Generators

WE'LL COVER THE FOLLOWING

ES6 brings us a new type of function, the generator function. Generators are a function that allows us to pause and resume execution. Generators exist in many other languages and are used mostly for control flow.

The syntax to create a generator is the same as a regular function, however we use the `*` at the end of the `function` keyword to indicate we are creating a generator.

```
function* functionName() {

}
```

Generators return an Iterator Object, similar to Map, Set, and Array. An Iterator Object is an object that implements a `.next()` method. This means that our generator function can provide multiple results. There is also a new keyword introduced called `yield`, the `yield` keyword is used to actually pause the execution of the function and return the value provided.

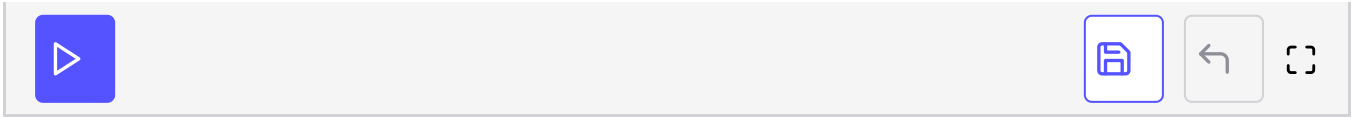
Our First Generator

Let's look at a simple example of a generator.

```
function* myGenerator() {
    yield "Isn't";
    yield "this";
    yield "cool!";
}
```

Inside of the generator we are able to `yield` as many values as you want. Here we are yielding three times, in order to get at those values we need to call our generator.

```
const generatorValues = myGenerator();
console.log(generatorValues.next());
/*
{
    value: "Isn't", done: false
}
*/
```

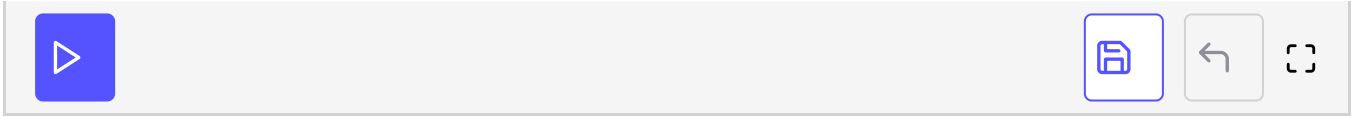


The next method returns an object that has the two keys on it, `value` and `done`. Until our generator has finished, the `done` value will always be false, and the `value` will be what we yield. Calling `.next()` three more times will look something like this.

```
console.log(generatorValues.next());
//{ value: 'this', done: false }

console.log(generatorValues.next());
//{ value: 'cool!', done: false }

console.log(generatorValues.next());
//{ value: undefined, done: true }
```



After we have exhausted our generator we can call it all we want, but it will only return the last object with `done` set to `true`. With this information in mind, let's look at a more complex example.

```
function* countUp(iterations) {
```

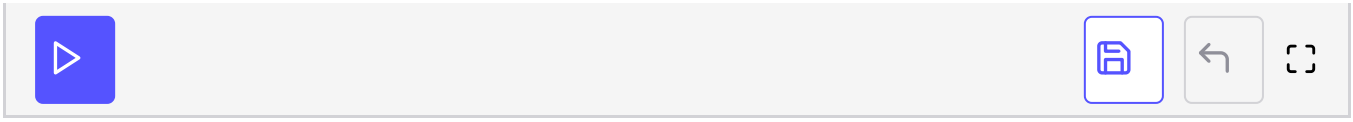
```
    let counter = 0;
    while(counter < iterations) {
        yield counter++;

    }
}
```

The `countUp` generator function here takes an argument that will be used as the number of iterations we want to go for. Using a `while` loop we can `yield` until we exhaust our iterations.

Remember that calling our generator function returns an Iterator Object. This means we can use the `for...of` loop to iterate over it.

```
for(let iteration of countUp(5)) {
  console.log(iteration);
}
// 0
// 1
// 2
// 3
// 4
```



Here instead of it providing the object with the `done` and `value` keys, it simply provides the value for us in the `iteration` variable.

yield*

Along with the regular `yield` keyword we can use the `*` as well to accept a value that it's self is an Iterator Object.

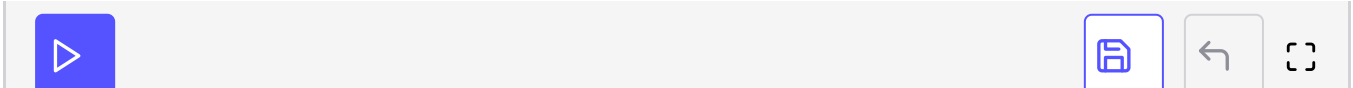
```
const locations = new Set(["Toronto","Montreal","New York"]);

function* locationGen() {
  yield* locations;
}

const myLocations = locationGen();

console.log(myLocations.next());
//{ value: 'Toronto', done: false }

console.log(myLocations.next());
//{ value: 'Montreal', done: false }
```



The `yield*` syntax will iterate over the object provided, in this case we are using a `Set` as the Iterable Object.

