JavaScript Iterators and Generators

by
R. Mark Volkmann, Partner
Object Computing, Inc. (OCI)

Introduction

The latest version of JavaScript defined by ECMAScript 2015 (a.k.a. ES6) adds many new features. While most are easy to understand, iterators and generators require a bit more effort. Everything you need to know to get started using them is provided here.

This article assumes that you are familiar with several ES 2015 features including arrow functions, classes, destructuring, for-of loop, let/const, and the spread operator. If you need to brush up on these, check out Luke Hoban's overview at https://github.com/lukehoban/es6features and my slides on ES 2015 at http://ociweb.com/mark. There is also a video of a talk I gave on ES 2015 at here.

Iterators are objects that have a `next` method. They are used to visit elements in a sequence. It is possible for the values in the sequence to be generated in a lazy manner.

The `next` method returns an object with `value` and/or `done` properties. It's best to return a new object from each call to `next` because callers might cache the object that is returned and not examine its properties until later. When the end of the sequence is reached, the `done` property will be true. Otherwise this property can be omitted since it will then be undefined which is treated as false (`return {value: some-value}`). For infinite sequences, the `done` property never becomes true.

Whether the `value` property has meaning when the `done` property is true depends on the iterator. For most iterators, the `value` property is not used when the `done` property is true. The three language constructs that consume iterables, the for-of loop, spread operator, and destructuring follow this convention. These are discussed in more detail later.

When the end of the sequence has been reached, the `value` property can be omitted (`return {done: true}`).

Iterables are objects that have a method whose name is the value of `Symbol.iterator`. This method returns an iterator object.

An object can be both an iterable and an iterator. When this is the case, the method with the name `Symbol.iterator` returns an iterator when it is called on the object and that same object has the `next` method required by iterators. Therefore, `obj[Symbol.iterator].call(obj) === obj`.

Iterable/Iterator Example

The following example generates numbers in the Fibonacci sequence. The object referred to by the variable `fibonacci` is an iterable. The `Symbol.iterator` method returns an iterator. Use of the fibonacci variable is illustrated with a for-of loop. Note that this loop breaks out when a value greater than 100 is returned. This is necessary since the sequence is infinite.

```
1  const fibonacci = {
2    [Symbol.iterator]() {
3      let n1 = 0, n2 = 1, value;
```

```
    return {
      next() {
        // The next line performs parallel assignment using destructuring.
        // It is equivalent to value = n1; n1 = n2; n2 = n1 + n2;
        [value, n1, n2] = [n1, n2, n1 + n2];

        // The next line is equivalent to return {value: value};
        return {value};
      }
    };
  }
};

// Note that "let" could be used in place of "const" on the next line,
// but "const" is more correct here because each iteration
// gets a new binding for the loop variable n
// and it is not modified in the loop body.
for (const n of fibonacci) {
  if (n > 100) break;
  console.log(n);
  // outputs 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, and 89
}
```

Iterable Objects

Objects from these builtin classes are iterable:

- `Array` - iterates over elements
- `Set` - iterates over elements
- `Map` - iterates over key/value pairs as [key, value]
- DOM `NodeList` - iterates over `Node` objects (requires browser support)

Primitive strings are iterable over their Unicode (UTF-16) code points, each occupying two or four bytes.

These methods on `Array` (including typed arrays), `Set`, and `Map` return an iterator:

- `entries` - over key/value pairs as [key, value]
- `keys` - over keys
- `values` - over values

The objects returned by these methods are both iterables and iterators.

For arrays, keys are indices. For sets, keys are the same as values.

Custom objects can be made iterable by adding a `Symbol.iterator` method. We'll see an example of this below.

Ordinary objects, such as those created from object literals, are not iterable. When this is desired, either use the `Map` class or write a function like the following:

```
function objectEntries(obj) {
  let index = 0;
  let keys = Reflect.ownKeys(obj); // This gets both string and symbol keys.
  return { // The object returned is both an iterable and an iterator.
    [Symbol.iterator]() { return this; },
    next() {
      if (index === keys.length) return {done: true};
```

```
        let k = keys[index++], v = obj[k];
        return {value: [k, v]};
      }
    };
  }

  let obj = {foo: 1, bar: 2, baz: 3};
  for (const [k, v] of objectEntries(obj)) {
    console.log(k, 'is', v);
  }
```

To avoid iterating over symbol keys, use `Object.getOwnPropertyNames(obj)` instead of `Reflect.ownKeys(obj)`.

An alternative to the function above is to use `Reflect.enumerate(obj)` to get an iterable over just the keys of an object.

