Iterators & Generators

Looping with ease in modern PHP

THRIVE MARKET

Eli White

php[tek] 2026 · eliw.com

Why me?

- 30+ years in the industry
- 25+ years exclusively* working in PHP
- Worked in numerous industries

What we'll cover

- The Iterator interface: Looping over your objects
- A quick tour of the SPL Iterator library
- Generators: a lighter alternative
- When to reach for which
- Real-world examples along the way

... and maybe mention a few bonus features while we are at it.

A brief history

SPL: the Standard PHP Library

A standard set of interfaces and classes for solving common problems with efficient data access.

- Iterator interface, shipped with PHP 5 (2004)
- Generators, added in PHP 5.5 (2013)

Both have only gotten more useful since.

Why?

Let your objects behave like native iterables.

foreach iteration

```
foreach ($array as $key => $value) {
    echo "{$key}: {$value}";
}
```

And, with Generators, do all of this with a single function.

A basic class

Imagine a Set object that loads data from a cache or DB:

```
class Set
{
    protected array $_set;

    public function __construct(?array $parameters = null) {
        $this->_set = $this->loadFromCache($parameters)
            ?? $this->loadFromDatabase($parameters)
            ?? [];
    }

    // loadFromCache() and loadFromDatabase() return ?array
}
```

Without these interfaces...

You're forced to expose internals to consumers, such as by creating getters:

```php
$myset = new Set();

// Want a single item? Add a get() method.
$item = $myset->get(3);

// Want to loop? Expose getAll() and hand out the array.
foreach ($myset->getAll() as $item) {
    /* ... */
}
```

Iterators

The Iterator interface

Five methods, all about managing position state — similar to a linked list:

`current()`:	Returns the current value
`key()`:	Returns the current value's access key
`next()`:	Moves the internal pointer to the next item
`rewind()`:	Resets the internal pointer to the first item
`valid()`:	Returns whether the pointer is at a valid item

```
interface Iterator extends Traversable {
    public function current(): mixed;
    public function key(): mixed;
    public function next(): void;
    public function rewind(): void;
    public function valid(): bool;
}
```

Implementing Iterator

When the data is already an array, the built-in pointer functions do the work:

```
class IterableSet extends Set implements Iterator
{
    public function current(): mixed  { return current($this->_set); }
    public function key(): mixed      { return key($this->_set); }
    public function next(): void      { next($this->_set); }
    public function rewind(): void    { reset($this->_set); }
    public function valid(): bool     { return key($this->_set) !== null; }
}
```

Now foreach just works

The object is iterable: no helper methods, no internal arrays exposed:

```php
$myset = new IterableSet();

foreach ($myset as $key => $item) {
    echo "{$key}: ", print_r($item, true), PHP_EOL;
}
```

Your object is now indistinguishable from an iterable,
to any consumer.

Related interfaces

Each makes your object behave more array-like:

Countable

One method: `count()`

Enables count($obj)

ArrayAccess

Four methods:

- `offsetGet`
- `offsetSet`
- `offsetExists`
- `offsetUnset`

Enables $obj[$key] = ...

But we're here for iteration, so let's keep moving.

The SPL Iterator Library

InfiniteIterator

Auto-rewinds when it reaches the end, loops forever.

```php
$forever = new InfiniteIterator(new IterableSet());

$count = 100;
foreach ($forever as $item) {
    print_r($item);
    if (!$count--) break;
}
```

LimitIterator

Take a slice: start offset and a max count.

```php
// First three items of our set
foreach (new LimitIterator(new IterableSet(), 0, 3) as $item) {
    print_r($item);
}

// Even cap an infinite iterator
$forever = new InfiniteIterator(new IterableSet());
foreach (new LimitIterator($forever, 0, 100) as $item) {
    print_r($item);
}
```

FilterIterator

Subclass it and define accept(): only matching items pass through.

```php
class ArrayOnlyFilter extends FilterIterator
{
    public function accept(): bool {
        return is_array($this->getInnerIterator()->current());
    }
}

foreach (new ArrayOnlyFilter(new IterableSet()) as $item) {
    print_r($item);
}
```

