drupol/collection Documentation

Release 1.0.0

Pol Dellaiera

Oct 17, 2020

Contents

Collection is a functional utility library for PHP greater than 7.1.3.

It's similar to other available collection libraries based on regular PHP arrays, but with a lazy mechanism under the hood that strives to do as little work as possible while being as flexible as possible.

Functions like array_map(), array_filter() and array_reduce() are great, but they create new arrays and everything is eagerly done before going to the next step. Lazy collection leverages PHP's generators, iterators and yield statements to allow you to work with very large data sets while keeping memory usage as low as possible.

For example, imagine your application needs to process a multi-gigabyte log file while taking advantage of this library's methods to parse the logs. Instead of reading the entire file into memory at once, this library may be used to keep only a small part of the file in memory at a given time.

On top of this, this library:

- is immutable,
- is extendable,
- leverages the power of PHP generators and iterators,
- uses S.O.L.I.D. principles,
- does not have any external dependency,
- fully tested,
- type safe (*type safe* @ > *98%*),
- framework agnostic.

Except a few methods, most methods are pure and return a new Collection object.

Also, unlike regular PHP arrays where keys must be either of type *int* or *string*, this collection library let you use any kind of type for keys: integer, string, objects, arrays, ... anything! This library could be a valid replacement for SplObjectStorage but with much more features. This way of working opens up new perspectives and another way of handling data, in a more functional way.

And last but not least, collection keys are preserved throughout most operations, and it might be leading to some confusions, carefully read this example for the full explanation.

This library has been inspired by:

- Laravel Support Package
- DusanKasan/Knapsack
- mtdowling/transducers
- Ruby Array
- Collect.js
- nikic/iter
- Lazy.js

CHAPTER 1

Requirements

1.1 PHP

PHP greater than 7.1.3 is required.

1.2 Dependencies

No dependency is required.

CHAPTER 2

Installation

The easiest way to install it is through Composer

```
composer require loophp/collection
```

CHAPTER 3

Usage

Find here some working examples.

3.1 Simple

```php
<?php

declare(strict_types=1);

include __DIR__ . '/../../../vendor/autoload.php';

use loophp\collection\Collection;

$collection = Collection::fromIterable(['A', 'B', 'C', 'D', 'E']);

// Get the result as an array.
$collection
    ->all(); // ['A', 'B', 'C', 'D', 'E']

// Get the first item.
$collection
    ->first(); // ['a']

// Get the first item.
$collection
    ->first()
    ->current(); // 'a'

// Append items.
$collection
    ->append('F', 'G', 'H')
    ->normalize()
    ->all(); // ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
```

(continues on next page)

(continued from previous page)

```php
// Prepend items.
$collection
    ->prepend('1', '2', '3')
    ->normalize()
    ->all(); // ['1', '2', '3', 'A', 'B', 'C', 'D', 'E']

// Split a collection into chunks of a given size.
$collection
    ->chunk(2)
    ->all(); // [['A', 'B'], ['C', 'D'], ['E']]

// Merge items.
$collection
    ->merge([1, 2], [3, 4], [5, 6])
    ->all(); // ['A', 'B', 'C', 'D', 'E', 1, 2, 3, 4, 5, 6]

// Map data
$collection
    ->map(
        static function ($value, $key) {
            return sprintf('%s.%s', $value, $value);
        }
    )
    ->all(); // ['A.A', 'B.B', 'C.C', 'D.D', 'E.E']

// Use \StdClass as input
$data = (object) array_combine(range('A', 'E'), range('A', 'E'));

// Keys are preserved during the map() operation.
Collection::fromIterable(['A' => 'A', 'B' => 'B', 'C' => 'C', 'D' => 'D', 'E' => 'E'])
    ->map(
        static function (string $value, string $key): string {
            return mb_strtolower($value);
        }
    )
    ->all(); // ['A' => 'a', B => 'b', 'C' => 'c', 'D' = >'d', 'E' => 'e']

// Tail
Collection::fromIterable(range('a', 'z'))
    ->tail(3)
    ->all(); // [23 => 'x', 24 => 'y', 25 => 'z']

// Reverse
Collection::fromIterable(range('a', 'z'))
    ->tail(4)
    ->reverse()
    ->all(); // [25 => 'z', 24 => 'y', 23 => 'x', 22 => 'w']

// Flip operation.
// array_flip() can be used in PHP to remove duplicates from an array.(dedup-licate␣
↪an array)
// See: https://www.php.net/manual/en/function.array-flip.php
// Example:
// $dedupArray = array_flip(array_flip(['a', 'b', 'c', 'd', 'a'])); // ['a', 'b', 'c',
↪ 'd']
// However, in loophp/collection it doesn't behave as such.
```

(continues on next page)

(continued from previous page)

```php
// As this library is based on PHP Generators, it's able to return multiple times the same key when iterating.
// You end up with the following result when issuing twice the ::flip() operation.
Collection::fromIterable(['a', 'b', 'c', 'd', 'a'])
    ->flip()
    ->flip()
    ->all(); // ['a', 'b', 'c', 'd', 'a']

// Get the Cartesian product.
Collection::fromIterable(['a', 'b'])
    ->product([1, 2])
    ->all(); // [['a', 1], ['a', 2], ['b', 1], ['b', 2]]

// Infinitely loop over numbers, cube them, filter those that are not divisible by 5, take the first 100 of them.
Collection::range(0, \INF)
    ->map(
        static function ($value, $key) {
            return $value ** 3;
        }
    )
    ->filter(
        static function ($value, $key) {
            return $value % 5;
        }
    )
    ->limit(100)
    ->all(); // [1, 8, 27, ..., 1815848, 1860867, 1906624]

// Apply a callback to the values without altering the original object.
// If the callback returns false, then it will stop.
Collection::fromIterable(range('A', 'Z'))
    ->apply(
        static function ($value, $key) {
            echo mb_strtolower($value);

            return true;
        }
    );

// Generate 300 distinct random numbers between 0 and 1000
$random = static function () {
    return mt_rand() / mt_getrandmax();
};

Collection::unfold($random)
    ->map(
        static function ($value) {
            return floor($value * 1000) + 1;
        }
    )
    ->distinct()
    ->limit(300)
    ->normalize()
    ->all();

// Fibonacci using the static method ::unfold()
```

(continues on next page)

(continued from previous page)

```php
$fibonacci = static function ($a = 0, $b = 1): array {
    return [$b, $b + $a];
};

Collection::unfold($fibonacci)
    // Get the first item of each result.
    ->pluck(0)
    // Limit the amount of results to 10.
    ->limit(10)
    // Convert to regular array.
    ->all(); // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

Collection::unfold($fibonacci)
    ->map(
        static function (array $value, $key) {
            return $value[1] / $value[0];
        }
    )
    ->limit(100)
    ->last(); // 1.6180339887499

// Use an existing Generator as input data.
$readFileLineByLine = static function (string $filepath): Generator {
    $fh = fopen($filepath, 'rb');

    while (false !== $line = fgets($fh)) {
        yield $line;
    }

    fclose($fh);
};

$hugeFile = __DIR__ . '/vendor/composer/autoload_static.php';

Collection::fromIterable($readFileLineByLine($hugeFile))
    // Add the line number at the end of the line, as comment.
    ->map(
        static function ($value, $key) {
            return str_replace(\PHP_EOL, ' // line ' . $key . \PHP_EOL, $value);
        }
    )
    // Find public static fields or methods among the results.
    ->filter(
        static function ($value, $key) {
            return false !== mb_strpos(trim($value), 'public static');
        }
    )
    // Drop the first result.
    ->drop(1)
    // Limit to 3 results only.
    ->limit(3)
    // Implode into a string.
    ->implode();

// Load a string
$string = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  Quisque feugiat tincidunt sodales.
```

(continues on next page)