Passing a value to .next()

When working with the `yield` keyword we can also pass a value to the generator through the `.next()` method. Consider the `add()` function below.

```
function* add() {
  let total = 0;
  while(true) {
    let value = yield;
    total += value;
    console.log(total)
  }
}
```

Here we have a generator function called `add` that will have a running total. There is a while loop that will run forever if left to do so, however since this is a generator it will pause at the `yield` keyword.

We assign `yield` to a new `value` variable and add that to the total. Below you can see how we might call this.

```
const addFn = add();
console.log(addFn.next()); //Used to call the generator up to the yield
console.log(addFn.next(2)); //2 will be passed to yield and added to total
console.log(addFn.next(2)); //2 will be added to 2
console.log(addFn.next(2)); //2 will be added to 4
```

Now that we have seen how to pass a value back to a generator, let's see how we can use generators to create synchronous asynchronous calls.

Synchronous asynchronous calls

Working with asynchronous JavaScript can be complex, Promises have made this easier, and in the future `async/await` will help a lot. But there is a neat pattern that we can use now. This is a pattern I have seen on Axel Rauschmayer site, as well as talked about by Kyle Simpson. The code we will

be working with is similar that talked about by Kyle. This pattern is also implemented in a library by TJ Holowaychuk, https://github.com/tj/co.

We will be implementing a coroutine function that will allow us to call multiple async calls and get them back in the order we called them. Take a look at the code below.

```
const pokemon = coro(function* () {
    let pokemon1 = yield getPokemon(1,pokemon);
    let pokemon2 = yield getPokemon(242,pokemon);
    console.log(pokemon1,pokemon2);
});
pokemon();
```

Here we use a function called `coro` that takes a generator callback function. This function can yield as many times as we want. We yield a function called `getPokemon` that takes a pokemon id and takes the `pokemon` function so that we can pass the data back.

The `getPokemon` function looks something like this, we are using `fetch` to call the pokeapi and return the json version of the response. We run the callback and pass the response back.

```
function getPokemon(id,cb) {
    fetch(`http://pokeapi.co/api/v2/pokemon/${id}`)
        .then(res=>res.json())
        .then(res=>cb(res));
}
```

The `coro` function looks like this. We take in a generator function and call it. We return a function that itself returns the `.next()` method and apply the original callback function and the arguments we pass to it.

```
function coro(g) {
    let it = g();
    return function() {
        return it.next.apply(it,arguments);
    }
}
```

coro step by step

Let's walk through the `coro` function and how it works, since it is a little complex. We create a new function called `pokemon`, using the `coro` function we pass in a generator to use as our control flow. Inside of the `coro` function it takes the generator and returns a function that returns a call to the `next()` method and it passes in the generator again, as well as any arguments provided. In the first case it will pass along no data as no arguments have been passed to the function.

When it gets to this line:

```
let pokemon1 = yield getPokemon(1,pokemon);
```

We are making a called to the `getPokemon` function to make our ajax call. We pass `pokemon` back to it so that when the `getPokemon` call resolves it will pass that data to `yield` ultimately storing it in the `pokemon1` variable. We could do this over and over, each time the when we get to a new line, we wait for the data to come back before processing on.

I would suggest trying this code out in your browser, the whole thing looks like this:

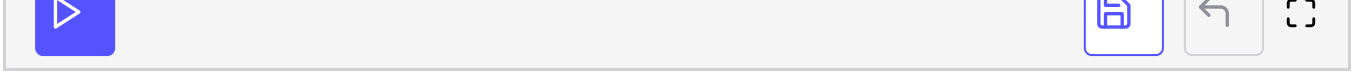
```
import "isomorphic-fetch"

function getPokemon(id,cb) {
    fetch(`http://pokeapi.co/api/v2/pokemon/${id}`)
        .then(res=>res.json())
        .then(res=>cb(res));
}

function coro(g) {
    let it = g();
    return function() {
        return it.next.apply(it,arguments);
    }
}

const pokemon = coro(function* () {
    let pokemon1 = yield getPokemon(1,pokemon);
    let pokemon2 = yield getPokemon(242,pokemon);
    console.log(pokemon1,pokemon2);
});

pokemon();
```



Try walking through your code with a `debugger` or simply some well placed `console.log` calls to watch how the data flows.

Additional Resources

- https://tc39.github.io/ecmascript-asyncawait/