RegexIterator

A pre-built FilterIterator that takes a regex.

```php
$regex = new RegexIterator(
    new IterableSet(),
    '/^tek[0-9]+/',
    RegexIterator::MATCH
);

foreach ($regex as $item) {
    print_r($item);
}
```

Modes

MATCH

GET_MATCH

ALL_MATCHES

SPLIT

REPLACE

USE_KEY

MultipleIterator

Walk multiple iterators in parallel, stops when any one runs out.

```php
$multiple = new MultipleIterator();
$multiple->attachIterator(new IterableSet());
$multiple->attachIterator(new IterableSet(['other', 'parameters']));

foreach ($multiple as $both) {
    $one = print_r($both[0], true);
    $two = print_r($both[1], true);
    echo "One: {$one} | Two: {$two}", PHP_EOL;
}
```

RecursiveIterator + RecursiveIteratorIterator

Define `hasChildren()` and `getChildren()`: `RecursiveIteratorIterator` flattens the tree.

```php
class RecursableSet extends IterableSet implements RecursiveIterator
{
    public function hasChildren(): bool {
        return is_array(current($this->_set));
    }

    public function getChildren(): RecursiveIterator {
        return new RecursiveArrayIterator(current($this->_set));
    }
}

foreach (new RecursiveIteratorIterator(new RecursableSet()) as $item) {
    echo " {$item} ";
}
```

RecursiveDirectoryIterator

Pair it with RecursiveIteratorIterator: flatten an entire filesystem tree into one foreach.

```php
$dir = new RecursiveDirectoryIterator(
    '/var/log',
    RecursiveDirectoryIterator::SKIP_DOTS
);

foreach (new RecursiveIteratorIterator($dir) as $file) {
    if ($file->getExtension() === 'log') {
        echo $file->getPathname(),
             ' (', $file->getSize(), ' bytes)', PHP_EOL;
    }
}
```

$file is an SplFileInfo: getPathname(), getSize(), getMTime(), isDir(), all there.

CallbackFilterIterator

Same job as FilterIterator, no subclass: pass the predicate as a closure.

```php
$nums = new ArrayIterator(range(1, 20));

$evens = new CallbackFilterIterator(
    $nums,
    fn($n) => $n % 2 === 0
);

foreach ($evens as $n) {
    echo $n, ' ';   // 2 4 6 8 10 12 14 16 18 20
}
```

Most filter cases don’t deserve a whole class. Reach for this first.

AppendIterator

Concatenate iterators into one stream. Lazy: nothing is materialized.

```php
$weekdays = new ArrayIterator(['Mon', 'Tue', 'Wed', 'Thu', 'Fri']);
$weekend  = new ArrayIterator(['Sat', 'Sun']);

$week = new AppendIterator();
$week->append($weekdays);
$week->append($weekend);

foreach ($week as $day) {
    echo $day, ' ';   // Mon Tue Wed Thu Fri Sat Sun
}
```

Works on any Iterator, including Generators. Glue streams without ever holding them all in memory.

…and many more

The SPL ships with a small army of iterator classes:

- `ArrayIterator`
- `CachingIterator`
- `DirectoryIterator`
- `EmptyIterator`
- `FilesystemIterator`
- `GlobIterator`
- `NoRewindIterator`
- `ParentIterator`
- `RecursiveCachingIterator`
- `RecursiveCallbackFilterIterator`
- `RecursiveFilterIterator`
- `RecursiveRegexIterator`
- `RecursiveTreeIterator`
- `SeekableIterator`
- *…and more*

The catch

Iterator wants a class.

- Great when iteration is a property of a stateful object you already have
- Awkward when all you want is a one-shot sequence, and there was no class yet
- May have issues if the sequence is huge, infinite, or you can't materialize the whole thing

For everything in the "wrong fit" column...

Enter: Generators.

Generators

What is a Generator?

A function that produces a sequence of values, one at a time.