(continued from previous page)

```
  Donec ut laoreet lectus, quis mollis nisl.
  Aliquam maximus, orci vel placerat dapibus, libero erat aliquet nibh, nec imperdiet␣
↪felis dui quis est.
  Vestibulum non ante sit amet neque tincidunt porta et sit amet neque.
  In a tempor ipsum. Duis scelerisque libero sit amet enim pretium pulvinar.
  Duis vitae lorem convallis, egestas mauris at, sollicitudin sem.
  Fusce molestie rutrum faucibus.';

// By default will have the same behavior as str_split().
Collection::fromString($string)
    ->explode(' ')
    ->count(); // 71

// Or add a separator if needed, same behavior as explode().
Collection::fromString($string, ',')
    ->count(); // 9

// Regular values normalization.
Collection::fromIterable([0, 2, 4, 6, 8, 10])
    ->scale(0, 10)
    ->all(); // [0, 0.2, 0.4, 0.6, 0.8, 1]

// Logarithmic values normalization.
Collection::fromIterable([0, 2, 4, 6, 8, 10])
    ->scale(0, 10, 5, 15, 3)
    ->all(); // [5, 8.01, 11.02, 12.78, 14.03, 15]

// Fun with function convergence.
// Iterator over the function: f(x) = r * x * (1-x)
// Change that parameter $r to see different behavior.
// More on this: https://en.wikipedia.org/wiki/Logistic_map
$function = static function ($x = .3, $r = 2) {
    return $r * $x * (1 - $x);
};

Collection::unfold($function)
    ->map(static function ($value) {
        return round($value, 2);
    })
    ->limit(10)
    ->all(); // [0.42, 0.48, 0.49, 0.49, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]

// Infinitely loop over a collection
Collection::fromIterable(['A', 'B', 'C'])
    ->cycle();

// Traverse the collection using windows of a given size.
Collection::fromIterable(range('a', 'z'))
    ->window(3)
    ->all(); // [['a'], ['a', 'b'], ['a', 'b', 'c'], ['b', 'c', 'd'], ['c', 'd', 'e'],
↪ ...]

Collection::fromIterable(range('a', 'd'))
    ->wrap()
    ->all(); // [['a'], ['b'], ['c'], ['d']]

Collection::fromIterable([['a'], ['b'], ['c'], ['d']])
```

(continues on next page)

(continued from previous page)

```
    ->unwrap()
    ->all(); // ['a', 'b', 'c', 'd']
```

3.2 Advanced

3.2.1 Manipulate keys and values

This example show the power of a lazy library and highlight also how to use it in a wrong way.

Unlike regular PHP arrays where there can only be one key of type int or string, a lazy library can have multiple times the same keys and they can be of any type !

```
// This following example is perfectly valid, despite that having array for keys
// in a regular PHP arrays is impossible.
$input = static function () {
    yield ['a'] => 'a';
    yield ['b'] => 'b';
    yield ['c'] => 'c';
};
Collection::fromIterable($input());
```

A lazy collection library can also have multiple times the same key.

Here we are going to make a frequency analysis on the text and see the result. We can see that some data are missing, why ?

```
$string = 'aaaaabbbbcccddddeeeee';

$collection = Collection::fromIterable($string)
    // Run the frequency analysis tool.
    ->frequency()
    // Convert to regular array.
    ->all(); // [5 => 'e', 4 => 'd', 3 => 'c']
```

The reason that the frequency analysis for letters ‘a’ and ‘b’ are missing is because when you call the method ->all(), the collection converts the lazy collection into a regular PHP array, and PHP doesn’t allow having multiple time the same key, so it overrides the previous data and there are missing information in the resulting array.

In order to circumvent this, you can either wrap the final result or normalize it. A better way would be to not convert this into an array and use the lazy collection as an iterator.

Wrapping the result will wrap each result into a PHP array. Normalizing the result will replace keys with a numerical index, but then you might lose some information then.

It’s up to you to decide which one you want to use.

```
$collection = Collection::fromIterable($string)
    // Run the frequency analysis tool.
    ->frequency()
    // Wrap each result into an array.
    ->wrap()
    // Convert to regular array.
    ->all();
/**
 * [
```

(continues on next page)

(continued from previous page)

```
 *    [5 => 'a'],
 *    [4 => 'b'],
 *    [3 => 'c'],
 *    [4 => 'd'],
 *    [5 => 'e'],
 * ]
 */
```

3.2.2 Manipulate strings

```
<?php

declare(strict_types=1);

include __DIR__ . '/../../../vendor/autoload.php';

use loophp\collection\Collection;

$string = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit.
      Quisque feugiat tincidunt sodales.
      Donec ut laoreet lectus, quis mollis nisl.
      Aliquam maximus, orci vel placerat dapibus, libero erat aliquet nibh, nec␣
↪imperdiet felis dui quis est.
      Vestibulum non ante sit amet neque tincidunt porta et sit amet neque.
      In a tempor ipsum. Duis scelerisque libero sit amet enim pretium pulvinar.
      Duis vitae lorem convallis, egestas mauris at, sollicitudin sem.
      Fusce molestie rutrum faucibus.';

// By default will have the same behavior as str_split().
$count = Collection::fromString($string)
    ->explode(' ')
    ->count(); // 107

var_dump($count);

// Or add a separator if needed, same behavior as explode().
$count = Collection::fromString($string, ',')
    ->count(); // 8

var_dump($count);
```

3.2.3 Random number generation

```
<?php

declare(strict_types=1);

include __DIR__ . '/../../../vendor/autoload.php';

use loophp\collection\Collection;

// Generate 300 distinct random numbers between 0 and 1000
```

(continues on next page)

(continued from previous page)

```
$random = static function () {
    return mt_rand() / mt_getrandmax();
};

$random_numbers = Collection::unfold($random)
    ->map(
        static function ($value) {
            return floor($value * 1000) + 1;
        }
    )
    ->distinct()
    ->limit(300)
    ->normalize()
    ->all();

print_r($random_numbers);
```

3.2.4 Approximate the number e

```
<?php

declare(strict_types=1);

include __DIR__ . '/../../../vendor/autoload.php';

use loophp\collection\Collection;

$multiplication = static function (float $value1, float $value2): float {
    return $value1 * $value2;
};

$addition = static function (float $value1, float $value2): float {
    return $value1 + $value2;
};

$fact = static function (int $number) use ($multiplication): float {
    return Collection::range(1, $number + 1)
        ->foldleft(
            $multiplication,
            1
        )
        ->current();
};

$e = static function (int $value) use ($fact): float {
    return $value / $fact($value);
};

$listInt = static function (int $init, callable $succ): Generator {
    yield $init;

    while (true) {
        yield $init = $succ($init);
    }
};
```

(continues on next page)

(continued from previous page)

```
$naturals = $listInt(1, static function (int $n): int {
    return $n + 1;
});

$number_e_approximation = Collection::fromIterable($naturals)
    ->map($e)
    ->until(static function (float $value): bool {
        return 10 ** -12 > $value;
    })
    ->foldleft($addition, 0)
    ->current();

var_dump($number_e_approximation); // 2.718281828459
```

3.2.5 Approximate the number Pi

```
<?php

declare(strict_types=1);

include __DIR__ . '/../../../vendor/autoload.php';

use loophp\collection\Collection;

$monteCarloMethod = static function ($in = 0, $total = 1) {
    $randomNumber1 = mt_rand(0, mt_getrandmax() - 1) / mt_getrandmax();
    $randomNumber2 = mt_rand(0, mt_getrandmax() - 1) / mt_getrandmax();

    if (1 >= (($randomNumber1 ** 2) + ($randomNumber2 ** 2))) {
        ++$in;
    }

    return ['in' => $in, 'total' => ++$total];
};

$pi_approximation = Collection::unfold($monteCarloMethod)
    ->map(
        static function ($value) {
            return 4 * $value['in'] / $value['total'];
        }
    )
    ->window(1)
    ->drop(1)
    ->until(
        static function (array $value): bool {
            return 0.00001 > abs($value[0] - $value[1]);
        }
    )
    ->last();

print_r($pi_approximation->all());
```

3.2.6 Fibonacci sequence

```php
<?php

declare(strict_types=1);

include __DIR__ . '/../../../vendor/autoload.php';

use loophp\collection\Collection;

$fibonacci = static function (int $a = 0, int $b = 1): array {
    return [$b, $b + $a];
};

$c = Collection::unfold($fibonacci)
    ->pluck(0)    // Get the first item of each result.
    ->limit(10);  // Limit the amount of results to 10.

print_r($c->all()); // [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

3.2.7 Gamma function

```php
<?php

declare(strict_types=1);

include __DIR__ . '/../../../vendor/autoload.php';

use loophp\collection\Collection;

$addition = static function (float $value1, float $value2): float {
    return $value1 + $value2;
};

$listInt = static function (int $init, callable $succ): Generator {
    yield $init;

    while (true) {
        yield $init = $succ($init);
    }
};

$ = $listInt(1, static function (int $n): int {
    return $n + 1;
});

$γ = static function (float $n): Closure {
    return static function (int $x) use ($n): float {
        return ($x ** ($n - 1)) * (\M_E ** (-$x));
    };
};

$ϵ = static function (float $value): bool {
    return 10 ** -12 > $value;
};
```

(continues on next page)

(continued from previous page)

```
// Find the factorial of this number. This is not bounded to integers!
// $number = 3; // 2 * 2 => 4
// $number = 6; // 5 * 4 * 3 * 2 => 120
$number = 5.75; // 78.78

