->contains(function ($i, $checker) use ($file)) {
        return $checker->canCheck($file);
    }
}
```

Instead of looping over all of the checkers manually to see if there's any that can do the check, we just ask:

*"Does our collection of checkers contain a checker that can check this file?"*

Great! Now how about actually retrieving the checker we need?

## Getting the Right Checker

Using an imperative style, getting the matching checker looks a lot like checking to see if we have a matching checker in the first place:

```
private function getCheckerFor($file)
{
    foreach ($this->checkers as $checker) {
        if ($checker->canCheck($file)) {
            return $checker;
        }
    }
}
```

Instead of returning `true` or `false` when we find a matching checker, we just return the `$checker` itself.

So how can we refactor this using collection methods?

## Replace Iteration with First

The collection class has a `first` method that returns the first element in the collection:

```
collect([1, 2, 3])->first();
// => 1
```

Okay, so what? *Well*, just like `contains`, `first` can also take a closure.

When you pass a closure to `first`, you're saying "give me the first element where this callback returns true."

For example, here's how we could find the first name in a collection that starts with the letter B:

```
$names = collect(['Adam', 'Tracy', 'Ben', 'Beatrice', 'Kyle']);
$names->first(function ($i, $name) {
    return $name[0] == 'B';
});
// => 'Ben'
```

Just like `contains`, the callback takes the element key as the first parameter, and the value as the second parameter.

So using `first`, we can refactor `getCheckerFor` to look like this:

```
private function getCheckerFor($file)
{
    return $this->checkers->first(function ($i, $checker) use ($file) {
        return $checker->canCheck($file);
    });
}
```

Thinking of this variation of `first` as more like a "first where" makes this a pretty expressive solution.

## A Hidden Rule

Something that bothers me about this code is that we have a bit of an unwritten rule in place.

Our `getCheckerFor` method assumes there's always a matching checker in that list, because it's expected we never call it unless we call `hasCheckerFor` first to verify there's a match. This is a bit of a code smell to me, so what can we do about it?

By default, `first` will return `null` if it doesn't find a match. Our imperative solution was doing the same thing, since PHP will always return `null` from a function if it never hits an explicit return.

Letting `null` leak into your code can result in a lot of annoying issues that are difficult to debug. One approach that's a bit better would be to throw an exception if a match isn't found, so we know about it right away.

Here's how you might think to write that with our current implementation:

```
private function getCheckerFor($file)
{
    $checker = $this->checkers->first(function ($i, $checker) use ($file) {
        return $checker->canCheck($file);
    });

    if ($checker === null) {
        throw new Exception("No matching style checker found!");
    }

    return $checker;
}
```

This works, but is there a way we can do it in a more "Tell Don't Ask" style? Certainly!

## Providing a Default

Just like the `get` method we talked about in the GitHub score example, `first` lets you specify a default value to use if no match is found.

For example, say we wanted to find the first name in a collection that starts with a B or default to 'Bryan' if none are found:

```
$names = collect(['Adam', 'Tracy', 'Kyle']);
$names->first(function ($i, $name) {
    return $name[0] == 'B';
}, 'Bryan');
// => 'Bryan'
```

*I have no idea why anyone would want to do this but it's the simplest example I could come up with.*

This sounds helpful right? But wait, we don't actually have a default value to provide, we want to throw an exception!

Thankfully, the collection has us covered. If we pass *another closure* as the default value, that closure will be invoked if and only if a match isn't found, meaning we can rewrite our function like this:

```
private function getCheckerFor($file)
{
    return $this->checkers->first(function ($i, $checker) use ($file) {
        return $checker->canCheck($file);
    }, function () {
        throw new Exception("No matching style checker found!");
    });
}
```

Kind of a cool trick! But I still don't really like this whole "throw an exception" solution.

If we think about the "Tell Don't Ask" principle some more, doesn't having to call `hasCheckerFor` in the first place seem a lot like asking instead of telling? On

top of that, we're basically duplicating effort, because `hasCheckerFor` does all the work `getCheckerFor` needs to do to find the checker, but only returns `true` or `false` instead of giving us the checker back.

Is there any way we can "tell" and maybe remove that duplication of effort?

## The Null Object Pattern

One refactoring you can use to replace checks like this is to introduce a *Null Object*.

A null object is an object you can use in place of a `null` value that acts like the object you really need, but has neutral or no behavior. Put another way, it's an object that has all the methods you need, but none of those methods actually do anything.

Now, "doing nothing" is context-dependent, so a null object can't just have empty methods. You need to figure out what "do nothing" means in your situation and build that into your null object.

For example, I gave a talk once about refactoring some code that dealt with applying coupons to orders in an e-commerce system.

Coupons all have a `discount` method that returns how much an order should be discounted when using that coupon. At one point in the talk, I introduce a `NullCoupon` that always discounts an order by zero dollars, effectively never giving a discount.

It looked something like this:

```
class NullCoupon
{
    public function discount($order)
    {
        return 0;
    }
}
```

This is what I mean when I say you need to figure out what "do nothing" means in your context.

So what would introducing a null object look like in our context?

## The Null Checker

Let's look at the code we started with again, highlighting what happens when we do have a matching checker vs. what happens when we don't:

```
private function checkFile($file)
{
    if ($this->hasCheckerFor($file)) {
        $checker = $this->getCheckerFor($file);
        $violations = $checker->check($file);
        return new CheckedFile($file, $violations);
    } else {
        return new CheckedFile($file, []);
    }
}
```

The only difference here is that when we don't have a matching checker, we just pass in an empty array instead of an array of violations.

So if we were to create a `NullChecker`, it would just need to return an empty array instead of an array of violations any time we asked it to check a file:

```
class NullChecker
{
    public function check($file)
    {
        return [];
    }
}
```

Taking small steps, let's see how this lets us refactor this code.

First, let's use the `NullChecker` to get the empty array instead of hard coding it in the `else` clause:

```
private function checkFile($file)
{
    if ($this->hasCheckerFor($file)) {
        $checker = $this->getCheckerFor($file);
        $violations = $checker->check($file);
        return new CheckedFile($file, $violations);
    } else {
        $checker = new NullChecker;
        $violations = $checker->check($file);
        return new CheckedFile($file, $violations);
    }
}
```

This is actually more code than we had before, but now both sides of the `if` statement look awfully similar don't they? What if we updated `getCheckerFor` to give us a `NullChecker` if it couldn't find a match?