Iterable Consumers

There are several new language constructs that consume iterables.

for-of Loop

```
for (const value of someIterable) { ... } // This iterates over all values.
```

spread Operator

```
// This can add all values from an iterable into a new array.
let arr = [firstElem, ...someIterable, lastElem];

// This can use all values from an iterable as arguments
// to a function, method, or constructor call.
someFunction(firstArg, ...someIterable, lastArg);
```

Positional Destructuring

```
let [a, b, c] = someIterable; // This gets the first three values.
```

Several constructors and methods of provided classes consume iterables. The `Set` constructor takes an iterable over values for initializing a new `Set`. The `Map` constructor takes an iterable over key/value pairs for initializing a new `Map`. The `Promise` methods `all` and `race` take an iterable over promises.

Generators

Generators are a special kind of iterator that is also iterable. They can be paused and resumed via multiple return points, each specified using `yield` keyword. The `yield` keyword can only be used in generator functions. Each call to `next` returns the value of the next `yield` expression. To yield a single value, use `yield value`. To yield each value returned by an iterable one at a time, use `yield* iterable`. Note that this iterable can be another generator, or even the same kind of generator obtained recursively.

A generator exits by running off the end of the function that defines it, returning a specific value using `return` keyword, or throwing an error. The done property will be true after any of these and will remain true.

A "generator function" returns a generator object. These are defined using `function*` instead of `function`. Generator functions can be defined in class definitions by preceding a method name with *.

A Basic Generator

```
// This is a generator function.
function* myGenFn() {
  yield 1;
  yield 2;
  return 3;
}

let myGen = myGenFn(); // This creates a generator.
console.log(myGen.next()); // {value: 1, done: false}
console.log(myGen.next()); // {value: 2, done: false}
console.log(myGen.next()); // {value: 3, done: true}

for (const n of myGenFn()) {
  // This outputs 1, then 2, but not 3 because done is true for this value.
  console.log(n);
}
```

Note that without the `return` statement in the generator, the call to `next` that returns a value of 3 would instead return a value of `undefined`.

Fibonacci Generator

Earlier we saw an example of generating numbers in the Fibonacci sequence using an iterable. We can generate the same sequence with less code by using a generator.

```
function* fibonacci() {
  let [prev, curr] = [0, 1];
  yield prev;
  yield curr;
  while (true) {
    [prev, curr] = [curr, prev + curr];
    yield curr;
  }
}

for (const n of fibonacci()) {
  if (n > 100) break;
  console.log(n);
}
```

This can also be implemented as an object that contains a generator method.

```
let fib = {
  * [Symbol.iterator]() {
    let [prev, curr] = [0, 1];
    yield prev;
    yield curr;
    while (true) {
      [prev, curr] = [curr, prev + curr];
      yield curr;
    }
  }
};

```

```
for (const n of fib) {
  if (n > 100) break;
  console.log(n);
}
```

This second approach, using an object with a generator method, is primarily useful for objects that will have multiple methods. Otherwise the first approach, using a generator function, is preferred.

Generator Methods

Three methods on generators affect their state.

- `next`

  This method gets the next value, similar to the iterator `next` method. It differs in that it takes an optional argument, but not on the first call. The optional argument specifies the value that the `yield` hit in this call will return at start of processing for the next call. It allows generators to act as data consumers.

- `return`

  This method takes a value and terminates the generator from the outside just as if the generator returned the specified value.

- `throw`

  This method takes an error description (typically an `Error` object) and terminates the generator from the outside just as if the generator used the `throw` keyword. It throws the error inside the generator at the yield where execution was paused. If the generator catches the error and yields a value, the generator will not be terminated. Otherwise it is terminated.

`Array` Methods

The `Array` class defines many methods that evaluate, find, filter, and transform contained elements. It would be useful if similar functions were available for any iterable sequence. ES 2015 does not provide these and they will likely not be provided in ES 2016. Before showing how these can be implemented, here's a review of the relevant `Array` methods.

- `includes` - determines whether a collection contains a given value
- `indexOf` - finds the index of the first occurrence of a given value
- `lastIndexOf` - finds the index of the last occurrence of a given value
- `find` - finds the first element that meets some condition
- `findIndex` - finds the index of first element that meets some condition
- `every` - determines whether every element meets a condition
- `some` - determines whether some element meets a condition
- `filter` - generates a new collection of elements that meet a condition
- `map` - generates a new collection of elements that are the results of passing each element to a given function
- `forEach` - passes each element to a given function one at a time
- `reduce` - calculates the final result of applying a given function to the previous result and the next element

star-it Library

"star-it" is a library of functions that take an iterable and mimic the functionality of many `Array` methods. The name comes from "star" for the asterisk wildcard character, representing the many `Array` methods that are mimiced, and "it" for iterable. This library is available in Github at https://github.com/mvolkmann/star-it. It is also available in NPM under the name "star-it" and can be installed by running "`npm install star-it`".

