Tip 41: Create Iterable Properties with Generators

In this tip, you'll learn how to convert complex data structures to iterables with generators.

We'll cover the following

In Tip 14, Iterate Over Key-Value Data with Map and the Spread Operator, you learned how simple it is to loop over maps thanks to iterables. And once you can iterate over a collection, you have access to the spread operator, array methods, and many other tools to transform your data. *Iterables* give your data *more* flexibility by allowing you to access each piece of data individually.

You also know that objects don't have a *built-in* iterator. You can't loop over an object directly—*you need to convert part of it to an array first*. That can be a major problem when you want the structure of an object but the flexibility of an *iterable*.

In this tip, you'll learn a technique that can make complex data structures as easy to use as simple arrays. You're going to use a new specialized function called a **generator** to return data one piece at time. In the process, you'll see how you can convert a deeply nested object into a simple structure.

Generators

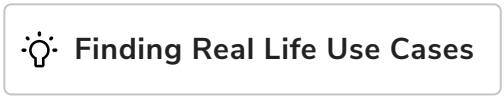
Generators aren't exclusive to classes. They're a *specialized* function. At the same time, they're very different from other functions. And while the JavaScript community has enthusiastically embraced most new features, they haven't quite figured out what to do with generators. In late 2016, a poll by Kent Dodds, a popular JavaScript developer, found that 81 percent of developers rarely or never used generators.

That's changing. Developers and library authors are discovering how to use

generators. One of the best use cases so far is to use generators to transform

objects into iterables.

What is a generator? The Mozilla Developer Network explains that a **generator** is a function that doesn't fully execute its body immediately when called.



This is different from a higher-order function, which fully executes but *returns* a new function. A generator is a *single* function that doesn't resolve its body immediately. What that means is that a generator is a function that has *break points* where it essentially *pauses* until the next step.

Using generators

To make a generator, you add an asterisk (`*`) after the `function` keyword. You then have access to a special method called `next()`, which returns a part of the function. Inside the function body, you *return* a piece of information with the keyword `yield`. When executing the function, use the `next()` method to get the information yielded by the function.

When you call `next()`, you get an object containing two keys: `value` and `done`. The item you declare with `yield` is the `value`. `done` indicates there are no items left.

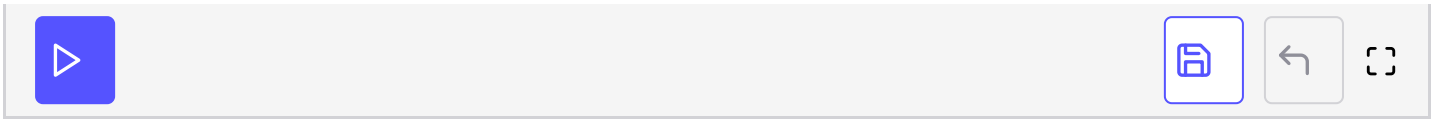
Example

For example, if you wanted to read Nobel Prize winner Naguib Mahfouz's Cairo Trilogy but you only wanted to know the titles one at a time, you'd write a function that would return the yields for each book in the trilogy. Each time you called `yield()`, you'd give the next book in the sequence.

To use the trilogy generator, you'd first have to call the function and assign it to a variable. You'd then call `next()` on the variable each time you wanted a new book.

```
function* getCairoTrilogy() {
    yield 'Palace Walk';
    yield 'Palace of Desire';
    yield 'Sugar Street';
}
const trilogy = getCairoTrilogy();
console.log(trilogy.next());
```

```
console.log(trilogy.next());
console.log(trilogy.next());
console.log(trilogy.next());
```



Notice how interesting that is. You can step through the function piece by piece. This is useful if you have lots of information and want to access it in parts. You could pull out one piece of information and pass the generator somewhere else to get the next piece. Like a *higher-order* function, you can use it in different places.

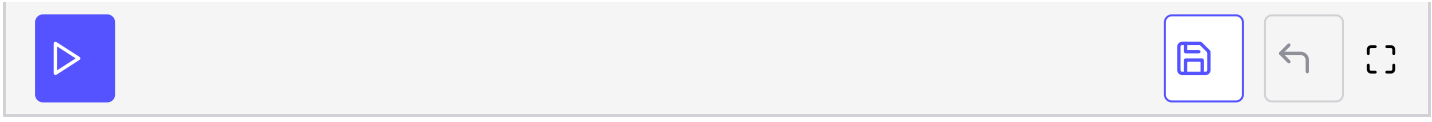
But that is not going to be your focus for this tip. Instead, it is far more interesting that generators turn a *function* into an *iterable*. Because you are accessing data one piece at a time, it is a simple step to turn them into *iterables*.

Generators & complex data structures

When you use a generator as an iterable, you don't need to use the `next()` method. Use any action that requires an iterable. The generator will go through the parts one at a time as if it were going through the indexes of an array or the keys of a map.