```
private function getCheckerFor($file)
{
    return $this->checkers->first(function ($i, $checker) use ($file) {
        return $checker->canCheck($file);
    }, new NullChecker);
}
```

This means there's *always* going to be a match now right? So as an intermediate step, let's update `hasCheckerFor` to just return `true`:

```
  private function hasCheckerFor($file)
  {
-     return $this->checkers->contains(function ($i, $checker) use ($file)) {
-         return $checker->canCheck($file);
-     }
+     return true;
  }
```

Now we can eliminate the `else` clause, giving us this:

```
private function checkFile($file)
{
    if ($this->hasCheckerFor($file)) {
        $checker = $this->getCheckerFor($file);
        $violations = $checker->check($file);
        return new CheckedFile($file, $violations);
    }
}
```

Since `hasCheckerFor` just returns `true`, we can actually eliminate the conditional altogether:

```
private function checkFile($file)
{
    $checker = $this->getCheckerFor($file);
    $violations = $checker->check($file);
    return new CheckedFile($file, $violations);
}
```

Now we're actually not calling `hasCheckerFor` at all, so we can delete that entire method, leaving us with this final solution:

```
class Nitpicker
{
    // ...

    private function checkFile($file)
    {
        $violations = $this->getCheckerFor($file)->check($file);
        return new CheckedFile($file, $violations);
    }
```

```
    private function getCheckerFor($file)
    {
        return $this->checkers->first(function ($i, $checker) use ($file) {
            return $checker->canCheck($file);
        }, new NullChecker);
    }
}
```

No more loops, conditionals, or exceptions! Just simple, concise, expressive code. Benign defaults are a powerful tool, use them!

## Tagging on the Fly

Imagine you are building a blog engine, and you want to be able to add tags to your blog posts.

A nice UI pattern for something like this is to use an autocomplete multiselect box like Select2 or similar, and allow someone to add tags from an existing list *or* create new ones on the fly.

This can get a little tricky on the server side though, because the request might contain mixed data.

For this example, we'll say that existing tags come through as IDs, and new tags come through as just a tag name:

```
[
    'tags' => [
        17,
        32,
        'recipes',
        11,
        'kitchen'
    ]
]
```

Using a dedicated `PostTagsController` to manage tags for a post, here's one way we could write the `update` action *(ignoring validation for brevity):*

```
class PostTagsController
{
    public function update($postId)
    {
        $post = Post::find($postId);

        $tagIds = [];

        foreach (request('tags') as $nameOrId) {
            if (is_numeric($nameOrId)) {
                $tagIds[] = $nameOrId;
            } else {
                $tag = Tag::create(['name' => $nameOrId]);
                $tagIds[] = $tag->id;
            }
        }

        $post->tags()->sync($tagIds);

        return view('posts.index');
    }
}
```

## Extracting the Loop

So how can we simplify this using collections? The first thing I want to do is extract this loop into a separate function:

```
class PostTagsController
{
    public function update($postId)
    {
        $post = Post::find($postId);

        $tagIds = [];
```

```
        foreach (request('tags') as $nameOrId) {
            if (is_numeric($nameOrId)) {
                $tagIds[] = $nameOrId;
            } else {
                $tag = Tag::create(['name' => $nameOrId]);
                $tagIds[] = $tag->id;
            }
        }

        $post->tags()->sync($tagIds);

        return view('posts.index');
    }
}
```

To extract this into a function, we need a good name.

So what is this block of code trying to do? To me, it looks like the job of this code is to take a list of mixed tag IDs and tag names and normalize that list into *just* tag IDs.

Let's extract a function called `normalizeTagsToIds`:

```
class PostTagsController
{
    public function update($postId)
    {
        $post = Post::find($postId);

        $tagIds = $this->normalizeTagsToIds(request('tags'));
        $post->tags()->sync($tagIds);

        return view('posts.index');
    }
```

```
    private function normalizeTagsToIds($tags)
    {
        $tagIds = [];

        foreach ($tags as $nameOrId) {
            if (is_numeric($nameOrId)) {
                $tagIds[] = $nameOrId;
            } else {
                $tag = Tag::create(['name' => $nameOrId]);
                $tagIds[] = $tag->id;
            }
        }

        return $tagIds;
    }
}
```

## Normalizing with Map

Looking at the code above, do you see any of the patterns we talked about in the first part of the book?

How about now?

```
private function normalizeTagsToIds($tags)
{
    $tagIds = [];

    foreach ($tags as $nameOrId) {
        if (is_numeric($nameOrId)) {
            $tagId = $nameOrId;
        } else {
            $tag = Tag::create(['name' => $nameOrId]);
            $tagId = $tag->id;
        }
```

```
        $tagIds[] = $id;
    }

    return $tagIds;
}
```

Aside from the conditional, this is just a standard `map` operation! And there's nothing stopping us from using an `if` statement inside a map:

```
private function normalizeTagsToIds($tags)
{
    return collect($tags)->map(function ($nameOrId) {
        if (is_numeric($nameOrId)) {
            return $nameOrId;
        }
        return Tag::create(['name' => $nameOrId])->id;
    })->all();
}
```

A lot of the time when I have a function meant to operate on a collection like `normalizeTagsToIds`, I also create a function for operating on the individual item to break up the code a bit more. Here's what the whole thing would look like:

```
class PostTagsController
{
    public function update($postId)
    {
        $post = Post::find($postId);

        $tagIds = $this->normalizeTagsToIds(request('tags'));
        $post->tags()->sync($tagIds);

        return view('posts.index');
    }
```

```
    private function normalizeTagsToIds($tags)
    {
        return collect($tags)->map(function ($nameOrId) {
            return $this->normalizeTagToId($nameOrId);
        });
    }

    private function normalizeTagToId($nameOrId)
    {
        if (is_numeric($nameOrId)) {
            return $nameOrId;
        }
        return Tag::create(['name' => $nameOrId])->id;
    }
}
```

Until now we've only used `map` to perform the exact same transformation on every item in a collection, but it can be a really useful tool in situations like this as well where you need to normalize a collection with some rough edges into a consistent data set.

## Nitpicking a Pull Request

I was working on a new Nitpick feature recently where I wanted to post a comment like "Code style looks great, nice job!" if a pull request was opened and no style violations were detected.

Before we get into adding that functionality, let's walk through the existing code.

Here's what it looks like to nitpick a pull request:

```
public function nitpick($pullRequest)
{
    $pullRequest->changedFiles()->flatMap(function ($changedFile) {
        return $this->checkFile($changedFile)->comments();
    })->reject(function ($comment) use ($pullRequest) {
        return $comment->isDuplicate($pullRequest);
    })->each(function ($comment) use ($pullRequest) {
        $pullRequest->postComment($comment);
    });
}
```

Pretty cool that the whole thing is just one big collection pipeline, huh? Let's break down what's happening here:

1. Given a pull request, get a collection of the files that changed in that PR.
2. Check each of those changed files for violations, and return a collection of style comments for each file.
3. Collapse those collections of comments into one flat collection of comments.
4. Since a PR is re-analyzed every time it's updated, reject any comments that have already been posted on the PR.
5. Post each comment on the pull request.

Of course there's more complexity hidden in the methods we call along the way in this pipeline, but it's really cool to me how well this demonstrates what you can do by just transforming data; thinking of your applications as just a big function that takes some input and produces some output.

## A Fork in the Code

In the existing code, after figuring out which comments need to be posted, we just post them all unconditionally. Conveniently, if there's no comments to post, the callback in `each` just never runs, so we never have to worry about the empty case. An empty collection is kind of like a Null Object in cases like this; pretty cool!

But now that we want to post a different comment if the collection is empty, we need to introduce a conditional. If we want our code to stay readable, we also need to introduce a temporary variable:

```
public function nitpick($pullRequest)
{
    $comments = $pullRequest->changedFiles()
        ->flatMap(function ($changedFile) {
            return $this->checkFile($changedFile)->comments();
        })->reject(function ($comment) use ($pullRequest) {
            return $comment->isDuplicate($pullRequest);
        });

    if ($comments->isEmpty()) {
        $pullRequest->postNoViolationsComment();
    } else {
        $comments->each(function ($comment) use ($pullRequest) {
            $pullRequest->postComment($comment);
        });
    }
}
```

Is this the worst thing in the world? Probably not, but I'll be damned if it wouldn't be cool to be able to write this code as one continuous pipeline.

## Learning from Smalltalk

Smalltalk is one of the earliest truly object-oriented programming languages. In Smalltalk, everything is an object, and every control structure is implemented as messages sent to those objects.

This has some interesting implications, namely that Smalltalk has no `if` statements!

So how do you write conditional code in Smalltalk? Well remember what I said about how everything in Smalltalk is an object? Even `true` and `false` are objects; instances of the `True` and `False` classes which both extend `Boolean`.

So whenever you need to write an `if` statement in Smalltalk, you do it by calling the `ifTrue` method on a `Boolean` instance, and pass it a block of code you'd like it to run if the boolean is true.

The Smalltalk syntax can seem very alien at first, so here's what it would look like if PHP worked the same way:

```
$boolean = true;

$boolean->ifTrue(function () {
    // do something
});
```

Smalltalk also has an `ifFalse` method, and `ifTrue` and `ifFalse` both return the original object, so they can even be chained together like so:

```
$boolean = true;

$boolean->ifTrue(function () {
    // do something
})->ifFalse(function () {
    // do something else
});
```

A nice side effect of using method calls for conditionals like this is that you can chain them after *other* methods as well. For example, say we had this code for throwing some sort of authorization error if a user didn't have the necessary permissions for some action:

```
if (! $user->isAdmin()) {
    throw new AuthorizationException;
}
```

Using the Smalltalk style, we could rewrite that like this:

```
$user->isAdmin()->ifFalse(function () {
    throw new AuthorizationException;
});
```

Starting to get an idea of where we're going with this?

## Collection Macros

One cool thing about the Laravel Collection class is that it's *macroable*.

Laravel's `support` package includes a trait called `Macroable` that allows you to add methods to a class at run time.

For example, here's how we could define a new method on the collection called `odd` that only returns the items at odd positions in the collection:

```
Collection::macro('odd', function () {
    return $this->values()->filter(function ($value, $i) {
        return $i % 2 !== 0;
    });
});

collect([0, 1, 2, 3, 4, 5])->odd();
// => [1, 3, 5]
```

When I'm working in a Laravel app, I keep all of my collection macros in a service provider like this:

```
class CollectionExtensions extends ServiceProvider
{
    public function boot()
    {
        Collection::macro('odd', function () {
            return $this->values()->filter(function ($value, $i) {
                return $i % 2 !== 0;
            });
        });

        // ...and any other lovely macros you'd like to add.
    }
```

```
    public function register()
    {
        // ...
    }
}
```

So inspired by Smalltalk and powered by macros, let's get our damn pipeline back!

## Chainable Conditions

In our situation, we have two cases to cover:

1. If the collection is empty, post an "all good!" comment.
2. If the collection has any comments, post those comments.

Let's macro in a few methods to cover these cases.

First let's create a method called `ifEmpty` that will execute it's closure only if the collection is empty. Here's what that would look like:

```
Collection::macro('ifEmpty', function ($callback) {
    if ($this->empty()) {
        $callback();
    }
    return $this;
});
```

All we do is run the callback if the collection is empty, then return the collection so we can continue to chain if necessary.

Next let's add a method that runs if the collection is *not* empty, or put another way, if it has *any* items.

We'll call this one `ifAny`, and we'll pass the collection into the callback the user provides so they have access to the collection if they need it:

```
Collection::macro('ifAny', function ($callback) {
    if (! $this->empty()) {
        $callback($this);
    }
    return $this;
});
```

Replacing our conditional with these new methods, we get this intermediate step:

```
public function nitpick($pullRequest)
{
    $comments = $pullRequest->changedFiles()->flatMap(function ($changedFile)
        return $this->checkFile($changedFile)->comments();
    })->reject(function ($comment) use ($pullRequest) {
        return $comment->isDuplicate($pullRequest);
    });

    $comments->ifEmpty(function ()  use ($pullRequest) {
        $pullRequest->postNoViolationsComment();
    });

    $comments->ifAny(function ($comments) use ($pullRequest) {
        $comments->each(function ($comment) use ($pullRequest)  {
            $pullRequest->postComment($comment);
        });
    });
}
```

Of course, now these operations are all chainable, so we can collapse this down into a single pipeline and remove the temporary variable entirely:

```
public function nitpick($pullRequest)
{
    $pullRequest->changedFiles()->flatMap(function ($changedFile) {
        return $this->checkFile($changedFile)->comments();
    })->reject(function ($comment) use ($pullRequest) {
        return $comment->isDuplicate($pullRequest);
    })->ifAny(function ($comments) use ($pullRequest) {
        $comments->each(function ($comment) use ($pullRequest) {
            $pullRequest->postComment($comment);
        });
    })->ifEmpty(function () use ($pullRequest) {
        $pullRequest->postNoViolationsComment();
    });
}
```

I'd also add a new function to the `$pullRequest` object that can post multiple comments, so we can get rid of that extra level of indentation:

```
public function nitpick($pullRequest)
{
    $pullRequest->changedFiles()->flatMap(function ($changedFile) {
        return $this->checkFile($changedFile)->comments();
    })->reject(function ($comment) use ($pullRequest) {
        return $comment->isDuplicate($pullRequest);
    })->ifAny(function ($comments) use ($pullRequest) {
        $pullRequest->postComments($comments);
    })->ifEmpty(function () use ($pullRequest) {
        $pullRequest->postNoViolationsComment();
    });
}
```

If you ask me, this is a pretty interesting way to write this code. Is it always the best solution? Probably not, but the idea of conditions as methods is pretty fascinating, and I think it opens up a lot of possibilities.

Keep it in your back pocket and play with it when it makes sense. And if you think this is cool, go study Smalltalk! :)

## Comparing Monthly Revenue

So far we've only worked with problems that started with a single collection of items, but collection pipelines can be useful in other cases as well.

Say we were asked to generate a report that compared revenue from every month this year to revenue from every month last year.

Given last year's monthly revenue and this year's monthly revenue, like so:

```
$lastYear = [
    2976.50,  // Jan
    2788.84,  // Feb
    2353.92,  // Mar
    3365.36,  // Apr
    2532.99,  // May
    1598.42,  // Jun
    2751.82,  // Jul
    2576.17,  // Aug
    2324.87,  // Sep
    2299.21,  // Oct
    3483.10,  // Nov
    2245.08,  // Dec
];

$thisYear = [
    3461.77,
    3665.17,
    3210.53,
    3529.07,
    3376.66,
    3825.49,
    2165.24,
    2261.40,
    3988.76,
    3302.42,
    3345.41,
    2904.80,
];
```

...we need to write a function that takes those collections, and spits out one collection showing the delta for each month.

