Generators in JavaScript

CPEN400A: Lecture 10

Outline

What are generators ?

Creating a simple generator

Iterating with a generator

Combining generators with promises

What is a generator ?

Generators are functions in JavaScript that can pause their execution at a certain point and resume from where they left off

Used to “remember” state of the function at the point where it left off

Solves the “inversion of control” problem encountered with call-backs

Inversion of control

Consider an array in JavaScript on which you want to perform some operation, say add a constant to all the numbers in the array.

The traditional way to do this is with a callback function and higher-order functions, i.e., pass the function as a call-back to another function that does the operation

However, this breaks the traditional control flow where the operation to be done (i.e., adding a constant) is separated from the iteration operation

- Makes code hard to read as one has to trace flow to a different call
- Need specialized asynchronous functions for each operation - proliferation

Enter generators

Generators solve the inversion of control problem, by providing a generic way to perform the iteration, so the operation doesn't have to be done in a call-back

Are very powerful when combined with Promises for async operations (later)

```
var g = generatorFunc();     // this is the generator function

for  (v of g) {

      // Perform operation on v directly

}
```

Outline

What are generators ?

Creating a simple generator

Iterating with a generator

Combining generators with promises

Generator function

These are specialized functions in ES6 and later

Denoted with a ‘*’ in the function declaration (function* foo())

Has a special kind of return called a ‘yield’ to pause the execution at that point, and resume the execution on a subsequent call to the generator

Yield can also return a value to the caller - typically value being iterated on

When end of function is reached, it returns ‘undefined’ (can also specify a return) value, and the generator is DONE. Cannot be resumed after that point ever.

SimpleGenerator

Consider the code in simple-generator.js

Returns a sequence of numbers starting with an initial value and a step size

Some points to note:

1. This is an infinite generator, and as such never terminates (in theory)
2. Generator automatically remembers its state from the previous invocation
3. Numbers are computer ‘lazily’ on demand with each call to the generator
4. This generator doesn’t ever return a value, only yields

Class activity - 1

Write a simple generator function to yield all the factors of a natural number (including itself) - the function should only calculate the factors on demand and not ahead of time. You can assume that the input has been validated already.

Solution: factor-generator.js

Outline

What are generators ?

Creating a simple generator

Iterating with a generator

Combining generators with promises

Iterating with generators

After a call to the generator function, you receive an object that you'd use for iteration by calling *next* method on it - this goes to the next yield in the function

The object returned by yield has a field called 'done' to denote end of iteration

There are 2 methods of iterating over generators:

1. Using a while loop, and checking explicitly for done (ugly method)
2. Using a 'for-of' loop, and checking implicitly for done (elegant method)

Yield

First time the next is called: executes until a ‘yield’ is encountered

- Control is then ceded back to the calle function (but state is remembered)

Each subsequent call to the yield resumes execution from after where the first ‘yield’ left off, and executes until the next yield statement (state is remembered), or a return statement or throw occurs in the function (the generator is terminated)

IMPORTANT: Yield can only be called within generator function itself, not even in nested functions or from asynchronous callbacks (we’ll talk about this later)

Value returned by yield

Yield returns an object called *IteratorResult* consisting of

1. value: represents the actual value returned by the yield statement
2. done: a boolean flag representing whether or not the iteration is finished

It's caller's responsibility to check value of 'done'. In addition, the generator can

a. Throw an exception - this can be caught via a try-catch statement
b. Return a value (using 'return') - sets the value of *IteratorResult* to value and the done property to *true*. Note that not returning a value will still set the done property to *true* and the value to 'undefined'

Example Code

Look at the code in iterator.js - it iterates over a list of names with a generator

The generator function takes the list as an argument, and yields one name at a time, in the list order.

The code demonstrates two ways to do the iteration - the ugly way and the elegant way. These are semantically identical to each other, with one difference.

- The value returned by the generator (not yielded) cannot be retrieved in the second method
- Can be

Class Activity - 2

Can you think of a way to get the number of elements iterated over by the generator function in the second method of iteration as well (using for-of loops) ?

HINT: Think of how else you can exit a generator function

Solution in iterator2.js

Activity 3

Write a generator function to iterate over a large string, and find all occurrences of a specific word in it. The generator should match the word on demand only and not pre-compute the matches. Also, after each match, it should start from the next location until the end of the string is reached. It should yield the number of characters traversed in the string after the previous match (until the current one)

Write another function to iterate using the generator function and count *all* occurrences of a given word in a string.

Solution: word-match.js

Outline

What are generators ?

Creating a simple generator

Iterating with a generator

Combining generators with promises

Generators and asynchronous call-backs

One of the main advantages of Generators is that they can be used to write code in a “natural” style, without using callbacks and closures

However, generators cannot (easily) call asynchronous functions…

Recall that callbacks defined inside the generators cannot call yield (as they’re not executing in the context of the generator function)

One possible solution: Using promises together with generators

Generators and Promises

Yield a promise in each call of the next() method

- Promise can be resolved or rejected later
- Attach then and catch handlers to the promises as usual

You can also send the results of the resolved promise to the generator as an argument of the next function to chain them (outside the scope of this class)

Makes for cleaner code than attaching multiple layers of call-backs to promises

Example code

Code: reader-generator.js

Functionality: Takes an array of files, and uses a generator to iterate over them - each time it returns a promise object for the read that can be either resolved or rejected depending on whether the read was successful.

Comments:

1. Yield does not know the state of the promise at the time of yielding
2. Need to attach .then etc. to the promises that are yielded by the generator

Class Activity - 4

Write a generator function that takes an array of function as the first argument and executes them asynchronously, each after a specified delay (second argument). The generator function should return a set of promises, one after the other, for each function in the list with the appropriate delays. It should also pass to each function a parameter representing the current index of the function in the list.

Iterate over the list using the generator and attach a .then and .catch clause to each of the promises yielded by the generator.

Solution: asyncGenerator.js