$gamma_factorial_approximation = Collection::fromIterable($)
    ->map($γ($number))
    ->until($ϵ)
    ->foldleft($addition, 0)
    ->current();

print_r($gamma_factorial_approximation);
```

3.2.8 Prime numbers

```
<?php

declare(strict_types=1);

include __DIR__ . '/../../../vendor/autoload.php';

use loophp\collection\Collection;

$primesGenerator = static function (Iterator $iterator) use (&$primesGenerator):␣
↪Generator {
    yield $primeNumber = $iterator->current();

    $iterator = new \CallbackFilterIterator(
        $iterator,
        static function (int $a) use ($primeNumber): bool {
            return 0 !== $a % $primeNumber;
        }
    );

    $iterator->next();

    return $iterator->valid() ?
        yield from $primesGenerator($iterator) :
        null;
};

$integerGenerator = static function (int $init, callable $succ) use (&
↪$integerGenerator): Generator {
    yield $init;

    return yield from $integerGenerator($succ($init), $succ);
};

$limit = 1000000;

$primes = $primesGenerator(
    $integerGenerator(
        2,
        static function (int $n): int {
```

(continues on next page)

(continued from previous page)

```
            return $n + 1;
        }
    )
);

// Create a lazy collection of Prime numbers from 2 to infinity.
$lazyPrimeNumbersCollection = Collection::fromIterable(
    $primesGenerator(
        $integerGenerator(
            2,
            static function (int $n): int {
                return $n + 1;
            }
        )
    )
);

// Print out the first 1 million of prime numbers.
foreach ($lazyPrimeNumbersCollection->limit($limit) as $prime) {
    var_dump($prime);
}

// Create a lazy collection of Prime numbers from 2 to infinity.
$lazyPrimeNumbersCollection = Collection::fromIterable(
    $primesGenerator(
        $integerGenerator(
            2,
            static function (int $n): int {
                return $n + 1;
            }
        )
    )
);

// Find out the Twin Prime numbers by filtering out unwanted values.
$lazyTwinPrimeNumbersCollection = Collection::fromIterable(
↪$lazyPrimeNumbersCollection)
    ->zip($lazyPrimeNumbersCollection->tail())
    ->filter(
        static function (array $chunk): bool {
            return 2 === $chunk[1] - $chunk[0];
        }
    );

foreach ($lazyTwinPrimeNumbersCollection->limit($limit) as $prime) {
    var_dump($prime);
}
```

3.2.9 Text analysis

```
<?php

declare(strict_types=1);

include __DIR__ . '/../../../vendor/autoload.php';
```

(continues on next page)

(continued from previous page)

```php
use loophp\collection\Collection;

$collection = Collection::fromString(file_get_contents('http://loripsum.net/api'))
    // Filter out some characters.
    ->filter(
        static function ($item, $key): bool {
            return (bool) preg_match('/^[a-zA-Z]+$/', $item);
        }
    )
    // Lowercase each character.
    ->map(static function (string $letter): string {
        return mb_strtolower($letter);
    })
    // Run the frequency tool.
    ->frequency()
    // Flip keys and values.
    ->flip()
    // Sort values.
    ->sort()
    // Convert to array.
    ->all();

print_r($collection);
```

3.2.10 Random number distribution

```php
<?php

declare(strict_types=1);

include __DIR__ . '/../../../vendor/autoload.php';

use loophp\collection\Collection;
use loophp\collection\Contract\Operation\Sortable;

$min = 0;
$max = 10000;
$groups = 1000;

$randomGenerator = static function () use ($min, $max): int {
    return random_int($min, $max);
};

$distribution = Collection::unfold($randomGenerator)
    ->limit($max)
    ->map(
        static function (int $value) use ($max, $groups): string {
            for ($i = 0; ($max / $groups) > $i; ++$i) {
                if ($i * $groups <= $value && ($i + 1) * $groups >= $value) {
                    return sprintf('%s <= x <= %s', $i * $groups, ($i + 1) * $groups);
                }
            }
        }
    )
```

(continues on next page)

(continued from previous page)

```
    ->groupBy(
        static function ($value, $key) {
            return $value;
        }
    )
    ->map(
        static function (array $value): int {
            return \count($value);
        }
    )
    ->sort(
        Sortable::BY_KEYS,
        static function (string $left, string $right): int {
            [$left_min_limit] = explode(' ', $left);
            [$right_min_limit] = explode(' ', $right);

            return $left_min_limit <=> $right_min_limit;
        }
    );

print_r($distribution->all());

/*
Array
(
    [0 <= x <= 100] => 101086
    [100 <= x <= 200] => 100144
    [200 <= x <= 300] => 99408
    [300 <= x <= 400] => 100079
    [400 <= x <= 500] => 99514
    [500 <= x <= 600] => 100227
    [600 <= x <= 700] => 99983
    [700 <= x <= 800] => 99942
    [800 <= x <= 900] => 99429
    [900 <= x <= 1000] => 100188
)
 */
```

3.2.11 Parse git log

```
<?php

declare(strict_types=1);

include __DIR__ . '/../../../vendor/autoload.php';

use loophp\collection\Collection;
use loophp\collection\Contract\Collection as CollectionInterface;
use loophp\collection\Contract\Operation\Splitable;

$commandStream = static function (string $command): Generator {
    $fh = popen($command, 'r');

    while (false !== $line = fgets($fh)) {
        yield $line;
```

(continues on next page)

(continued from previous page)

```
    }

    fclose($fh);
};

$buildIfThenElseCallbacks = static function (string $lineStart): array {
    return [
        static function ($line) use ($lineStart): bool {
            return \is_string($line) && 0 === mb_strpos($line, $lineStart);
        },
        static function ($line) use ($lineStart): array {
            [, $line] = explode($lineStart, $line);

            return [
                sprintf(
                    '%s:%s',
                    mb_strtolower(str_replace(':', '', $lineStart)),
                    trim($line)
                ),
            ];
        },
    ];
};

$c = Collection::fromIterable($commandStream('git log'))
    ->map(
        static function (string $value): string {
            return trim($value);
        }
    )
    ->compact('', ' ', "\n")
    ->ifThenElse(...$buildIfThenElseCallbacks('commit'))
    ->ifThenElse(...$buildIfThenElseCallbacks('Date:'))
    ->ifThenElse(...$buildIfThenElseCallbacks('Author:'))
    ->ifThenElse(...$buildIfThenElseCallbacks('Merge:'))
    ->ifThenElse(...$buildIfThenElseCallbacks('Signed-off-by:'))
    ->split(
        Splitable::BEFORE,
        static function ($value): bool {
            return \is_array($value) ?
                (1 === preg_match('/^commit:\b[0-9a-f]{5,40}\b/', $value[0])) :
                false;
        }
    )
    ->map(
        static function (array $value): CollectionInterface {
            return Collection::fromIterable($value);
        }
    )
    ->map(
        static function (CollectionInterface $collection): CollectionInterface {
            return $collection
                ->groupBy(
                    static function ($value): ?string {
                        return \is_array($value) ? 'headers' : null;
                    }
                )
```

(continues on next page)

(continued from previous page)

```
                ->groupBy(
                    static function ($value): ?string {
                        return \is_string($value) ? 'log' : null;
                    }
                )
                ->ifThenElse(
                    static function ($value, $key): bool {
                        return 'headers' === $key;
                    },
                    static function ($value, $key): array {
                        return Collection::fromIterable($value)
                            ->unwrap()
                            ->associate(
                                static function ($carry, $key, string $value): string
↪{
                                    [$key, $line] = explode(':', $value, 2);

                                    return $key;
                                },
                                static function ($carry, $key, string $value): string
↪{
                                    [$key, $line] = explode(':', $value, 2);

                                    return trim($line);
                                }
                            )
                            ->all();
                    }
                );
        }
    )
    ->map(
        static function (CollectionInterface $collection): CollectionInterface {
            return $collection
                ->flatten()
                ->groupBy(
                    static function ($value, $key): ?string {
                        if (is_numeric($key)) {
                            return 'log';
                        }

                        return null;
                    }
                );
        }
    )
    ->map(
        static function (CollectionInterface $collection): array {
            return $collection->all();
        }
    )
    ->limit(100);

print_r($c->all());
```

3.2.12 Collatz conjecture

```php
<?php

declare(strict_types=1);

include __DIR__ . '/../../../vendor/autoload.php';

use loophp\collection\Collection;

// The Collatz conjecture (https://en.wikipedia.org/wiki/Collatz_conjecture)
$collatz = static function (int $value): int {
    return 0 === $value % 2 ?
        $value / 2 :
        $value * 3 + 1;
};

$c = Collection::unfold($collatz, 25)
    ->until(
        static function ($number): bool {
            return 1 === $number;
        }
    );

print_r($c->all()); // [25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13,␣
↪40, 20, 10, 5, 16, 8, 4, 2, 1]
```

3.2.13 Read a file

```php
<?php

declare(strict_types=1);

include __DIR__ . '/../../../vendor/autoload.php';

use loophp\collection\Collection;

$c = Collection::fromFile(__FILE__)
    ->lines()
    ->count();

print_r(sprintf('There are %s lines in this file.', $c)); // There are 13 lines in␣
↪this file.
```

CHAPTER 4

API

4.1 Static constructors

4.1.1 empty

Create an empty Collection.

