Generators

In this lesson you'll learn all about generators and what to do with them.

WE'LL COVER THE FOLLOWING

What is a generator?

A generator function is a function that we can start and stop, for an indefinite amount of time. And also restart with the possibility of passing additional data at a later point in time.

To create a generator function we write like this:

```
function* fruitList(){
  yield 'Banana';
  yield 'Apple';
  yield 'Orange';
}

const fruits = fruitList();

fruits;
// Generator
console.log(fruits.next());
// Object { value: "Banana", done: false }
console.log(fruits.next());
// Object { value: "Apple", done: false }
console.log(fruits.next());
// Object { value: "Orange", done: false }
console.log(fruits.next());
// Object { value: undefined, done: true }
```

Let's have a look at the code piece by piece:

- we declared the function using `function*`
- we used the keyword `yield` before our content
- we start our function using `.next()`
- the last time we call `.next()` we receive and empty object and we get `done: true`

Our function is paused between each `.next()` call.

Looping over an array with a generator

We can use the `for of` loop to iterate over our generator and `yield` the content at each loop.

```
// create an array of fruits
const fruitList = ['Banana','Apple','Orange','Melon','Cherry','Mango'];

// create our looping generator
function* loop(arr) {
  for (const item of arr) {
    yield `I like to eat ${item}s`;
  }
}


const fruitGenerator = loop(fruitList);
console.log(fruitGenerator.next());
// Object { value: "I like to eat Bananas", done: false }
console.log(fruitGenerator.next());
// Object { value: "I like to eat Apples", done: false }
console.log(fruitGenerator.next().value);
// "I like to eat Oranges"
```

- Our new generator will loop over the array and print one value at a time every time we call `.next()`
- If you are only concerned about getting the value, then use `.next().value` and it will not print the status of the generator

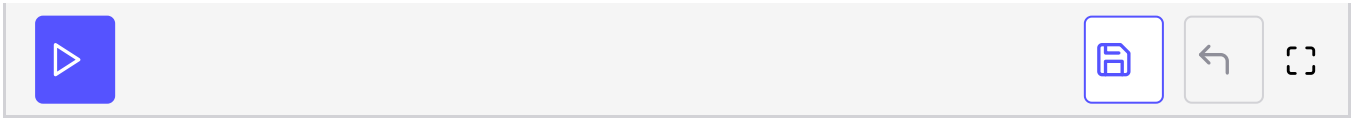
Finish the generator with `.return()`

Using `.return()` we can return a given value and finish the generator.

```
function* fruitList(){
  yield 'Banana';
  yield 'Apple';
  yield 'Orange';
}

const fruits = fruitList();

console.log(fruits.return());
// Object { value: undefined, done: true }
```

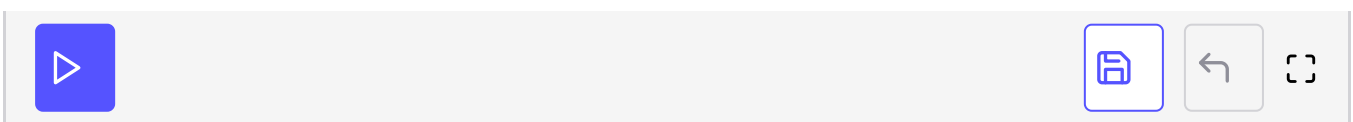


In this case we got `value: undefined` because we did not pass anything in the `return()`.

Catching errors with `.throw()`

```
function* gen(){
  try {
    yield "Trying...";
    yield "Trying harder...";
    yield "Trying even harder..";
  }
  catch(err) {
    console.log("Error: " + err );
  }
}

const myGenerator = gen();
console.log(myGenerator.next());
// Object { value: "Trying...", done: false }
console.log(myGenerator.next());
// Object { value: "Trying harder...", done: false }
console.log(myGenerator.throw("ooops"));
// Error: ooops
// Object { value: undefined, done: true }
```



As you can see when we called `.throw()`, the `generator` returned us the error and finished even though we still had one more `yield` to execute.

Combining Generators with Promises

As we have previously seen, Promises are very useful for asynchronous programming, and by combining them with generators we have a very powerful tool at our disposal to avoid problems like the *callback hell*.

As we are solely discussing ES6, I won't be talking about async functions as they were introduced in ES2017, but know that the way they work is based on what you will see now.

You can read more about async functions in the lesson ES2018: Async Iteration and more.

Using a generator in combination with a `Promise` will allow us to write asynchronous code that feels like synchronous code.

What we want to do is wait for a promise to resolve and then pass the resolved value back into our generator in the `.next()` call.

```
const myPromise = () => new Promise((resolve) => {
  resolve("our value is...");
});

function* gen() {
  let result = "";
  // returns promise
  yield myPromise().then(data => { result = data }) ;
  // wait for the promise and use its value
  yield result + ' 2';
};

// Call the async function and pass params
const asyncFunc = gen();
const val1 = asyncFunc.next();
console.log(val1);
// call the promise and wait for it to resolve
// {value: Promise, done: false}
val1.value.then(() => {
  console.log(asyncFunc.next());
})
// Object { value: "our value is... 2", done: false }
```

The first time we call `.next()` at line 15 it will call our promise and wait for it to resolve (in our simple example it resolves immediately). And when we call `.next()` again at line 18 it will utilize the value returned by the promise to do something else (in this case just interpolate a string).

Now, let's take a quiz and a coding challenge to test the concepts covered in this lesson.