```
compare_revenue($thisYear, $lastYear);
// => [
//      485.27,
//      876.33,
//      856.61,
//      163.71,
//      843.67,
//     2227.07,
//     -586.58,
//     -314.77,
//     1663.89,
//     1003.21,
//     -137.69,
//      659.72,
// ];
```

## Matching on Index

Here's how we might solve this problem using a `foreach` loop:

```
function compare_revenue($thisYear, $lastYear)
{
    $deltas = [];

    foreach ($lastYear as $month => $monthlyRevenue) {
        $deltas[] = $thisYear[$month] - $monthlyRevenue;
    }

    return $deltas;
}
```

This is pretty short and it works, but some things just feel off about it. For example, why iterate over `$lastYear` instead of `$thisYear`? There's no real reason for it, it's just arbitrary.

It feels like we've just arranged some code that happens to work, instead of trying to model what we're trying to do in a meaningful, expressive way.

Let's try and write this using a more declarative style.

## Zipping Things Together

I'd like to introduce you to an operation called `zip`.

`zip` lets you take one collection, and pair every element in that collection with the corresponding element in another collection.

For example, here we're zipping `[1, 2, 3]` with `['a', 'b', 'c']` to produce a new collection of pairs:

```
collect([1, 2, 3])->zip(['a', 'b', 'c']);
// => [
//   [1, 'a'],
//   [2, 'b'],
//   [3, 'c'],
// ];
```

Think of each collection as being one side of a zipper on a jacket. When we zip the two sides together, each tooth on the first side is paired up with a tooth from the second side.

## Using Zip to Compare

As you might have guessed, `zip` is really handy when you need to compare corresponding values between two collections.

If we zip our two years of revenue together, we get this:

```
collect($thisYear)->zip($lastYear);
// => [
//   [2976.50, 3461.77],
//   [2788.84, 3665.17],
//   [2353.92, 3210.53],
//   [3365.36, 3529.07],
//   [2532.99, 3376.66],
//   [1598.42, 3825.49],
//   [2751.82, 2165.24],
//   [2576.17, 2261.40],
//   [2324.87, 3988.76],
//   [2299.21, 3302.42],
//   [3483.10, 3345.41],
//   [2245.08, 2904.80],
// ];
```

Now that we have each corresponding month grouped in pairs, we can `map` those pairs into their deltas, giving us this solution:

```
function compare_revenue($thisYear, $lastYear)
{
    return collect($thisYear)->zip($lastYear)->map(function ($thisAndLast) {
        return $thisAndLast[0] - $thisAndLast[1];
    });
}
```

Like always, we were able to solve this problem using collection pipelines by trying to break it down into small, discrete steps.

`zip` was a tricky one to find use cases for when I first learned it, but nowadays I run into opportunities to use it all the time. Pay attention to situations where you want to loop over two arrays at once, `zip` is the secret to solving those problems with a collection pipeline.

# Transposing Form Input

Dealing with arrays in form submissions is a pain in the ass.

Imagine you need to build a page that allows users to add multiple contacts at once. If a contact has a `name`, `email`, and `occupation`, ideally the incoming request would look something like this:

```
[
    'contacts' => [
        [
            'name' => 'Jane',
            'occupation' => 'Doctor',
            'email' => 'jane@example.com',
        ],
        [
            'name' => 'Bob',
            'occupation' => 'Plumber',
            'email' => 'bob@example.com',
        ],
        [
            'name' => 'Mary',
            'occupation' => 'Dentist',
            'email' => 'mary@example.com',
        ],
    ],
];
```

The problem is that crafting a form that actually submits this format is surprisingly complicated.

If you haven't had to do this before, you might think you can get away with something like this, using just a pinch of JavaScript to duplicate the form fields while keeping all of the field names the same:

```
<form method="POST" action="/contacts">
    <div>
        <label>
            Name
            <input name="contacts[][name]">
        </label>
        <label>
            Email
            <input name="contacts[][email]">
        </label>
        <label>
            Occupation
            <input name="contacts[][occupation]">
        </label>
    </div>

    <!-- Adds another set of form fields using JavaScript -->
    <button type="button">Add another contact</button>

    <button type="submit">Save contacts</button>
</form>
```

...but this gives you a request that looks like this:

```
[
    'contacts' => [
        [ 'name' => 'Jane' ],
        [ 'occupation' => 'Doctor' ],
        [ 'email' => 'jane@example.com' ],
        [ 'name' => 'Bob' ],
        [ 'occupation' => 'Plumber' ],
        [ 'email' => 'bob@example.com' ],
        [ 'name' => 'Mary' ],
        [ 'occupation' => 'Dentist' ],
        [ 'email' => 'mary@example.com' ],
    ],
];
```

To get the form to submit in the correct format, you need to give each set of fields an explicit index:

```
<form method="POST" action="/contacts">
    <div>
        <label>
            Name
            <input name="contacts[0][names]">
        </label>
        <label>
            Email
            <input name="contacts[0][emails]">
        </label>
        <label>
            Occupation
            <input name="contacts[0][occupations]">
        </label>
    </div>

    <!-- Adds another set of form fields using JavaScript -->
    <button type="button">Add another contact</button>

    <button type="submit">Save contacts</button>
</form>
```

...which means that when you add another set of fields, you need to change the name of every input, incrementing the index by one.

Doesn't seem too unreasonable at first, just count the sets of fields and add one for the new set right?

*Wrong!* What if a user removes a set of fields? Or two sets of fields? Now there might only be 3 sets remaining but the last set still has an index of 4, so just counting the fields is going to result in a collision.

So what can you do? Well, you have a few options:

1. Parse out the index from the last set of fields and add one to that number whenever you add new fields.

2. Keep track of the index as state in your JavaScript.
3. Throw away *all* of the indexes and recalculate them every time you add or remove a set of fields.

All of a sudden this seems like a lot more work on the front-end than you signed up for! But there's one other option:

*Submit the data in a different format and deal with it on the server.*

As long as we aren't nesting *past* the empty square brackets, PHP is happy to let us leave out the index. So what you'll commonly see people do in this situation *(and what you may have done yourself)* is name the form fields like this:

```
<form method="POST" action="/contacts">
    <div>
        <label>
            Name
            <input name="names[]">
        </label>
        <label>
            Email
            <input name="emails[]">
        </label>
        <label>
            Occupation
            <input name="occupations[]">
        </label>
    </div>

    <!-- Adds another set of form fields using JavaScript -->
    <button type="button">Add another contact</button>

    <button type="submit">Save contacts</button>
</form>
```

The benefit of course is that now we don't have to keep track of the index. We can reuse the same markup for every set of fields, never worrying about

the total number of fields in the form, or what happens when a set of fields is removed. *Excellent!*

The disadvantage is that now our incoming request looks like this:

```
[
    'names' => [
        'Jane',
        'Bob',
        'Mary',
    ],
    'emails' => [
        'jane@example.com',
        'bob@example.com',
        'mary@example.com',
    ],
    'occupations' => [
        'Doctor',
        'Plumber',
        'Dentist',
    ],
];
```

*Ruh-roh!*

## Quick and Dirty

We need to get these contacts out of the request and into our system. Say we want our controller action to take this general form:

```
public function store()
{
    $contacts = /* Build the contacts using the request data */;

    Auth::user()->contacts()->saveMany($contacts);

    return redirect()->home();
}
```

How can we translate our request data into actual `Contact` objects? An imperative solution might look something like this:

```
public function store(Request $request)
{
    $contacts = [];

    $names = $request->get('names');
    $emails = $request->get('emails');
    $occupations = $request->get('occupations');

    foreach ($names as $i => $name) {
        $contacts[] = new Contact([
            'name' => $name,
            'email' => $emails[$i],
            'occupation' => $occupations[$i],
        ]);
    }

    Auth::user()->contacts()->saveMany($contacts);

    return redirect()->home();
}
```

First, we grab the names, emails, and occupations from the request. Then we arbitrarily iterate over one of them *(the names in this case)*, pull out the other fields we need by matching up the index, and create our `Contact` objects.

There's certainly nothing *wrong* with this approach, I mean, it works, right? But we're breaking the golden rule, and those temporary variables are bugging me. Can we refactor this into a series of independent transformations that don't rely on temporary state? Let's see!

## Identifying a Need

First things first, let's get our request data into a collection.