```
$collection = Collection::empty();
```

4.1.2 fromCallable

Create a collection from a callable.

```
$callback = static function () {
    yield 'a';
    yield 'b';
    yield 'c';
};

$collection = Collection::fromCallable($callback);
```

4.1.3 fromFile

Create a collection from a file.

```
Collection::fromFile('http://loripsum.net/api');
```

4.1.4 fromIterable

Create a collection from an iterable.

```
$collection = Collection::fromIterable(['a', 'b', 'c']);
```

4.1.5 fromResource

Create a collection from a resource.

```
$stream = fopen('data://text/plain,' . $string, 'rb');

$collection = Collection::fromResource($stream);
```

4.1.6 fromString

Create a collection from a string.

```
$data = file_get_contents('http://loripsum.net/api');

$collection = Collection::fromString($data);
```

4.1.7 range

Build a collection from a range of values.

Signature: `Collection::range(int $start = 0, $end = INF, $step = 1);`

```
$fibonacci = static function ($a = 0, $b = 1): array {
    return [$b, $a + $b];
};

$even = Collection::range(0, 20, 2); // [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Another example

```
$even = Collection::unfold(static function ($carry) {return $carry + 2;}, -2);
$odd = Collection::unfold(static function ($carry) {return $carry + 2;}, -1);
// Is the same as
$even = Collection::range(0, \INF, 2);
$odd = Collection::range(1, \INF, 2);
```

4.1.8 times

Create a collection by invoking a callback a given amount of times.

If no callback is provided, then it will create a simple list of incremented integers.

Signature: `Collection::times($number = INF, ?callable $callback = null);`

```
$collection = Collection::times(10);
```

4.1.9 unfold

Create a collection by yielding from a callback with a initial value.

> **Warning:** The callback return values are reused as callback arguments at the next callback call.

Signature: `Collection::unfold(callable $callback, ...$parameters);`

```
// A list of Naturals from 1 to Infinity.
Collection::unfold(fn($n) => $n + 1, 1)
    ->normalize();
```

```
$fibonacci = static function ($a = 0, $b = 1): array {
    return [$b, $a + $b];
};

Collection::unfold($fibonacci)
    ->limit(10); // [[0, 1], [1, 1], [1, 2], [2, 3], [3, 5], [5, 8], [8, 13], [13,␣
↪21], [21, 34], [34, 55]]
```

Another example

```
$even = Collection::unfold(static function (int $carry): int {return $carry + 2;}, -
↪2);
$odd = Collection::unfold(static function (int $carry): int {return $carry + 2;}, -1);
// Is the same as
$even = Collection::range(0, \INF, 2);
$odd = Collection::range(1, \INF, 2);
```

4.1.10 with

> **Warning:** Will be deprecated soon. Use `fromCallable`, `fromIterable`, `fromResource`, `fromString` instead.

Create a collection with the provided data.

Signature: `Collection::with($data = [], ...$parameters);`

```
// With an iterable
$collection = Collection::with(['a', 'b']);

// With a string
$collection = Collection::with('string');

$callback = static function () {
    yield 'a';
    yield 'b';
    yield 'c';
};

// With a callback
$collection = Collection::with($callback);
```

(continues on next page)

(continued from previous page)

```
// With a resource/stream
$collection = Collection::with(fopen( __DIR__ . '/vendor/autoload.php', 'r'));
```

4.2 Methods (operations)

Operations always returns a new collection object.

4.2.1 all

Convert the collection into an array.

This is a lossy operation because PHP array keys cannot be duplicated and must either be int or string.

Interface: Allable

Signature: `Collection::all();`

4.2.2 append

Add one or more items to a collection.

Warning: If appended values overwrite existing values, you might find that this operation doesn't work correctly when the collection is converted into an array. It's always better to never convert the collection to an array and use it in a loop. However, if for some reason, you absolutely need to convert it into an array, then use the `Collection::normalize()` operation.

Interface: Appendable

Signature: `Collection::append(...$items);`

```
Collection::fromIterable([1 => '1', 2 => '2', 3 => '3'])
    ->append('4'); // [1 => '1', 2 => '2', 3 => '3', 0 => '4']

Collection::fromIterable(['1', '2', '3'])
    ->append('4')
    ->append('5', '6'); // [0 => 5, 1 => 6, 2 => 3]

Collection::fromIterable(['1', '2', '3'])
    ->append('4')
    ->append('5', '6')
    ->normalize(); // ['1', '2', '3', '4', '5', '6']
```

4.2.3 apply

Execute a callback for each element of the collection without altering the collection item itself.

If the callback does not return *true* then it stops.

Interface: Applyable

Signature: `Collection::apply(...$callbacks);`

```php
$callback = static function ($value, $key): bool
    {
        var_dump('Value is: ' . $value . ', key is: ' . $key);

        return true;
    };

$collection = Collection::fromIterable(['1', '2', '3']);

$collection
    ->apply($callback);
```

4.2.4 associate

Transform keys and values of the collection independently and combine them.

Interface: Associateable

Signature: `Collection::associate(?callable $callbackForKeys = null, ?callable $callbackForValues = null);`

```php
$input = range(1, 10);

Collection::fromIterable($input)
    ->associate(
        static function ($key, $value) {
            return $key * 2;
        },
        static function ($key, $value) {
            return $value * 2;
        }
    );

// [
//   0 => 2,
//   2 => 4,
//   4 => 6,
//   6 => 8,
//   8 => 10,
//   10 => 12,
//   12 => 14,
//   14 => 16,
//   16 => 18,
//   18 => 20,
// ]
```

4.2.5 cache

Useful when using a resource as input and you need to run through the collection multiple times.

Interface: Cacheable

Signature: `Collection::cache(CacheItemPoolInterface $cache = null);`

```php
$fopen = fopen(__DIR__ . '/vendor/autoload.php', 'r');

$collection = Collection::withResource($fopen)
    ->cache();
```

4.2.6 chunk

Chunk a collection of item into chunks of items of a given size.

Interface: Chunkable

Signature: `Collection::chunk(int $size);`

```php
$collection = Collection::fromIterable(range(0, 10));

$collection->chunk(2);
```

4.2.7 collapse

Collapse a collection of items into a simple flat collection.

Interface: Collapseable

Signature: `Collection::collapse();`

```php
$collection = Collection::fromIterable([[1,2], [3, 4]]);

$collection->collapse();
```

4.2.8 column

Return the values from a single column in the input iterables.

Interface: Columnable

Signature: `Collection::column($index);`

```php
$records = [
    [
        'id' => 2135,
        'first_name' => 'John',
        'last_name' => 'Doe',
    ],
    [
        'id' => 3245,
        'first_name' => 'Sally',
        'last_name' => 'Smith',
    ],
    [
        'id' => 5342,
        'first_name' => 'Jane',
        'last_name' => 'Jones',
    ],
    [
```

(continues on next page)

(continued from previous page)

```
        'id' => 5623,
        'first_name' => 'Peter',
        'last_name' => 'Doe',
    ],
];

$result = Collection::fromIterable($records)
    ->column('first_name');
```

4.2.9 combinate

Get all the combinations of a given length of a collection of items.

Interface: Combinateable

Signature: `Collection::combinate(?int $length);`

```
$collection = Collection::fromIterable(['a', 'b', 'c', 'd'])
    ->combinate(3);
```

4.2.10 combine

Combine a collection of items with some other keys.

Interface: Combineable

Signature: `Collection::combine(...$keys);`

```
$collection = Collection::fromIterable(['a', 'b', 'c', 'd'])
    ->combine('w', 'x', 'y', 'z')
```

4.2.11 compact

Remove given values from the collection, if no values are provided, it removes only the null value.

Interface: Compactable

Signature: `Collection::compact(...$values);`

```
$collection = Collection::fromIterable(['a', 1 => 'b', null, false, 0, 'c'];)
    ->compact(); // ['a', 1 => 'b', 3 => false, 4 => 0, 5 => 'c']

$collection = Collection::fromIterable(['a', 1 => 'b', null, false, 0, 'c'];)
    ->compact(null, 0); // ['a', 1 => 'b', 3 => false, 5 => 'c']
```

4.2.12 contains

Check if the collection contains one or more value.

