Chapter 41: Generators

Generator functions (defined by the `function*` keyword) run as coroutines, generating a series of values as they're requested through an iterator.

Section 41.1: Generator Functions

A *generator function* is created with a `function*` declaration. When it is called, its body is **not** immediately executed. Instead, it returns a *generator object*, which can be used to "step through" the function's execution.

A `yield` expression inside the function body defines a point at which execution can suspend and resume.

```
function* nums() {
    console.log('starting');  // A
    yield 1;                  // B
    console.log('yielded 1'); // C
    yield 2;                  // D
    console.log('yielded 2'); // E
    yield 3;                  // F
    console.log('yielded 3'); // G
}
var generator = nums(); // Returns the iterator. No code in nums is executed

generator.next();  // Executes lines A,B returning { value: 1, done: false }
// console: "starting"
generator.next();  // Executes lines C,D returning { value: 2, done: false }
// console: "yielded 1"
generator.next();  // Executes lines E,F returning { value: 3, done: false }
// console: "yielded 2"
generator.next();  // Executes line G returning { value: undefined, done: true }
// console: "yielded 3"
```

Early iteration exit

```
generator = nums();
generator.next(); // Executes lines A,B returning { value: 1, done: false }
generator.next(); // Executes lines C,D returning { value: 2, done: false }
generator.return(3); // no code is executed returns { value: 3, done: true }
// any further calls will return done = true
generator.next(); // no code executed returns { value: undefined, done: true }
```

Throwing an error to generator function

```
function* nums() {
    try {
        yield 1;                  // A
        yield 2;                  // B
        yield 3;                  // C
    } catch (e) {
        console.log(e.message);    // D
    }
}

var generator = nums();

generator.next();  // Executes line A returning { value: 1, done: false }
generator.next();  // Executes line B returning { value: 2, done: false }
generator.throw(new Error("Error!!"));  // Executes line D returning { value: undefined, done: true}
// console: "Error!!"
generator.next();  // no code executed. returns { value: undefined, done: true }
```

Section 41.2: Sending Values to Generator

It is possible to *send* a value to the generator by passing it to the `next()` method.

```
function* summer() {
    let sum = 0, value;
    while (true) {
        // receive sent value
        value = yield;
        if (value === null) break;

        // aggregate values
        sum += value;
    }
    return sum;
}
let generator = summer();

// proceed until the first "yield" expression, ignoring the "value" argument
generator.next();

// from this point on, the generator aggregates values until we send "null"
generator.next(1);
generator.next(10);
generator.next(100);

// close the generator and collect the result
let sum = generator.next(null).value;  // 111
```

Section 41.3: Delegating to other Generator

From within a generator function, the control can be delegated to another generator function using `yield*`.

```
function* g1() {
  yield 2;
  yield 3;
  yield 4;
}

function* g2() {
  yield 1;
  yield* g1();
  yield 5;
}

var it = g2();

console.log(it.next()); // 1
console.log(it.next()); // 2
console.log(it.next()); // 3
console.log(it.next()); // 4
console.log(it.next()); // 5
console.log(it.next()); // undefined
```

Section 41.4: Iteration

A generator is *iterable*. It can be looped over with a `for...of` statement, and used in other constructs which depend on the iteration protocol.

```
function* range(n) {
    for (let i = 0; i < n; ++i) {
        yield i;
    }
}

// looping
for (let n of range(10)) {
    // n takes on the values 0, 1, ... 9
}

// spread operator
let nums = [...range(3)];  // [0, 1, 2]
let max = Math.max(...range(100));  // 99
```

Here is another example of use generator to custom iterable object in ES6. Here anonymous generator function `function *` used.

```
let user = {
  name: "sam", totalReplies: 17, isBlocked: false
};

user[Symbol.iterator] = function *(){

  let properties = Object.keys(this);
  let count = 0;
  let isDone = false;

  for(let p of properties){
    yield this[p];
  }
};

for(let p of user){
  console.log( p );
}
```