```
public function store(Request $request)
{
    $requestData = collect($request->only('names', 'emails', 'occupations'));

    // ...
}
```

This pulls `names`, `emails`, and `occupations` out into a new collection, which is about the best starting point we're going to get from that form submission.

Next, we need to somehow get our `Contact` objects out of this collection.

```
public function store(Request $request)
{
    $requestData = collect($request->only('names', 'emails', 'occupations'));

    $contacts = $requestData->/* ??? */;

    // ...
}
```

Typically when we have a collection of data and we need to transform each element into something new, we use `map`.

But in order to map our contact data into `Contact` objects, we need each element in our collection to contain the `name`, `email`, and `occupation` for a single contact. Right now, the first element in our array is *all* of the `names`, the second element is all `emails`, and the last element is all `occupations`.

So before we can use `map`, we need some mystery function to get our data into the right structure.

```
public function store(Request $request)
{
    $requestData = collect($request->only('names', 'emails', 'occupations'));
```

```
    $contacts = $requestData->/*

        Mystery operation!

    */->map(function ($contactData) {
        return new Contact([
            'name' => $contactData['name'],
            'email' => $contactData['email'],
            'occupation' => $contactData['occupation'],
        ]);
    });

    // ...
}
```

## Introducing Transpose

*Transpose* is an often overlooked list operation that I first noticed in Ruby.

The goal of `transpose` is to rotate a multidimensional array, turning the rows into columns and the columns into rows.

Say we had this array:

```
$before = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
];
```

If we transpose that array, `[1, 2, 3]` becomes the first *column* rather than the first row, `[4, 5, 6]` becomes the second column, and `[7, 8, 9]` becomes the last column.

```
$after = [
    [1, 4, 7],
    [2, 5, 8],
    [3, 6, 9],
];
```

Let's look at our incoming request again:

```
[
    'names' => [
        'Jane',
        'Bob',
        'Mary',
    ],
    'emails' => [
        'jane@example.com',
        'bob@example.com',
        'mary@example.com',
    ],
    'occupations' => [
        'Doctor',
        'Plumber',
        'Dentist',
    ],
];
```

If we get rid of the keys, we're left with a multidimensional array that looks like this:

```
[
    ['Jane', 'Bob', 'Mary'],
    ['jane@example.com', 'bob@example.com', 'mary@example.com'],
    ['Doctor', 'Plumber', 'Dentist'],
];
```

I wonder what happens if we `transpose` that array?

```
[
    ['Jane', 'jane@example.com', 'Doctor'],
    ['Bob', 'bob@example.com', 'Plumber'],
    ['Mary', 'mary@example.com', 'Dentist'],
];
```

*Whoa!* This looks pretty close to the structure we wanted in first place, albeit without the keys. We can work with this!

## Implementing Transpose

Laravel's `Collection` class doesn't implement `transpose` out of the box, but since collections are macroable, we can add it at runtime.

Here's what a basic implementation looks like:

```
Collection::macro('transpose', function () {
    $items = array_map(function (...$items) {
        return $items;
    }, ...$this->values());

    return new static($items);
});
```

## Transpose in Practice

Now that we've found our mystery function, we can finish off our controller action:

```
public function store(Request $request)
{
    $requestData = collect($request->only('names', 'emails', 'occupations'));
```

```
    $contacts = $requestData->transpose()->map(function ($contactData) {
        return new Contact([
            'name' => $contactData[0],
            'email' => $contactData[1],
            'occupation' => $contactData[2],
        ]);
    });

    Auth::user()->contacts()->saveMany($contacts);

    return redirect()->home();
}
```

We can even collapse this down further, turning just about the whole action into a single chain:

```
public function store(Request $request)
{
    collect($request->only([
        'names',
        'emails',
        'occupations'
    ]))->transpose()->map(function ($contactData) {
        return new Contact([
            'name' => $contactData[0],
            'email' => $contactData[1],
            'occupation' => $contactData[2],
        ]);
    })->each(function ($contact) {
        Auth::user()->contacts()->save($contact);
    });

    return redirect()->home();
}
```

Now instead of being deep in the details worrying about looping over a data set and matching up keys between different arrays, we're operating on the entire data set at once, using a more declarative style at a higher level of abstraction.

# Ranking a Competition

Here's one I ran into on a client project a few months ago. I was working on an app for managing competitions and I needed to write some code for ranking how the teams did after a competition was finished.

I started with a collection of team scores that looked something like this:

```
$scores = collect([
    ['score' => 76, 'team' => 'A'],
    ['score' => 62, 'team' => 'B'],
    ['score' => 82, 'team' => 'C'],
    ['score' => 86, 'team' => 'D'],
    ['score' => 91, 'team' => 'E'],
    ['score' => 67, 'team' => 'F'],
    ['score' => 67, 'team' => 'G'],
    ['score' => 82, 'team' => 'H'],
]);
```

So what does it mean to rank these scores? At first, it might seem as simple as just sorting them in reverse order by score and calling it a day.

That's easy enough using the `sortByDesc` method, which takes the name of the field to sort by as a parameter:

```
$rankedScores = $scores->sortByDesc('score');
// => [
//     ['score' => 91, 'team' => 'E'],
//     ['score' => 86, 'team' => 'D'],
//     ['score' => 82, 'team' => 'C'],
//     ['score' => 82, 'team' => 'H'],
//     ['score' => 76, 'team' => 'A'],
//     ['score' => 67, 'team' => 'F'],
//     ['score' => 67, 'team' => 'G'],
//     ['score' => 62, 'team' => 'B'],
// ];
```

Now that they are in order, we can just use the array index + 1 as the rank right? Not quite, because `sortByDesc` actually maintains the old keys, so despite our `$rankedScores` being in the correct order, they still have explicit keys that don't match the expected ranking:

```
$rankedScores = $scores->sortByDesc('score');
// => [
//     4 => ['score' => 91, 'team' => 'E'],
//     3 => ['score' => 86, 'team' => 'D'],
//     2 => ['score' => 82, 'team' => 'C'],
//     7 => ['score' => 82, 'team' => 'H'],
//     0 => ['score' => 76, 'team' => 'A'],
//     5 => ['score' => 67, 'team' => 'F'],
//     6 => ['score' => 67, 'team' => 'G'],
//     1 => ['score' => 62, 'team' => 'B'],
// ];
```

One way to fix this is to call the `values` method on the collection, which removes any explicit keys and resets them back to normal:

```
$rankedScores = $scores->sortByDesc('score')->values();
// => [
//     0 => ['score' => 91, 'team' => 'E'],
//     1 => ['score' => 86, 'team' => 'D'],
//     2 => ['score' => 82, 'team' => 'C'],
//     3 => ['score' => 82, 'team' => 'H'],
//     4 => ['score' => 76, 'team' => 'A'],
//     5 => ['score' => 67, 'team' => 'F'],
//     6 => ['score' => 67, 'team' => 'G'],
//     7 => ['score' => 62, 'team' => 'B'],
// ];
```

That's a bit better, but our actual rankings are still off by one right? I think it would be better if we could add an explicit `rank` field to each score that held the actual rank number, starting from 1 instead of 0.

## Zipping-in the Ranks

One way to do this is to `zip` the scores with a list of ranks.

We can generate the list of ranks using PHP's `range($start, $end)` function to create an array starting at `1` and ending at `$scores->count()`:

```
$rankedScores = $scores->sortByDesc('score')
    ->zip(range(1, $scores->count()));
// => [
//     [['score' => 91, 'team' => 'E'], 1],
//     [['score' => 86, 'team' => 'D'], 2],
//     [['score' => 82, 'team' => 'C'], 3],
//     [['score' => 82, 'team' => 'H'], 4],
//     [['score' => 76, 'team' => 'A'], 5],
//     [['score' => 67, 'team' => 'F'], 6],
//     [['score' => 67, 'team' => 'G'], 7],
//     [['score' => 62, 'team' => 'B'], 8],
// ];
```

A nice side effect of using this approach is that we can drop the `values` call, since we don't really need to worry about the keys anymore.

After zipping the scores with their ranks, we can use `map` to turn the rank into an actual field in each score:

```
$rankedScores = $scores->sortByDesc('score')
    ->zip(range(1, $scores->count()))
    ->map(function ($scoreAndRank) {
        list($score, $rank) = $scoreAndRank;
        return array_merge($score, [
            'rank' => $rank
        ]);
    });
```

```
// => [
//     ['rank' => 1, 'score' => 91, 'team' => 'E'],
//     ['rank' => 2, 'score' => 86, 'team' => 'D'],
//     ['rank' => 3, 'score' => 82, 'team' => 'C'],
//     ['rank' => 4, 'score' => 82, 'team' => 'H'],
//     ['rank' => 5, 'score' => 76, 'team' => 'A'],
//     ['rank' => 6, 'score' => 67, 'team' => 'F'],
//     ['rank' => 7, 'score' => 67, 'team' => 'G'],
//     ['rank' => 8, 'score' => 62, 'team' => 'B'],
// ];
```

Nice! Time to call it a day right? Not just yet...

## Dealing with Ties

If you look closely at our ranked scores, you'll notice there's actually two sets of ties:

```
[
    ['rank' => 1, 'score' => 91, 'team' => 'E'],
    ['rank' => 2, 'score' => 86, 'team' => 'D'],
    ['rank' => 3, 'score' => 82, 'team' => 'C'],
    ['rank' => 4, 'score' => 82, 'team' => 'H'],
    ['rank' => 5, 'score' => 76, 'team' => 'A'],
    ['rank' => 6, 'score' => 67, 'team' => 'F'],
    ['rank' => 7, 'score' => 67, 'team' => 'G'],
    ['rank' => 8, 'score' => 62, 'team' => 'B'],
];
```

Is it really fair that team C gets third place and team H gets fourth place, even though they have the same score? Why not the other way around?

The way this is handled in standard competition ranking is to give tied scores the same rank and skip the ranks those scores would've got otherwise. Sounds sort of confusing, but really it's pretty intuitive when you see it.

Here's what our scores would look like adjusted for standard competition ranking:

```
[
    ['rank' => 1, 'score' => 91, 'team' => 'E'],
    ['rank' => 2, 'score' => 86, 'team' => 'D'],
    ['rank' => 3, 'score' => 82, 'team' => 'C'],
    ['rank' => 3, 'score' => 82, 'team' => 'H'],
    ['rank' => 5, 'score' => 76, 'team' => 'A'],
    ['rank' => 6, 'score' => 67, 'team' => 'F'],
    ['rank' => 6, 'score' => 67, 'team' => 'G'],
    ['rank' => 8, 'score' => 62, 'team' => 'B'],
];
```

Notice that teams C and H both get third place now, but we skip fourth place, so team A is still in fifth as they were before.

Ok sure sounds reasonable, but how on earth do we implement this?!

## One Step at a Time

I have a confession to make:

A lot of the time when I'm solving problems with collection pipelines, *I have no idea what the solution is going to be before I start programming.*

One of the nicest things about collection pipelines is that each step is small and discrete. We've talked a bit about how that makes code easier to follow, but it also makes code easier to write.

Instead of having to figure out the whole algorithm in advance, I only ever have to worry about getting one step closer to the solution than I am right now. If I do that enough times, eventually I end up at the solution.

So what could we try to get us one step closer to implementing standard competition ranking?

## Grouping by Score

If teams with the same score are all supposed to get the same rank, would grouping the results by their score get us any closer to a solution? Let's give it a shot.

We can group items in a collection using the `groupBy` method. `groupBy` takes a closure as a parameter, and groups items based on the return value of that closure.

For example, say we wanted to group a list of names by length. We would pass a closure that returns the length of each name, like so:

```
$names = collect(['Adam', 'Bryan', 'Jane', 'Dan', 'Kayla']);
$names->groupBy(function ($name) {
    return strlen($name);
});
```

This would give us a collection that looked like this:

```
[
    4 => ['Adam', 'Jane'],
    5 => ['Bryan', 'Kayla'],
    3 => ['Dan'],
]
```

So we can group our results by score by just returning the score of each result:

```
$rankedScores = $scores->sortByDesc('score')
    ->zip(range(1, $scores->count()))
    ->map(function ($scoreAndRank) {
        list($score, $rank) = $scoreAndRank;
        return array_merge($score, [
            'rank' => $rank
        ]);
    })
```

```
    ->groupBy(function ($rankedScore) {
        return $rankedScore['score'];
    });
```

Conveniently, if you're just grouping by an object property or associative array field, you can also just pass a string to `groupBy`, telling it which field to use:

```
$rankedScores = $scores->sortByDesc('score')
    ->zip(range(1, $scores->count()))
    ->map(function ($scoreAndRank) {
        list($score, $rank) = $scoreAndRank;
        return array_merge($score, [
            'rank' => $rank
        ]);
    })
    ->groupBy('score');
```

In our case, that gives us a collection of grouped scores that looks like this:

```
[
    91 => [
        ['rank' => 1, 'score' => 91, 'team' => 'E']
    ],
    86 => [
        ['rank' => 2, 'score' => 86, 'team' => 'D']
    ],
    82 => [
        ['rank' => 3, 'score' => 82, 'team' => 'C'],
        ['rank' => 4, 'score' => 82, 'team' => 'H'],
    ],
    76 => [
        ['rank' => 5, 'score' => 76, 'team' => 'A']
    ],
```

```
    67 => [
        ['rank' => 6, 'score' => 67, 'team' => 'F'],
        ['rank' => 7, 'score' => 67, 'team' => 'G'],
    ],
    62 => [
        ['rank' => 8, 'score' => 62, 'team' => 'B']
    ],
];
```

What next?

## Adjusting the Ranks

So we have all of our results grouped by score, and we want to make sure any teams that have the same score tie for the best possible rank. Let's look at one of the ties and see if we can think of a way to do this.