To run the tests for this library,

1. install Node.js
2. clone the star-it Github repo
3. cd to the star-it directory
4. run `npm install`
5. run `npm test`

Next we will walk through code from the library. It provides good examples of working with iterables and generators and reinforces what you have learned about them so far. Each function is accompanied by Jasmine test assertions that demonstrate how to use the function.

Note that only the `filter`, `map`, `skip`, and `take` methods make sense when working with infinite sequences (where the `done` property is never set to true).

The tests utilize an array (`arradd`, `isEven`, and `isOdd`), and a class (`TreeNode`). Here is the code that implements these:

```
const arr = [1, 3, 5, 6, 7, 3, 1];

const add = (x, y) => x + y;
const isEven = x => x % 2 === 0;
const isOdd = x => x % 2 === 1;

class TreeNode {
  constructor(value) {
    this.value = value;
    this.children = [];
    this.depthFirst = true;
  }

  addChildren(...children) {
    this.children.push(...children);
  }

  // This traverses all descendants of this TreeNode,
  // depth-first if this.depthFirst = true (the default)
  // or breadth-first otherwise.
  * [Symbol.iterator]() {
    if (this.depthFirst) {
      for (const child of this.children) {
        yield child;
        yield* child; // This yields all of its children.
      }
    } else { // breadth-first
      let queue = this.children, newQueue;
      while (queue.length) {
        // Yield all nodes at current level.
        yield* queue;
        // Get all children one level down.
        newQueue = [];
        for (const child of queue) {
          newQueue.push(...child.children);
```

```
          }
          queue = newQueue;
        }
      }
    }
  }
```

The functions in the star-it library do some verification of the types of arguments passed to them. There are verifications that an object is a function, iterator, or iterable. Study these functions to confirm your understanding of the requirements for implementing iterators and iterables.

```
function assertIsFunction(value) {
  if (typeof value !== 'function') {
    throw new Error('expected a function, but got', value);
  }
}

function assertIsIterator(value) {
  const nextFn = value.next;
  if (!nextFn || typeof nextFn !== 'function') {
    throw new Error('expected an iterator, but got', value);
  }
}

function assertIsIterable(value) {
  const iteratorFn = value[Symbol.iterator];
  if (!iteratorFn || typeof iteratorFn !== 'function') {
    throw new Error('expected an iterable, but got', value);
  }

  // Obtain an iterator from the iterable.
  const iterator = iteratorFn.apply(value);
  assertIsIterator(iterator);
}
```

And now, the functions in the star-it library! A common feature of these functions is that each is fairly short and relatively easy to understand. Understanding them will take you a long way toward being able to use and implement iterables, iterators, and generator functions. The code that follows each function definition is a test snippet that demonstrates its use.

every

```
function every(obj, predicate) {
  assertIsIterable(obj);
  assertIsFunction(predicate);
  for (const element of obj) {
    if (!predicate(element)) return false;
  }
  return true;
}
```

```
expect(starIt.every(arr, isOdd)).toBeFalsy();
```

filter

```
function* filter(obj, predicate) {
  assertIsIterable(obj);
  assertIsFunction(predicate);
  for (const element of obj) {
    if (predicate(element)) yield element;
  }
```

```
}
```

```
let iterable = starIt.filter(arr, isOdd);
let result = [...iterable];
expect(result).toEqual([1, 3, 5, 7, 3, 1]);
```

find

```
function find(obj, predicate) {
  assertIsIterable(obj);
  assertIsFunction(predicate);
  for (const element of obj) {
    if (predicate(element)) return element;
  }
  return undefined;
}
```

```
expect(starIt.find(arr, isEven)).toBe(6);
```

findIndex

```
 function findIndex(obj, predicate) {
   assertIsIterable(obj);
   assertIsFunction(predicate);
   let index = 0;
   for (const element of obj) {
     if (predicate(element)) return index;
     index++;
   }
   return -1;
 }
```

```
expect(starIt.findIndex(arr, isEven)).toBe(3);
```