- Write it like a normal function, but use yield instead of return*
- Execution pauses on each yield, and resumes when the caller asks for the next value
- Calling it returns a Generator object that implements Iterator
- Works everywhere an Iterator works, including the SPL wrappers

The yield keyword

Compare an array-returning function to its generator equivalent:

Array approach

```php
function counter(): array {
    $result = [];
    for ($i = 1; $i <= 3; $i++) {
        $result[] = $i;
    }
    return $result;
}

foreach (counter() as $n) {
    echo $n;
}
```

Generator approach

```php
function counter(): Generator {
    for ($i = 1; $i <= 3; $i++) {
        yield $i;
    }
}

foreach (counter() as $n) {
    echo $n;
}
```

Memory efficiency

Only one item in memory at a time.

range(): allocates everything

```php
$sum = 0;
foreach (range(1, 99999) as $n) {
    $sum += $n;
}

// ~3 MB allocated
// for an array we throw away
```

yield: one at a time

```php
function ints(int $from, int $to): Generator {
    for ($i = $from; $i <= $to; $i++) {
        yield $i;
    }
}

$sum = 0;
foreach (ints(1, 99999) as $n) {
    $sum += $n;
}
// constant memory
```

Generators are Iterators

A Generator object implements Iterator: every SPL wrapper just works.

```php
function naturals(): Generator {
    $n = 1;
    while (true) {
        yield $n++;
    }
}

// Cap the infinite generator with a LimitIterator
foreach (new LimitIterator(naturals(), 0, 10) as $n) {
    echo $n, PHP_EOL;
}
```

Yielding keys

Same key => value syntax you already know from arrays.

```php
function userEmailToName(array $users): Generator {
    foreach ($users as $user) {
        yield $user->email => $user->name;
    }
}

foreach (userEmailToName($users) as $email => $name) {
    echo "{$email}: {$name}", PHP_EOL;
}
```

yield from: Delegation & Multiple Yields

Hand off to another iterable: array, Iterator, or another Generator.

```php
function firstNames(): Generator {
    yield from ['Alice', 'Bob', 'Carol'];
}

function allNames(): Generator {
    yield from firstNames();
    yield from ['Dave', 'Eve'];
    yield 'Frank';
}

foreach (allNames() as $name) { echo $name, PHP_EOL; }
```

Generators can return

A return statement immediately ends the generator.
Since PHP 7, a generator can also return a final value, fetched via getReturn().

```php
function countLines(string $path): Generator {
    $count = 0;
    foreach (new SplFileObject($path) as $line) {
        $count++;
        yield $line;
    }
    return $count;
}

$gen = countLines('access.log');
foreach ($gen as $line) { /* process */ }
echo 'Total: ', $gen->getReturn();
```

Real-world: reading large files

Stream a multi-GB log file line by line, constant memory.

```php
function readLines(string $path): Generator {
    $fp = fopen($path, 'r');
    try {
        while (($line = fgets($fp)) !== false) {
            yield $line;
        }
    } finally {
        fclose($fp);
    }
}

foreach (readLines('huge.log') as $line) {
    if (str_contains($line, 'ERROR')) echo $line;
}
```

Real-world: streaming DB results

fetchAll() loads everything. A generator hands you one row at a time.

```php
function queryRows(PDO $db, string $sql, array $params = []): Generator {
    $stmt = $db->prepare($sql);
    $stmt->execute($params);
    while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
        yield $row;
    }
}

foreach (queryRows($db, 'SELECT * FROM orders') as $order) {
    process($order);
}
```

Real-world: paginated APIs

Hide pagination from the consumer: they just see a sequence.

```php
function allCustomers(ApiClient $api): Generator {
    $page = 1;
    do {
        $response = $api->get('/customers', ['page' => $page]);
        yield from $response->items;
        $page++;
    } while ($response->hasMore);
}

foreach (allCustomers($api) as $customer) {
    syncToWarehouse($customer);
}
```

Generators

Sending Data In?