```
$tiedScores = collect([
    ['rank' => 3, 'score' => 82, 'team' => 'C'],
    ['rank' => 4, 'score' => 82, 'team' => 'H'],
]);
```

Given this group of results, how could we make sure both teams tie for third place?

First we need to find the best rank in group. We can do that by using `pluck` to get a collection of every rank, then using the `min` function to find the lowest rank in the collection:

```
$lowestRank = $tiedScores->pluck('rank')->min();
```

Easy enough! Now we just need to assign that same rank to each team. We can do that using `map` to transform each result:

```
$lowestRank = $tiedScores->pluck('rank')->min();

$adjustedScores = $tiedScores->map(function ($rankedScore) use ($lowestRank)
    $rankedScore['rank'] = $lowestRank;
    return $rankedScore;
})
```

We have to be a bit careful here because we're not supposed to mutate inside `map` remember? In this case, changing the `rank` key doesn't technically mutate anything outside of the closure because arrays in PHP are passed by value, but if we were working with objects this would be a big no-no.

For the sake of consistency I would recommend returning a *new* array here anyways, using `array_merge` to replace the old rank with the new one:

```
  $lowestRank = $tiedScores->pluck('rank')->min();

  $adjustedScores = $tiedScores->map(function ($rankedScore) use ($lowestRank
-     $rankedScore['rank'] = $lowestRank;
-     return $rankedScore;
+     return array_merge($rankedScore, [
+         'rank' => $lowestRank
+     ]);
  })
```

After applying this transformation, we're left with a set of scores that looks like this:

```
[
    ['rank' => 3, 'score' => 82, 'team' => 'C'],
    ['rank' => 3, 'score' => 82, 'team' => 'H'],
];
```

Now both teams are tied for third place like we wanted, perfect!

To apply this transformation to *every* group of scores, we just need to `map` each group through this transformation:

```
$rankedScores = $scores->sortByDesc('score')
    ->zip(range(1, $scores->count()))
    ->map(function ($scoreAndRank) {
        list($score, $rank) = $scoreAndRank;
        return array_merge($score, [
            'rank' => $rank
        ]);
    })
    ->groupBy('score')
    ->map(function ($tiedScores) {
        $lowestRank = $tiedScores->pluck('rank')->min();

        return $tiedScores->map(function ($rankedScore) use ($lowestRank) {
            return array_merge($rankedScore, [
                'rank' => $lowestRank
            ]);
        });
    });
```

Now we're left with a collection of adjusted rankings, where any ties are given the same rank:

```
[
    91 => [
        ['rank' => 1, 'score' => 91, 'team' => 'E']
    ],
    86 => [
        ['rank' => 2, 'score' => 86, 'team' => 'D']
    ],
    82 => [
        ['rank' => 3, 'score' => 82, 'team' => 'C'],
        ['rank' => 3, 'score' => 82, 'team' => 'H'],
    ],
    76 => [
        ['rank' => 5, 'score' => 76, 'team' => 'A']
    ],
```

```
    67 => [
        ['rank' => 6, 'score' => 67, 'team' => 'F'],
        ['rank' => 6, 'score' => 67, 'team' => 'G'],
    ],
    62 => [
        ['rank' => 8, 'score' => 62, 'team' => 'B']
    ],
];
```

This is starting to look pretty close to a solution!

## Collapse and Sort

Right now our results are still grouped by score. We can flatten them down using `collapse`, like we've seen in earlier examples:

```
$rankedScores = $scores->sortByDesc('score')
    ->zip(range(1, $scores->count()))
    ->map(function ($scoreAndRank) {
        list($score, $rank) = $scoreAndRank;
        return array_merge($score, [
            'rank' => $rank
        ]);
    })
    ->groupBy('score')
    ->map(function ($tiedScores) {
        $lowestRank = $tiedScores->pluck('rank')->min();

        return $tiedScores->map(function ($rankedScore) use ($lowestRank) {
            return array_merge($rankedScore, [
                'rank' => $lowestRank
            ]);
        });
    })
    ->collapse();
```

This leaves us with a flat collection of scores, properly ranked using standard competition ranking:

```
[
    ['rank' => 1, 'score' => 91, 'team' => 'E'],
    ['rank' => 2, 'score' => 86, 'team' => 'D'],
    ['rank' => 3, 'score' => 82, 'team' => 'C'],
    ['rank' => 3, 'score' => 82, 'team' => 'H'],
    ['rank' => 5, 'score' => 76, 'team' => 'A'],
    ['rank' => 6, 'score' => 67, 'team' => 'F'],
    ['rank' => 6, 'score' => 67, 'team' => 'G'],
    ['rank' => 8, 'score' => 62, 'team' => 'B'],
];
```

Awesome, we have the result we were looking for!

Despite this answer being correct at this point, I would still recommend finishing this pipeline by sorting it by rank:

```
$rankedScores = $scores->sortByDesc('score')
    ->zip(range(1, $scores->count()))
    ->map(function ($scoreAndRank) {
        list($score, $rank) = $scoreAndRank;
        return array_merge($score, [
            'rank' => $rank
        ]);
    })
    ->groupBy('score')
    ->map(function ($tiedScores) {
        $lowestRank = $tiedScores->pluck('rank')->min();

        return $tiedScores->map(function ($rankedScore) use ($lowestRank) {
            return array_merge($rankedScore, [
                'rank' => $lowestRank
            ]);
        });
    })
    ->collapse();
    ->sortBy('rank');
```

The reason I recommend this is that up to this point, the results were only really in the correct order by chance. The `groupBy` method happened to keep our results ordered by score even after grouping, but this isn't really a guarantee that `groupBy` makes; it's just a coincidence that the grouping algorithm happens to return the results in that order.

Explicitly sorting by rank at the end makes sure that our code will continue to work if the grouping algorithm ever changes, so in my opinion it's definitely worth the extra operation.

## Cleaning Up

Here's what this behemoth looks like if we stuff it into a function:

```
function rank_scores($scores)
{
    return collect($scores)
        ->sortByDesc('score')
        ->zip(range(1, $scores->count()))
        ->map(function ($scoreAndRank) {
            list($score, $rank) = $scoreAndRank;
            return array_merge($score, [
                'rank' => $rank
            ]);
        })
        ->groupBy('score')
        ->map(function ($tiedScores) {
            $lowestRank = $tiedScores->pluck('rank')->min();

            return $tiedScores->map(function ($rankedScore) use ($lowestRank)
                return array_merge($rankedScore, [
                    'rank' => $lowestRank
                ]);
            });
        })
        ->collapse()
        ->sortBy('rank');
}
```

While I promise this is still easier to read than any procedural solution I was able to come up with, it's still pretty grim.

Looking at this function, the first thing I'd like to extract is this second `map` call, responsible for assigning the same rank to each tied score:

```
function rank_scores($scores)
{
    return collect($scores)
        ->sortByDesc('score')
        ->zip(range(1, $scores->count()))
        ->map(function ($scoreAndRank) {
            list($score, $rank) = $scoreAndRank;
            return array_merge($score, [
                'rank' => $rank
            ]);
        })
        ->groupBy('score')
        ->map(function ($tiedScores) {
            $lowestRank = $tiedScores->pluck('rank')->min();

            return $tiedScores->map(function ($rankedScore) use ($lowestRank)
                return array_merge($rankedScore, [
                    'rank' => $lowestRank
                ]);
            });
        })
        ->collapse()
        ->sortBy('rank');
}
```

Let's pull the body of that out into a separate function called `apply_min_rank`:

```
function rank_scores($scores)
{
    return collect($scores)
        ->sortByDesc('score')
        ->zip(range(1, $scores->count()))
        ->map(function ($scoreAndRank) {
            list($score, $rank) = $scoreAndRank;
            return array_merge($score, [
                'rank' => $rank
            ]);
        })
        ->groupBy('score')
        ->map(function ($tiedScores) {
            return apply_min_rank($tiedScores);
        })
        ->collapse()
        ->sortBy('rank');
}

function apply_min_rank($tiedScores)
{
    $lowestRank = $tiedScores->pluck('rank')->min();

    return $tiedScores->map(function ($rankedScore) use ($lowestRank) {
        return array_merge($rankedScore, [
            'rank' => $lowestRank
        ]);
    });
}
```

That's a *little* better. But even so, this still isn't that expressive, and now there's not really anything meaningful to extract.

Or is there?

## Grouping Operations

Looking at our `rank_scores` function, there's a few steps that stand out to me as being part of something bigger.

For example, together these three steps are used to *assign the initial rankings* to the scores:

```
function rank_scores($scores)
{
    return collect($scores)
        ->sortByDesc('score')
        ->zip(range(1, $scores->count()))
        ->map(function ($scoreAndRank) {
            list($score, $rank) = $scoreAndRank;
            return array_merge($score, [
                'rank' => $rank
            ]);
        })
        ->groupBy('score')
        ->map(function ($tiedScores) {
            return apply_min_rank($tiedScores);
        })
        ->collapse()
        ->sortBy('rank');
}
```

Similarly, these three steps are used to *adjust the rankings for tied scores:*

```
function rank_scores($scores)
{
    return collect($scores)
        ->sortByDesc('score')
        ->zip(range(1, $scores->count()))
```

```
        ->map(function ($scoreAndRank) {
            list($score, $rank) = $scoreAndRank;
            return array_merge($score, [
                'rank' => $rank
            ]);
        })
        ->groupBy('score')
        ->map(function ($tiedScores) {
            return apply_min_rank($tiedScores);
        })
        ->collapse()
        ->sortBy('rank');
}
```

What would it look like to extract these groups of operations into their own functions?

## Breaking the Chain

Here's what an `assign_initial_rankings` function would look like:

```
function assign_initial_rankings($scores)
{
    return $scores->sortByDesc('score')
        ->zip(range(1, $scores->count()))
        ->map(function ($scoreAndRank) {
            list($score, $rank) = $scoreAndRank;
            return array_merge($score, [
                'rank' => $rank
            ]);
        });
}
```

...and here's what `adjust_rankings_for_ties` would look like:

```
function adjust_rankings_for_ties($scores)
{
    return $scores->groupBy('score')
        ->map(function ($tiedScores) {
            return apply_min_rank($tiedScores);
        })
        ->collapse();
}
```

Both are simple enough to understand when they are broken out like this, but how do we integrate these into our pipeline? It turns out we can't, we'd have to break the pipeline and use intermediate variables, like this:

```
function rank_scores($scores)
{
    $rankedScores = assign_initial_rankings(collect($scores));
    $adjustedScores = adjust_rankings_for_ties($rankedScores);
    return $adjustedScores->sortBy('rank');
}
```

We could add these functions as methods on our collection using macros, but they don't really make sense as collection methods. Both of these methods are very domain-specific, and they don't really belong with the rest of the general purpose collection operations.

For a long time, I just accepted this limitation and broke out of the pipeline when I needed to, but recently I stumbled on a pattern that gives me the best of both worlds.

## The Pipe Macro

Check out this very simple little macro:

```
Collection::macro('pipe', function ($callback) {
    return $callback($this);
});
```

All it does is define a method called `pipe` that takes a callback, passes the collection into the callback, and returns the result.

Brutally simple, but look at what it lets us do:

```
function rank_scores($scores)
{
    return collect($scores)
        ->pipe(function ($scores) {
            return assign_initial_rankings($scores);
        })
        ->pipe(function ($rankedScores) {
            return adjust_rankings_for_ties($rankedScores);
        })
        ->sortBy('rank');
}
```

We've got our pipeline back! Now it's extremely easy to "pipe" our collection through domain specific transformations, saving us from reverting back to temporary variables, or bloating up the collection class with domain-specific macros.

PHP lets you treat a string as a callback if it matches the name of a function, so we can even do this:

```
function rank_scores($scores)
{
    return collect($scores)
        ->pipe('assign_initial_rankings')
        ->pipe('adjust_rankings_for_ties')
        ->sortBy('rank');
}
```

Doesn't get much more expressive than that.

Here's the whole thing for completeness' sake:

```
function rank_scores($scores)
{
    return collect($scores)
        ->pipe('assign_initial_rankings')
        ->pipe('adjust_rankings_for_ties')
        ->sortBy('rank');
}

function assign_initial_rankings($scores)
{
    return $scores->sortByDesc('score')
        ->zip(range(1, $scores->count()))
        ->map(function ($scoreAndRank) {
            list($score, $rank) = $scoreAndRank;
            return array_merge($score, [
                'rank' => $rank
            ]);
        });
}

function adjust_rankings_for_ties($scores)
{
    return $scores->groupBy('score')->map(function ($tiedScores) {
        return apply_min_rank($tiedScores);
    })->collapse();
}

function apply_min_rank($tiedScores)
{
    $lowestRank = $tiedScores->pluck('rank')->min();
    return $tiedScores->map(function ($rankedScore) use ($lowestRank) {
        return array_merge($rankedScore, [
            'rank' => $lowestRank
        ]);
    });
}
```

Not bad for a pretty complicated problem!

# Afterword

For many years, I used anonymous functions in my code when interacting with various libraries and didn't think much of it. jQuery needs a callback to run after I make an AJAX request? Sure, but I never really understood why I would ever use them in *my* code. What benefits could anonymous functions really give *me?*

It wasn't until I first learned how to use `array_map` that the power of anonymous functions really started to click for me. I couldn't believe how many times I had used that pattern in the past and never once thought about how it could be abstracted! If there had been a function for doing that all along, what else was I missing?

After map I learned filter, then reduce, and at that point I was hooked. Once I combined this with collection objects to allow the pipeline style we've covered in this book, I felt like a whole new world of solutions had opened up to me.

It's hard for me to put in to words, but there's something beautiful about taking some data, piping it through a series of discrete transformations, and having the solution come out on the other side. There's something clean and pure about it that makes it an extremely seductive style of programming.

To this day, every time I encounter a new problem the first thing I think is "how can I solve this with a collection pipeline?", and I am *continually amazed* by just how often I can use this approach to find an elegant solution.

If you're excited by the ideas in this book, the best way to get comfortable with them is to practice in your own code. Every time you want to write a loop, force yourself to solve the problem with a collection. *Never* write a `foreach` loop ever again.

You'll be amazed what you can do with this style of programming if you push it hard enough. I still haven't found the limit myself.

– Adam Wathan, May 2016