forEach

```
function forEach(obj, fn) {
  assertIsIterable(obj);
  assertIsFunction(fn);
  for (const element of obj) {
    fn(element);
  }
}
```

```
const visited = [];
starIt.forEach(arr, v => visited.push(v));
expect(visited).toEqual(arr);
```

includes

```
function includes(obj, value) {
  assertIsIterable(obj);
  for (const element of obj) {
    if (element === value) return true;
  }
  return false;
}
```

```
expect(starIt.includes(arr, 5)).toBeTruthy();
expect(starIt.includes(arr, 4)).toBeFalsy();
```

indexOf

```
function indexOf(obj, value) {
  assertIsIterable(obj);
  let index = 0;
  for (const element of obj) {
    if (element === value) return index;
    index++;
  }
  return -1;
}
```

```
expect(starIt.indexOf(arr, 3)).toBe(1);
expect(starIt.indexOf(arr, 4)).toBe(-1);
```

lastIndexOf

```
function lastIndexOf(obj, value) {
  assertIsIterable(obj);
  let index = 0, lastIndex = -1;
  for (const element of obj) {
    if (element === value) lastIndex = index;
    index++;
  }
  return lastIndex;
}
```

```
expect(starIt.lastIndexOf(arr, 3)).toBe(5);
expect(starIt.lastIndexOf(arr, 4)).toBe(-1);
```

map

```
function* map(obj, fn) {
  assertIsIterable(obj);
  assertIsFunction(fn);
  for (const element of obj) {
    yield fn(element);
  }
}
```

```
let iterable = starIt.map(arr, isOdd);
let result = [...iterable];
expect(result).toEqual([
  true, true, true, false,
  true, true, true
]);

iterable = starIt.map([], isOdd);
result = [...iterable];
expect(result).toEqual([]);
```

reduce

```
function reduce(obj, fn, initial) {
  assertIsIterable(obj);
  assertIsFunction(fn);
  const it = obj[Symbol.iterator]();

  let done = false, value;
  if (initial === undefined) {
```

```
      ({value, done} = it.next());
    } else {
      value = initial;
    }

    let result = value;
    while (!done) {
      ({value, done} = it.next());
      if (!done) result = fn(result, value);
    }

    return result;
  }
```

```
expect(starIt.reduce(arr, add)).toBe(26);
expect(starIt.reduce([19], add)).toBe(19);
expect(starIt.reduce([], add, 0)).toBe(0);
```

some

```
function some(obj, predicate) {
  assertIsIterable(obj);
  assertIsFunction(predicate);
  for (const element of obj) {
    if (predicate(element)) return true;
  }
  return false;
}
```

```
expect(starIt.some(arr, isOdd)).toBeTruthy();
```

Here are some bonus functions that are not in the `Array` class but are useful when working with iterables.

skip

```
// This skips the first n values of an iterable
// and yields the rest.
function* skip(obj, n) {
  assertIsIterable(obj);
  const iterator = obj[Symbol.iterator]();
  let result;

  // Skip the first n values.
  for (let i = 0; i <= n; i++) {
    result = iterator.next();
    if (result.done) return;
  }

  // Yield the rest of the values.
  while (!result.done) {
    yield result.value;
    result = iterator.next();
  }
}
```

```
const gen = starIt.skip(arr, 2);
expect(gen.next().value).toBe(5);
expect(gen.next().value).toBe(6);
```

take

```
// Yields only the first n values of an iterable.
function* take(obj, n) {
  assertIsIterable(obj);
  const iterator = obj[Symbol.iterator]();
  while (n > 0) {
    yield iterator.next().value;
    n--;
  }
}
```

```
const gen = starIt.take(arr, 2);
expect(gen.next().value).toBe(1);
expect(gen.next().value).toBe(3);
expect(gen.next().value).toBe(undefined);
```

Summary

JavaScript iterators are cool! JavaScript generators are even cooler! Understanding these is important to fully utilize for-of loops, the spread operator, and destructuring. As seen in the `TreeNode` example class, it is sometimes useful to write classes in such a way that objects created from them are iterable.

References

- ECMAScript specification - http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts (see "Final Draft" section)
- Luke Hoban's ES6 introduction - https://github.com/lukehoban/es6features
- Iterables and iterators in ECMAScript 6 - http://www.2ality.com/2015/02/es6-iteration.html
- ES6 generators in depth - http://www.2ality.com/2015/03/es6-generators.html
- star-it library - https://github.com/mvolkmann/star-it
- my slides on ES 2015 and many other topics - http://ociweb.com/mark
- video of a talk I gave on ES 2015 - https://www.youtube.com/watch?v=13kawyfG_mc&feature=youtu.be