Interface: Containsable

Signature: `Collection::contains(...$value);`

4.2.13 current

Get the value of an item in the collection given a numeric index, default index is 0.

Interface: Currentable

Signature: `Collection::current(int $index = 0);`

```
Collection::fromIterable(['a', 'b', 'c', 'd'])->current(); // Return 'a'
Collection::fromIterable(['a', 'b', 'c', 'd'])->current(0); // Return 'a'
Collection::fromIterable(['a', 'b', 'c', 'd'])->current(1); // Return 'b'
Collection::fromIterable(['a', 'b', 'c', 'd'])->current(10); // Return null
```

4.2.14 cycle

Cycle around a collection of items.

Interface: Cycleable

Signature: `Collection::cycle(int $length = 0);`

```
$collection = Collection::fromIterable(['a', 'b', 'c', 'd'])
    ->cycle(10)
```

4.2.15 diff

It compares the collection against another collection or a plain array based on its values. This method will return the values in the original collection that are not present in the given collection.

Interface: Diffable

Signature: `Collection::diff(...$values);`

```
$collection = Collection::fromIterable(['a', 'b', 'c', 'd', 'e'])
    ->diff('a', 'b', 'c', 'x'); // [3 => 'd', 4 => 'e']
```

4.2.16 diffKeys

It compares the collection against another collection or a plain object based on its keys. This method will return the key / value pairs in the original collection that are not present in the given collection.

Interface: Diffkeysable

Signature: `Collection::diffKeys(...$values);`

```
$collection = Collection::fromIterable(['a', 'b', 'c', 'd', 'e'])
    ->diffKeys(1, 2); // [0 => 'a', 3 => 'd', 4 => 'e']
```

4.2.17 distinct

Remove duplicated values from a collection.

Interface: Distinctable

Signature: `Collection::distinct();`

```
$collection = Collection::fromIterable(['a', 'b', 'c', 'd', 'a'])
    ->distinct()
```

4.2.18 drop

Drop the n first items of the collection.

Interface: Dropable

Signature: `Collection::drop(int ...$counts);`

```
Collection::fromIterable(range(10, 20))
    ->drop(2); // [12,13,14,15,16,17,18,19,20]
```

4.2.19 dropWhile

It inspects the original collection and takes from it its elements from the moment when the condition fails for the first time till the end of the list.

Interface: DropWhileable

Signature: `Collection::dropWhile(callable ...$callbacks);`

```
$isSmallerThanThree = static function (int $value): bool {
    return 3 > $value;
};

Collection::fromIterable([1,2,3,4,5,6,7,8,9,1,2,3])
    ->dropWhile($isSmallerThanThree); // [3,4,5,6,7,8,9,1,2,3]
```

4.2.20 duplicate

Find duplicated values from the collection.

Interface: Duplicateable

Signature: `Collection::duplicate();`

```
// It might returns duplicated values !
Collection::fromIterable(['a', 'b', 'c', 'a', 'c', 'a'])
        ->duplicate(); // [3 => 'a', 4 => 'c', 5 => 'a']

// Use ::distinct() and ::normalize() to get what you want.
Collection::fromIterable(['a', 'b', 'c', 'a', 'c', 'a'])
        ->duplicate()
        ->distinct()
        ->normalize() // [0 => 'a', 1 => 'c']
```

4.2.21 every

This operation tests whether all elements in the collection pass the test implemented by the provided callback.

Interface: Everyable

Signature: `Collection::every(callable $callback);`

```php
$callback = static function ($value): bool {
    return $value < 20;
};

Collection::fromIterable(range(0, 10))
    ->every($callback)
    ->current(); // true
```

4.2.22 explode

Explode a collection into subsets based on a given value.

This operation use the Split operation with the flag `Splitable::REMOVE` and thus, values used to explode the collection are removed from the chunks.

Interface: Explodeable

Signature: `Collection::explode(...$items);`

```php
$string = 'I am just a random piece of text.';

$collection = Collection::fromIterable($string)
    ->explode('o');
```

4.2.23 falsy

Check if the collection contains falsy values.

Interface: Falsyable

Signature: `Collection::falsy();`

4.2.24 filter

Filter collection items based on one or more callbacks.

Interface: Filterable

Signature: `Collection::filter(callable ...$callbacks);`

```php
$callback = static function($value): bool {
    return 0 === $value % 3;
};

$collection = Collection::fromIterable(range(1, 100))
    ->filter($callback);
```

4.2.25 first

Get the first items from the collection.

Interface: Firstable

Signature: `Collection::first();`

```php
$generator = static function (): Generator {
    yield 'a' => 'a';
    yield 'b' => 'b';
    yield 'c' => 'c';
    yield 'a' => 'd';
    yield 'b' => 'e';
    yield 'c' => 'f';
};

Collection::fromIterable($generator())
    ->first(); // ['a' => 'a']
```

4.2.26 flatten

Flatten a collection of items into a simple flat collection.

Interface: Flattenable

Signature: `Collection::flatten(int $depth = PHP_INT_MAX);`

```php
$collection = Collection::fromIterable([0, [1, 2], [3, [4, [5, 6]]]])
    ->flatten();
```

4.2.27 flip

Flip keys and items in a collection.

Interface: Flipable

Signature: `Collection::flip(int $depth = PHP_INT_MAX);`

```php
$collection = Collection::fromIterable(['a', 'b', 'c', 'a'])
    ->flip();
```

Tip: array_flip() and Collection::flip() can behave different, check the following examples.

When using regular arrays, array_flip() can be used to remove duplicates (dedup-licate an array).

```php
$dedupArray = array_flip(array_flip(['a', 'b', 'c', 'd', 'a']));
```

This example will return `['a', 'b', 'c', 'd']`.

However, when using a collection:

```php
$dedupCollection = Collection::fromIterable(['a', 'b', 'c', 'd', 'a'])
    ->flip()
    ->flip()
    ->all();
```

This example will return `['a', 'b', 'c', 'd', 'a']`.

4.2.28 foldLeft

Takes the initial value and the first item of the list and applies the function to them, then feeds the function with this result and the second argument and so on. See *scanLeft* for intermediate results.

Interface: FoldLeftable

Signature: `Collection::foldLeft(callable $callback, $initial = null);`

4.2.29 foldLeft1

Takes the first 2 items of the list and applies the function to them, then feeds the function with this result and the third argument and so on. See *scanLeft1* for intermediate results.

Interface: FoldLeft1able

Signature: `Collection::foldLeft1(callable $callback);`

4.2.30 foldRight

Takes the initial value and the last item of the list and applies the function, then it takes the penultimate item from the end and the result, and so on. See *scanRight* for intermediate results.

Interface: FoldRightable

Signature: `Collection::foldRight(callable $callback, $initial = null);`

4.2.31 foldRight1

Takes the last two items of the list and applies the function, then it takes the third item from the end and the result, and so on. See *scanRight1* for intermediate results.

Interface: FoldRight1able

Signature: `Collection::foldRight1(callable $callback);`

4.2.32 forget

Remove items having specific keys.

Interface: Forgetable

Signature: `Collection::forget(...$keys);`

```
$collection = Collection::fromIterable(range('a', 'z'))
    ->forget(5, 6, 10, 15);
```

4.2.33 frequency

Calculate the frequency of the values, frequencies are stored in keys.

Values can be anything (object, scalar, ...).

Interface: Frequencyable

Signature: `Collection::frequency();`

```php
$collection = Collection::fromIterable(['a', 'b', 'c', 'b', 'c', 'c'])
    ->frequency()
    ->all(); // [1 => 'a', 2 => 'b', 3 => 'c'];
```

4.2.34 get

Get a specific element of the collection from a key, if the key doesn't exists, returns the default value.

Interface: Getable

Signature: `Collection::get($key, $default = null);`

4.2.35 group

Takes a list and returns a list of lists such that the concatenation of the result is equal to the argument. Moreover, each sublist in the result contains only equal elements.

Interface: Groupable

Signature: `Collection::group();`

```php
Collection::fromString('Mississippi')
    ->group(); // [ [0 => 'M'], [1 => 'i'], [2 => 's', 3 => 's'], [4 => 'i'], [5 => 's
↪', 6 => 's'], [7 => 'i'], [8 => 'p', 9 => 'p'], [10 => 'i'] ]
```

4.2.36 groupBy

Group items, the key used to group items can be customized in a callback. By default it's the key is the item's key.