For example, if you want the Cairo trilogy in the form of an array, you'd simply use the *spread* operator.

```
function* getCairoTrilogy() {
    yield 'Palace Walk';
    yield 'Palace of Desire';
    yield 'Sugar Street';
}
const trilogy = [...getCairoTrilogy()];
console.log(trilogy);
```



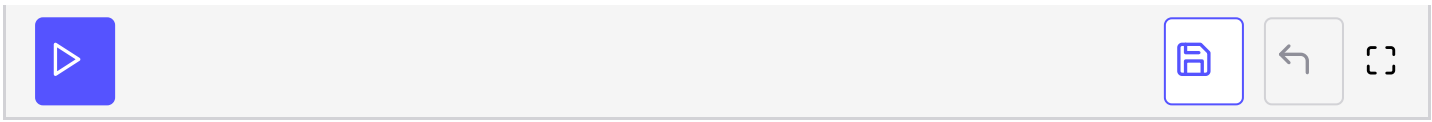
If you want to add all the books to your reading list, all you'd need is a simple `for...of` loop.

```
function* getCairoTrilogy() {
    yield 'Palace Walk';
    yield 'Palace of Desire';
    yield 'Sugar Street';
}

const readingList = {
```

```
    'Visit from the Goon Squad': true,
    'Manhattan Beach': false,
};

for (const book of getCairoTrilogy()) {
    readingList[book] = false;
}
console.log(readingList);
```



How does this fit into classes? Generators are awesome because, like getters and setters, they can give your classes a simple interface. You can make a class with a complex data structure but design it in such a way that developers using it will be able to access the data as if it were a simple array.

Consider a simple data structure: *a family tree* with a single branch. A person in a family tree would have a name and children. And each child would have children of their own.

A tree data structure would have advantages for *searches* and *lookups*, but flattening the information would be pretty difficult. You'd have to make a method to create an empty array and fill it with family members before returning.

```
class FamilyTree {
    constructor() {
        this.family = {
            name: 'Dolores',
            child: {
                name: 'Martha',
                child: {
                    name: 'Dyan',
                    child: {
                        name: 'Bea',
                    },
                },
            },
        };
    }
    getMembers() {
        const family = [];
        let node = this.family;
        while (node) {
            family.push(node.name);
            node = node.child;
        }
        return family;
    }
}
const family = new FamilyTree();
console.log(family.getMembers());
```

With a generator, you can return the data directly without pushing it to an array. As a bonus, your users wouldn't need to look up a method name. They could treat the property holding the family tree as if it were holding an array.

Converting the method to a generator is simple. You're just combining ideas from the method with ideas from your `getCairoTrilogy()` generator.

Start off by changing the method name from `getMembers()` to `* [Symbol.iterator] ()`. It looks confusing, but here's what's happening. First, the *asterisk* signifies that you're creating a generator. The phrase `Symbol.iterator` is attaching the generator to an iterable on the class. This is similar to how the map object has a `MapIterator`.

Inside the body of the method, add the `while` loop. Unlike your `getCairoTrilogy()` generator, you aren't going to `yield` an explicit value. Instead, you'll `yield` the value from each cycle of the loop. As long as there's something to return, the generator will keep going.

Instead of `family.push(node.name);`, all you need to do is `yield` the result: `yield node.name`. This means you don't need the intermediate array. Delete that. Everything else is the same

Now when you need any action that requires an iterable, such as the spread or the `for...of` loop, you can call it directly on the class instance.

```
class FamilyTree {
    constructor() {
        this.family = {
            name: 'Dolores',
            child: {
                name: 'Martha',
                child: {
                    name: 'Dyan',
                    child: {
                        name: 'Bea',
                    },
                },
            },
        };
    } *
        [Symbol.iterator]() {
        let node = this.family;
        while (node) {
            yield node.name;
            node = node.child;
        }
    }
```

```
}
const family = new FamilyTree();
console.log([...family]);
```

Is the extra complexity of the generator worth it? It depends on your goals. The advantage with a generator is that other developers don't need to get caught up in the implementation details of your class. They don't need to know that the class is actually using a tree data structure. To them, the class contains an iterable.

Of course, sometimes hiding complexity makes debugging more difficult. As with getters and setters, be careful about hiding too much from other developers. Still, when you want to use more complicated data structures but you don't want to burden others with implementation details, generators are a great solution.

1

Which of the following is true about the generator given below?

```
function* nums() {
  let val = 2;
  while (true) {
    yield val;
    val = val + 1
  }
}
```

2

What will the following code yield?

```
function* gen() {
    let curr = 0;
    let next = 1;

    while(true) {
        yield curr;
        let num = curr + next;
        curr = next;
        next = num;
    }
}

let iter = gen();

console.log(iter.next().value);
console.log(iter.next().value);
console.log(iter.next().value);
console.log(iter.next().value);
console.log(iter.next().value);
```

Retake Quiz

In the next tip, you'll see how context problems can sneak into classes and how you can solve them using `bind()`.