Section 41.5: Async flow with generators

Generators are functions which are able to pause and then resume execution. This allows to emulate async functions using external libraries, mainly q or co. Basically it allows to write functions that wait for async results in order to go on:

```
function someAsyncResult() {
    return Promise.resolve('newValue')
}

q.spawn(function * () {
    var result = yield someAsyncResult()
    console.log(result) // 'newValue'
})
```

This allows to write async code as if it were synchronous. Moreover, try and catch work over several async blocks. If the promise is rejected, the error is caught by the next catch:

```
function asyncError() {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
```

```
            reject(new Error('Something went wrong'))
        }, 100)
    })
}

q.spawn(function * () {
    try {
        var result = yield asyncError()
    } catch (e) {
        console.error(e) // Something went wrong
    }
})
```

Using co would work exactly the same but with `co(function * (){...})` instead of `q.spawn`

Section 41.6: Iterator-Observer interface

A generator is a combination of two things - an `Iterator` and an `Observer`.

Iterator

An iterator is something when invoked returns an `iterable`. An `iterable` is something you can iterate upon. From ES6/ES2015 onwards, all collections (Array, Map, Set, WeakMap, WeakSet) conform to the Iterable contract.

> A generator(iterator) is a producer. In iteration the consumer `PULL`s the value from the producer.

Example:

```
function *gen() { yield 5; yield 6; }
let a = gen();
```

Whenever you call `a.next()`, you're essentially `pull`-ing value from the Iterator and `pause` the execution at `yield`. The next time you call `a.next()`, the execution resumes from the previously paused state.

Observer

A generator is also an observer using which you can send some values back into the generator.

```
function *gen() {
  document.write('<br>observer:', yield 1);
}
var a = gen();
var i = a.next();
while(!i.done) {
  document.write('<br>iterator:', i.value);
  i = a.next(100);
}
```

Here you can see that `yield 1` is used like an expression which evaluates to some value. The value it evaluates to is the value sent as an argument to the `a.next` function call.

So, for the first time `i.value` will be the first value yielded (1), and when continuing the iteration to the next state, we send a value back to the generator using `a.next(100)`.

Doing async with Generators

Generators are widely used with `spawn` (from taskJS or co) function, where the function takes in a generator and allows us to write asynchronous code in a synchronous fashion. This does NOT mean that async code is converted to sync code / executed synchronously. It means that we can write code that looks like `sync` but internally it is still `async`.

> Sync is BLOCKING; Async is WAITING. Writing code that blocks is easy. When PULLing, value appears in the assignment position. When PUSHing, value appears in the argument position of the callback.

When you use iterators, you `PULL` the value from the producer. When you use callbacks, the producer `PUSH`es the value to the argument position of the callback.

```
var i = a.next() // PULL
dosomething(..., v => {...}) // PUSH
```

Here, you pull the value from `a.next()` and in the second, `v => {...}` is the callback and a value is `PUSH`ed into the argument position `v` of the callback function.

Using this pull-push mechanism, we can write async programming like this,

```
let delay = t => new Promise(r => setTimeout(r, t));
spawn(function*() {
  // wait for 100 ms and send 1
  let x = yield delay(100).then(() => 1);
  console.log(x); // 1

   // wait for 100 ms and send 2
  let y = yield delay(100).then(() => 2);
  console.log(y); // 2
});
```

So, looking at the above code, we are writing async code that looks like it's `blocking` (the yield statements wait for 100ms and then continue execution), but it's actually `waiting`. The `pause` and `resume` property of generator allows us to do this amazing trick.

How does it work ?

The spawn function uses `yield promise` to PULL the promise state from the generator, waits till the promise is resolved, and PUSHes the resolved value back to the generator so it can consume it.

Use it now

So, with generators and spawn function, you can clean up all your async code in NodeJS to look and feel like it's synchronous. This will make debugging easy. Also the code will look neat.

This feature is coming to future versions of JavaScript - as `async...await`. But you can use them today in ES2015/ES6 using the spawn function defined in the libraries - taskjs, co, or bluebird