Interface: GroupByable

Signature: `Collection::groupBy(?callable $callback = null);`

```php
$callback = static function () {
        yield 1 => 'a';

        yield 1 => 'b';

        yield 1 => 'c';

        yield 2 => 'd';

        yield 2 => 'e';

        yield 3 => 'f';
};

$collection = Collection::fromIterable($callback)
    ->groupBy();
```

4.2.37 has

Check if the collection has a value. The value must be provided as a callback.

Interface: Hasable

Signature: `Collection::has(callable $callback);`

4.2.38 head

Interface: Headable

Signature: `Collection::head();`

```php
$generator = static function (): \Generator {
        yield 1 => 'a';
        yield 1 => 'b';
        yield 1 => 'c';
        yield 2 => 'd';
        yield 2 => 'e';
        yield 3 => 'f';
};

Collection::fromIterable($generator())
    ->head(); // [1 => 'a']
```

4.2.39 ifThenElse

Execute a callback when a condition is met.

Interface: IfThenElseable

Signature: `Collection::ifThenElse(callable $condition, callable $then, ?callable $else = null);`

```php
$input = range(1, 5);

$condition = static function (int $value): bool {
    return 0 === $value % 2;
};

$then = static function (int $value): int {
    return $value * $value;
};

$else = static function (int $value): int {
    return $value + 2;
};

Collection::fromIterable($input)
    ->ifThenElse($condition, $then); // [1, 4, 3, 16, 5]

Collection::fromIterable($input)
    ->ifThenElse($condition, $then, $else) // [3, 4, 5, 16, 7]
```

4.2.40 implode

Convert all the elements of the collection to a single string.

The glue character can be provided, default is the empty character.

Interface: Implodeable

Signature: `Collection::implode(string $glue = '');`

4.2.41 init

Returns the collection without its last item.

Interface: Initable

Signature: `Collection::init();`

```
Collection::fromIterable(range('a', 'e'))
    ->init(); // ['a', 'b', 'c', 'd']
```

4.2.42 inits

Returns all initial segments of the collection, shortest first.

Interface: Initsable

Signature: `Collection::inits();`

```
Collection::fromIterable(range('a', 'c'))
    ->inits(); // [[], ['a'], ['a', 'b'], ['a', 'b', 'c']]
```

4.2.43 intersect

Removes any values from the original collection that are not present in the given collection.

Interface: Intersectable

Signature: `Collection::intersect(...$values);`

```
$collection = Collection::fromIterable(range('a', 'e'))
    ->intersect('a', 'b', 'c'); // ['a', 'b', 'c']
```

4.2.44 intersectKeys

Removes any keys from the original collection that are not present in the given collection.

Interface: Intersectkeysable

Signature: `Collection::intersectKeys(...$values);`

```
$collection = Collection::fromIterable(range('a', 'e'))
    ->intersectKeys(0, 2, 4); // ['a', 'c', 'e']
```

4.2.45 intersperse

Insert a given value at every n element of a collection and indices are not preserved.

Interface: Intersperseable

Signature: `Collection::intersperse($element, int $every = 1, int $startAt = 0);`

```
$collection = Collection::fromIterable(range('a', 'z'))
    ->intersperse('foo', 3);
```

4.2.46 key

Get the key of an item in the collection given a numeric index, default index is 0.

Interface: Keyable

Signature: `Collection::key(int $index = 0);`

```
Collection::fromIterable(['a', 'b', 'c', 'd'])->key(); // Return 0
Collection::fromIterable(['a', 'b', 'c', 'd'])->key(0); // Return 0
Collection::fromIterable(['a', 'b', 'c', 'd'])->key(1); // Return 1
Collection::fromIterable(['a', 'b', 'c', 'd'])->key(10); // Return null
```

4.2.47 keys

Get the keys of the items.

Interface: Keysable

Signature: `Collection::keys();`

```
$collection = Collection::fromIterable(range('a', 'z'))
    ->keys();
```

4.2.48 last

Extract the last element of a collection, which must be finite and non-empty.

Interface: Lastable

Signature: `Collection::last();`

```
$generator = static function (): Generator {
    yield 'a' => 'a';
    yield 'b' => 'b';
    yield 'c' => 'c';
    yield 'a' => 'd';
    yield 'b' => 'e';
    yield 'c' => 'f';
};

Collection::fromIterable($generator())
    ->last(); // ['c' => 'f']
```

4.2.49 limit

Limit the amount of values in the collection.

Interface: Limitable

Signature: `Collection::limit(int $limit);`

```
$fibonacci = static function ($a = 0, $b = 1): array {
    return [$b, $a + $b];
};

$collection = Collection::unfold($fibonacci)
    ->limit(10);
```

4.2.50 lines

Split a string into lines.

Interface: Linesable

Signature: `Collection::lines();`

```
$string = <<<'EOF'
The quick brow fox jumps over the lazy dog.

This is another sentence.
EOF;

Collection::fromString($string)
    ->lines();
```

4.2.51 map

Apply one or more supplied callbacks to every item of a collection and use the return value.

Warning: Keys are preserved, use the "normalize" operation if you want to re-index the keys.

Interface: Mapable

Signature: `Collection::map(callable ...$callbacks);`

```
$mapper = static function($value, $key) {
    return $value * 2;
};

$collection = Collection::fromIterable(range(1, 100))
    ->map($mapper);
```

4.2.52 merge

Merge one or more collection of items onto a collection.

Interface: Mergeable

Signature: `Collection::merge(...$sources);`

```
$collection = Collection::fromIterable(range(1, 10))
    ->merge(['a', 'b', 'c'])
```

4.2.53 normalize

Replace, reorder and use numeric keys on a collection.

Interface: Normalizeable

Signature: `Collection::normalize();`

```
$collection = Collection::fromIterable(['a' => 'a', 'b' => 'b', 'c' => 'c'])
    ->normalize();
```

4.2.54 nth

Get every n-th element of a collection.

Interface: Nthable

Signature: `Collection::nth(int $step, int $offset = 0);`

```
$collection = Collection::fromIterable(range(10, 100))
    ->nth(3);
```

4.2.55 nullsy

Check if the collection contains nullsy values.

Interface: Nullsyable

Signature: `Collection::nullsy();`

4.2.56 only

Get items having corresponding given keys.

Interface: Onlyable

Signature: `Collection::only(...$keys);`

```
$collection = Collection::fromIterable(range(10, 100))
    ->only(3, 10, 'a', 9);
```

4.2.57 pack

Wrap each items into an array containing 2 items: the key and the value.

Interface: Packable

Signature: `Collection::pack();`

```
$input = ['a' => 'b', 'c' => 'd', 'e' => 'f'];

$c = Collection::fromIterable($input)
    ->pack();

 // [
```

(continues on next page)

(continued from previous page)

```
//   ['a', 'b'],
//   ['c', 'd'],
//   ['e', 'f'],
// ]
```

4.2.58 pad

Pad a collection to the given length with a given value.

Interface: Padable

Signature: `Collection::pad(int $size, $value);`

```
$collection = Collection::fromIterable(range(1, 5))
    ->pad(10, 'foo');
```

4.2.59 pair

Make an associative collection from pairs of values.

Interface: Pairable

Signature: `Collection::pair();`

```
$input = [
    [
        'key' => 'k1',
        'value' => 'v1',
    ],
    [
        'key' => 'k2',
        'value' => 'v2',
    ],
    [
        'key' => 'k3',
        'value' => 'v3',
    ],
    [
        'key' => 'k4',
        'value' => 'v4',
    ],
    [
        'key' => 'k4',
        'value' => 'v5',
    ],
];

$c = Collection::fromIterable($input)
    ->unwrap()
    ->pair()
    ->group()
    ->all();

// [
//     [k1] => v1
```

(continues on next page)

(continued from previous page)

```
//     [k2] => v2
//     [k3] => v3
//     [k4] => [
//         [0] => v4
//         [1] => v5
//     ]
// ]
```

4.2.60 permutate

Find all the permutations of a collection.

Interface: Permutateable

Signature: `Collection::permutate(int $size, $value);`

```
$collection = Collection::fromIterable(['hello', 'how', 'are', 'you'])
    ->permutate();
```

4.2.61 pluck

Retrieves all of the values of a collection for a given key.

Interface: Pluckable

Signature: `Collection::pluck($pluck, $default = null);`

```
$fibonacci = static function ($a = 0, $b = 1): array {
    return [$b, $a + $b];
};

$collection = Collection::unfold($fibonacci)
    ->limit(10)
    ->pluck(0);
```

4.2.62 prepend

Push an item onto the beginning of the collection.

Warning: If prepended values overwrite existing values, you might find that this operation doesn't work correctly when the collection is converted into an array. It's always better to never convert the collection to an array and use it in a loop. However, if for some reason, you absolutely need to convert it into an array, then use the `Collection::normalize()` operation.