Two-way: ->send()

Generators are two-way streets: you can push values back in. (Coroutines, in essence.)

```
function logger(): Generator {
    while (true) {
        printf("[%s] %s\n", date('H:i:s'), yield);
    }
}

$log = logger();
$log->send('User signed in'); // [19:33:20] User signed in
$log->send('Cache miss');     // [19:33:21] Cache miss
```

In this case, the yield ‘gets’ the value...

Sending & Yielding

This works both ways, you can both ‘return values’ via yielding, while sending values in...

```php
function bartender(): Generator {
    $name = yield "What's your name?";
    $drink = yield "Hi $name, want a drink?";
    yield "One $drink coming up.";
}

$kallum = bartender();
echo $kallum->current(), PHP_EOL;             // What's your name?
echo $kallum->send("Donut"), PHP_EOL;         // Hi Donut, want a drink?
echo $kallum->send("Dirty Shirley"), PHP_EOL; // One Dirty Shirley coming up.
```

Use Cases for this are limited ... but it’s cool right?

Why this matters less than it used to

send() was the foundation of pre-Fiber async PHP. PHP 8.1 changed that.

- amphp v2, ReactPHP, recoil all rode on yield + send() as their coroutine primitive
- Fibers (PHP 8.1+) give real suspension: pause anywhere, no generator gymnastics
- Modern async PHP — amphp v3, Revolt, fiber-based React — doesn't use send() anymore
- send() lives on for state machines, interactive scripts, embedded DSLs … niche but real

Cool to know about. Maybe not the right answer in new code.

So, Useful?

When to use which

Reach for Iterator when...

- Iteration is a property of a stateful object
- You need rewind() to work: multiple passes over the same data
- You're building a reusable class that needs Countable / ArrayAccess too
- *IteratorAggregate + a generator is often the sweet spot*

Reach for Generator when...

- You just need a sequence, once
- Memory matters: huge files, big result sets, infinite streams
- You want lazy evaluation: only compute what's consumed
- The producer logic is easier to read as a function

Combine them

IteratorAggregate + a generator: full iterable class, almost no boilerplate.

```php
class Set implements IteratorAggregate, Countable
{
    public function __construct(private array $_set = []) {}

    public function getIterator(): Generator {
        foreach ($this->_set as $key => $value) {
            yield $key => $value;
        }
    }

    public function count(): int { return count($this->_set); }
}
```

Static Analysis

PHPStan + Generators

`Generator<TKey, TValue, TSend, TReturn>`: **four generics, one source of truth.**

```php
/** @return Generator<int, string, void, int> */
function readLines(string $path): Generator {
    $n = 0;
    foreach (new SplFileObject($path) as $line) {
        yield $n++ => $line;
    }
    return $n;
}

foreach (readLines('big.log') as $lineNo => $line) {
    // PHPStan knows: $lineNo is int, $line is string
}

$total = readLines('big.log')->getReturn();  // int
```

PHPStan will also flag wrong yield types and bad send() arguments.

PHPStan + Iterators

`Iterator<TKey, TValue>` / `IteratorAggregate<TKey, TValue>`: **foreach inherits the types.**

```php
/** @implements IteratorAggregate<int, Customer> */
class CustomerList implements IteratorAggregate, Countable
{
    /** @return Generator<int, Customer> */
    public function getIterator(): Generator {
        foreach ($this->rows as $i => $row) {
            yield $i => new Customer($row);
        }
    }

    public function count(): int { return count($this->rows); }
}

foreach (new CustomerList($rows) as $customer) {
    // PHPStan knows: $customer is Customer
}
```

Annotate once at the class. Every foreach, array_map, iterator_to_array call inherits the types.

Wrap up

- Iterator gives you foreach on your objects, for a price (5 methods, state).
- The SPL iterator library composes around anything iterable, your classes included.
- Generators let you write one function and skip the ceremony.
- Memory and lazy-loading are why Generators are a useful tool for sequences today.
- Mix and match: IteratorAggregate + Generator is a great pattern for stateful iterables.

Rate the talk:
joind.in/talk/df521

Thank you

Eli White

eliw.com

Slides will be posted there shortly

Questions?