Generators: the Gnarly Bits

By Jeff Posnick
Web DevRel @ Google

The ECMAScript 6 draft specification has already yielded many sources of joy for the modern JavaScript developer. We covered some new collections classes and `for..of` iteration loops in a previous post. In this post, we're going to talk about something that goes hand-in-hand with `for..of` loops: generator functions.

There's a host of great material out there already that covers the why and how of using generators. In a nutshell, generators are special functions which create iterators, and iterators are objects that have a `next()` method, which can be called to obtain a value. Within a generator function, the keyword `yield` provides the value for `next()`. Using `yield` *suspends* execution of the generator function, preserving the state until `next()` is called again, at which point the code starts back up and continues, until it `yield`s another value (or until the generator function terminates). There are several canonical use cases for generator functions, like using them to iterate over the numbers in the Fibonacci sequence.

With the basics out of the way, let's go in-depth with a JavaScript sample that covers some of the gotchas, or "gnarly bits", of working with generators. There are extensive comments throughout, and you can play around with the live version of the code before reading through it:

So what are some big takeaways from the code?

First, constructing a generator results in a unique iterator with it's own distinct state, and you can pass in parameters to the generator constructor that can control behavior.

Second, you can pass in a parameter when calling an iterator's `next()` method, and that value will be assigned to whatever's on the left-hand side of the `yield` statement from the previous iterator invocation. This is a great way to vary the output of the iterator—here, we use it to control whether the word that's yielded is in uppercase or not. If you want to influence the very first value that's yielded, do it via a parameter to the generator's constructor.

Finally, generators can produce either finite or infinite iterators. If you're working with an infinite iterator, make sure that you have some sort of terminal condition based on the value `yield`ed—it's very easy to accidentally write infinite loops, especially when using `for..of` for

iteration. If you're working with a finite iterator via calls to `next()`, then the `.done` property of the object that's returned signals whether iteration is complete.

We hope this sample, along with the other resources available on the web, yields some excitement and gets you thinking about how you can use generators in your own code. Versions of Firefox starting with 31 and Chrome starting with 39 support generators natively. The Regenerator project offers generator support for other browsers, and using Traceur is an option as well.

Thanks to Erik Arvidsson for his help reviewing this article.

Last updated July 2, 2018.