Interface: Prependable

Signature: `Collection::prepend(...$items);`

```
Collection::fromIterable([1 => '1', 2 => '2', 3 => '3'])
    ->prepend('4'); // [0 => 4, 1 => '1', 2 => '2', 3 => '3']
```

(continues on next page)

(continued from previous page)

```
Collection::fromIterable(['1', '2', '3'])
    ->prepend('4')
    ->prepend('5', '6'); // [0 => 1, 1 => 2, 2 => 3]

Collection::fromIterable(['1', '2', '3'])
    ->prepend('4')
    ->prepend('5', '6')
    ->normalize(); // ['5', '6', '4', '1', '2', '3']
```

4.2.63 product

Get the the cartesian product of items of a collection.

Interface: Productable

Signature: `Collection::product(iterable ...$iterables);`

```
$collection = Collection::fromIterable(['4', '5', '6'])
    ->product(['1', '2', '3'], ['a', 'b'], ['foo', 'bar']);
```

4.2.64 random

It returns a random item from the collection. An optional integer can be passed to random to specify how many items you would like to randomly retrieve.

Interface: Randomable

Signature: `Collection::random(int $size = 1);`

```
$collection = Collection::fromIterable(['4', '5', '6'])
    ->random(); // ['6']
```

4.2.65 reduction

Reduce a collection of items through a given callback.

Interface: Reductionable

Signature: `Collection::reduction(callable $callback, $initial = null);`

```
$multiplication = static function ($value1, $value2) {
    return $value1 * $value2;
};

$addition = static function ($value1, $value2) {
    return $value1 + $value2;
};

$fact = static function (int $number) use ($multiplication) {
    return Collection::range(1, $number + 1)
        ->reduce(
            $multiplication,
            1
```

(continues on next page)

(continued from previous page)

```
        );
};

$e = static function (int $value) use ($fact): float {
    return $value / $fact($value);
};

$number_e_approximation = Collection::times()
    ->map($e)
    ->limit(10)
    ->reduction($addition);
```

4.2.66 reverse

Reverse order items of a collection.

Interface: Reverseable

Signature: `Collection::reverse();`

```
$collection = Collection::fromIterable(['a', 'b', 'c'])
    ->reverse();
```

4.2.67 rsample

Work in progress... sorry.

4.2.68 scale

Scale/normalize values.

Interface: Scaleable

Signature: `Collection::scale(float $lowerBound, float $upperBound, ?float $wantedLowerBound = null, ?float $wantedUpperBound = null, ?float $base = null);`

```
$collection = Collection::range(0, 10, 2)
    ->scale(0, 10);

$collection = Collection::range(0, 10, 2)
    ->scale(0, 10, 5, 15, 3);
```

4.2.69 scanLeft

Takes the initial value and the first item of the list and applies the function to them, then feeds the function with this result and the second argument and so on. It returns the list of intermediate and final results.

Interface: ScanLeftable

Signature: `Collection::scanLeft(callable $callback, $initial = null);`

```php
$callback = static function ($carry, $value) {
    return $carry / $value;
};

Collection::fromIterable([4, 2, 4])
    ->scanLeft($callback, 64)
    ->normalize(); // [64 ,16 ,8 ,2]

Collection::empty()
    ->scanLeft($callback, 3)
    ->normalize(); // [3]
```

4.2.70 scanLeft1

Takes the first 2 items of the list and applies the function to them, then feeds the function with this result and the third argument and so on. It returns the list of intermediate and final results.

> **Warning:** You might need to use the `normalize` operation after this.

Interface: ScanLeft1able

Signature: `Collection::scanLeft1(callable $callback);`

```php
$callback = static function ($carry, $value) {
    return $carry / $value;
};

Collection::fromIterable([64, 4, 2, 8])
    ->scanLeft1($callback); // [64 ,16 ,8 ,1]

Collection::fromIterable([12])
    ->scanLeft1($callback); // [12]
```

4.2.71 scanRight

Takes the initial value and the last item of the list and applies the function, then it takes the penultimate item from the end and the result, and so on. It returns the list of intermediate and final results.

Interface: ScanRightable

Signature: `Collection::scanRight(callable $callback, $initial = null);`

```php
$callback = static function ($carry, $value) {
    return $value / $carry;
};

Collection::fromIterable([8, 12, 24, 4])
    ->scanRight($callback, 2); // [8, 1, 12, 2, 2]

Collection::empty()
    ->scanRight($callback, 3); // [3]
```

4.2.72 scanRight1

Takes the last two items of the list and applies the function, then it takes the third item from the end and the result, and so on. It returns the list of intermediate and final results.

> **Warning:** You might need to use the `normalize` operation after this.

Interface: ScanRight1able

Signature: `Collection::scanRight1(callable $callback);`

```php
$callback = static function ($carry, $value) {
    return $value / $carry;
};

Collection::fromIterable([8, 12, 24, 2])
    ->scanRight1($callback); // [8, 1, 12, 2]

Collection::fromIterable([12])
    ->scanRight1($callback); // [12]
```

4.2.73 since

Skip items until callback is met.

Interface: Sinceable

Signature: `Collection::since(callable ...$callbacks);`

```php
// Parse the composer.json of a package and get the require-dev dependencies.
$collection = Collection::withResource(fopen(__DIR__ . '/composer.json', 'rb'))
    // Group items when EOL character is found.
    ->split(
        static function (string $character): bool {
            return "\n" === $character;
        }
    )
    // Implode characters to create a line string
    ->map(
        static function (array $characters): string {
            return implode('', $characters);
        }
    )
    // Skip items until the string "require-dev" is found.
    ->since(
        static function ($line) {
            return false !== strpos($line, 'require-dev');
        }
    )
    // Skip items after the string "}" is found.
    ->until(
        static function ($line) {
            return false !== strpos($line, '}');
        }
    )
```

(continues on next page)

(continued from previous page)

```
    // Re-index the keys
    ->normalize()
    // Filter out the first line and the last line.
    ->filter(
        static function ($line, $index) {
            return 0 !== $index;
        },
        static function ($line) {
            return false === strpos($line, '}');
        }
    )
    // Trim remaining results and explode the string on ':'.
    ->map(
        static function ($line) {
            return trim($line);
        },
        static function ($line) {
            return explode(':', $line);
        }
    )
    // Take the first item.
    ->pluck(0)
    // Convert to array.
    ->all();

    print_r($collection);
```

4.2.74 slice

Get a slice of a collection.

Interface: Sliceable

Signature: `Collection::slice(int $offset, ?int $length = null);`

```
$collection = Collection::fromIterable(range('a', 'z'))
    ->slice(5, 5);
```

4.2.75 sort

Sort a collection using a callback. If no callback is provided, it will sort using natural order.

By default, it will sort by values and using a callback. If you want to sort by keys, you can pass a parameter to change the behavior or use twice the flip operation. See the example below.

Interface: Sortable

Signature: `Collection::sort(?callable $callback = null);`

```
// Regular values sorting
$collection = Collection::fromIterable(['z', 'y', 'x'])
    ->sort();

// Regular values sorting
$collection = Collection::fromIterable(['z', 'y', 'x'])
```

(continues on next page)

(continued from previous page)

```
    ->sort(Operation\Sortable::BY_VALUES);

// Regular values sorting with a custom callback
$collection = Collection::fromIterable(['z', 'y', 'x'])
    ->sort(
            Operation\Sortable::BY_VALUES,
            static function ($left, $right): int {
                // Do the comparison here.
                return $left <=> $right;
            }
    );

// Regular keys sorting (no callback is needed here)
$collection = Collection::fromIterable(['z', 'y', 'x'])
    ->sort(
            Operation\Sortable::BY_KEYS
    );

// Regular keys sorting using flip() operations.
$collection = Collection::fromIterable(['z', 'y', 'x'])
    ->flip() // Exchange values and keys
    ->sort() // Sort the values (which are now the keys)
    ->flip(); // Flip again to put back the keys and values, sorted by keys.
```

4.2.76 span

Returns a tuple where first element is longest prefix (possibly empty) of elements that satisfy the callback and second element is the remainder.

Interface: Spanable

Signature: `Collection::span(callable $callback);`

```
$input = range(1, 10);

Collection::fromIterable($input)
    ->span(fn ($x) => $x < 4); // [ [1, 2, 3], [4, 5, 6, 7, 8, 9, 10] ]
```

4.2.77 split

Split a collection using one or more callbacks.

A flag must be provided in order to specify whether the value used to split the collection should be added at the end of a chunk, at the beginning of a chunk, or completely removed.

Interface: Splitable

Signature: `Collection::split(int $type = Splitable::BEFORE, callable ... $callbacks);`

```
$splitter = static function ($value): bool {
    return 0 === $value % 3;
};

$collection = Collection::fromIterable(range(0, 10))
```

(continues on next page)

(continued from previous page)

```
    ->split(Splitable::BEFORE, $splitter); [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10]]

$collection = Collection::fromIterable(range(0, 10))
    ->split(Splitable::AFTER, $splitter); [[0], [1, 2, 3], [4, 5, 6], [7, 8, 9], [10]]

$collection = Collection::fromIterable(range(0, 10))
    ->split(Splitable::REMOVE, $splitter); [[1, 2], [4, 5], [7, 8], [10]]
```

4.2.78 tail

Get the collection items except the first.

Interface: Tailable

Signature: `Collection::tail();`

```
Collection::fromIterable(['a', 'b', 'c'])
    ->tail(); // [1 => 'b', 2 => 'c']
```

4.2.79 tails

Returns the list of initial segments of its argument list, shortest last.

Interface: Tailsable

Signature: `Collection::tails();`

```
Collection::fromIterable(['a', 'b', 'c'])
    ->tails(); // [['a', 'b', 'c'], ['b', 'c'], ['c'], []]
```

4.2.80 takeWhile

It inspects the original collection and takes from it its elements to the moment when the condition fails, then it stops processing.

Interface: TakeWhileable

Signature: `Collection::takeWhile(callable $callback);`

```
$isSmallerThanThree = static function (int $value): bool {
    return 3 > $value;
};

Collection::fromIterable([1,2,3,4,5,6,7,8,9,1,2,3])
    ->takeWhile($isSmallerThanThree); // [1,2]
```

4.2.81 transpose

Matrix transposition.

Interface: Transposeable

Signature: `Collection::transpose();`

```php
$records = [
    [
        'id' => 2135,
        'first_name' => 'John',
        'last_name' => 'Doe',
    ],
    [
        'id' => 3245,
        'first_name' => 'Sally',
        'last_name' => 'Smith',
    ],
    [
        'id' => 5342,
        'first_name' => 'Jane',
        'last_name' => 'Jones',
    ],
    [
        'id' => 5623,
        'first_name' => 'Peter',
        'last_name' => 'Doe',
    ],
];

$result = Collection::fromIterable($records)
    ->transpose();
```

4.2.82 truthy

Check if the collection contains truthy values.

Interface: Truthyable

Signature: `Collection::truthy();`

4.2.83 unlines

Create a string from lines.

Interface: Unlinesable

Signature: `Collection::unlines();`

```php
$lines = [
    'The quick brow fox jumps over the lazy dog.',
    '',
    'This is another sentence.',
];

Collection::fromIterable($lines)
    ->unlines()
    ->current();
```

4.2.84 unpack

Unpack items.

Interface: Unpackable

Signature: `Collection::unpack();`

```
$input = [['a', 'b'], ['c', 'd'], ['e', 'f']];

$c = Collection::fromIterable($input)
    ->unpack();

// [
//     ['a' => 'b'],
//     ['c' => 'd'],
//     ['e' => 'f'],
// ];
```

4.2.85 unpair

Unpair a collection of pairs.

Interface: Unpairable

Signature: `Collection::unpair();`

```
$input = [
    'k1' => 'v1',
    'k2' => 'v2',
    'k3' => 'v3',
    'k4' => 'v4',
];

$c = Collection::fromIterable($input)
    ->unpair();

// [
//     ['k1', 'v1'],
//     ['k2', 'v2'],
//     ['k3', 'v3'],
//     ['k4', 'v4'],
// ];
```

4.2.86 until

Limit a collection using a callback.

Interface: Untilable

Signature: `Collection::until(callable ...$callbacks);`

```
// The Collatz conjecture (https://en.wikipedia.org/wiki/Collatz_conjecture)
$collatz = static function (int $value): int
{
    return 0 === $value % 2 ?
        $value / 2:
        $value * 3 + 1;
};
```

(continues on next page)

(continued from previous page)

```
$collection = Collection::unfold($collatz, 10)
    ->until(static function ($number): bool {
        return 1 === $number;
    });
```

4.2.87 unwindow

Contrary of `Collection::window()`, usually needed after a call to that operation.

Interface: Unwindowable

Signature: `Collection::unwindow();`

```
// Drop all the items before finding five 9 in a row.
$input = [1, 2, 3, 4, 5, 6, 7, 8, 9, 9, 9, 9, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18];

Collection::fromIterable($input)
    ->window(4)
    ->dropWhile(
        static function (array $value): bool {
            return $value !== [9, 9, 9, 9];
        }
    )
    ->unwindow()
    ->drop(1)
    ->normalize(); // [10, 11, 12, 13, 14, 15, 16, 17, 18]
```

4.2.88 unwords

Create a string from words.

Interface: Unwordsable

Signature: `Collection::unwords();`

```
$words = [
    'The',
    'quick',
    'brow',
    'fox',
    'jumps',
    'over',
    'the',
    'lazy',
    "dog.\n\nThis",
    'is',
    'another',
    'sentence.',
];

Collection::fromIterable($words)
    ->unwords();
```

4.2.89 unwrap

Unwrap every collection element.

Interface: Unwrapable

Signature: `Collection::unwrap();`

```
$data = [['a' => 'A'], ['b' => 'B'], ['c' => 'C']];

$collection = Collection::fromIterable($data)
   ->unwrap();
```

4.2.90 unzip

Unzip a collection.

Interface: Unzipable

Signature: `Collection::unzip();`

```
$a = Collection::fromIterable(['a' => 'a', 'b' => 'b', 'c' => 'c'])
    ->zip(['d', 'e', 'f', 'g'], [1, 2, 3, 4, 5]);

$b = Collection::fromIterable($a)
    ->unzip(); // [ ['a','b','c',null,null], ['d','e','f','g',null], [1,2,3,4,5] ]
```

4.2.91 window

Loop the collection by yielding a specific window of data of a given length.

Interface: Windowable

Signature: `Collection::window(int $size);`

```
$data = range('a', 'z');

Collection::fromIterable($data)
   ->window(2)
   ->all(); // [ ['a'], ['a', 'b'], ['b', 'c'], ['c', 'd'], ... ]
```

4.2.92 words

Get words from a string.

Interface: Wordsable

Signature: `Collection::words();`

```
$string = <<<'EOF'
The quick brow fox jumps over the lazy dog.

This is another sentence.
EOF;
```

(continues on next page)

(continued from previous page)

```
Collection::fromString($string)
    ->words()
```

4.2.93 wrap

Wrap every element into an array.

Interface: Wrapable

Signature: `Collection::wrap();`

```
$data = ['a' => 'A', 'b' => 'B', 'c' => 'C'];

$collection = Collection::fromIterable($data)
   ->wrap();
```

4.2.94 zip

Zip a collection together with one or more iterables.

Interface: Zipable

Signature: `Collection::zip(iterable ...$iterables);`

```
$even = Collection::range(0, INF, 2);
$odd = Collection::range(1, INF, 2);

$positiveIntegers = Collection::fromIterable($even)
    ->zip($odd)
    ->limit(100)
    ->flatten();
```

CHAPTER 5

Tests, code quality and code style

Every time changes are introduced into the library, Github Actions run the tests.

Tests are written with PHPSpec and you can find the coverage percentage on a badge on the README file.

PHPInfection is also triggered used to ensure that your code is properly tested.

The code style is based on PSR-12 plus a set of custom rules. Find more about the code style in use in the package drupol/php-conventions.

A PHP quality tool, Grumphp, is used to orchestrate all these tasks at each commit on the local machine, but also on the continuous integration tools.

To run the whole tests tasks locally, do

```
composer grumphp
```

or

```
./vendor/bin/grumphp run
```

Here's an example of output that shows all the tasks that are setup in Grumphp and that will check your code

```
$ ./vendor/bin/grumphp run
GrumPHP is sniffing your code!
Running task  1/14: SecurityChecker... ✓
Running task  2/14: Composer... ✓
Running task  3/14: ComposerNormalize... ✓
Running task  4/14: YamlLint... ✓
Running task  5/14: JsonLint... ✓
Running task  6/14: PhpLint... ✓
Running task  7/14: TwigCs... ✓
Running task  8/14: PhpCsAutoFixerV2... ✓
Running task  9/14: PhpCsFixerV2... ✓
Running task 10/14: Phpcs... ✓
Running task 11/14: Psalm... ✓
Running task 12/14: PhpStan... ✓
```

(continues on next page)

(continued from previous page)

```
Running task 13/14: Phpspec... ✓
Running task 14/14: Infection... ✓
$
```

CHAPTER 6

Contributing

See the file CONTRIBUTING.md but feel free to contribute to this library by sending Github pull requests.