Notes - JavaScript - Iterator and Iterable

- Dr Nick Hayward

Notes on options and usage for *iterators* and *iterables* in JavaScript.

Contents

Intro Iteration - a new option in ES6 for *traversing* data.

Two parts to traversing data,

- *iterable* - a data structure to provide iterable access to the public, achieved with a method `Symbol.iterator` . This is a factory for *iterators*.
- *iterator* - a pointer for traversing elements in a data structure (n.b. similar to cursor in DB)

So, for a custom function, we may return `an iterable object &c. with an iterator` (return an iterator for an iterable…).

Built-in iteration constructs Standard language constructs that access iteration include,

- destructuring via an array pattern

```
// destructuring via an Array pattern
const [a,b] = new Set(['hello', 'world', '!']);
// returns array of values from Set...
console.log([a,b]); // outputs ['hello', 'world']
```

- for-of loop

```
// for-of loop usage
for (const x of ['hello', 'world']) {
  // returns each array value...
  console.log(x);
}
```

- Array.from()

```
// EX3: Array.from
const arr = Array.from(new Set(['hello', 'world']))
// returns standard array - iterable as usual
console.log(arr[1]);
```

- Spread operator `(...)`

```
// EX4: Spread operator
const arrSpread = [...new Set(['hello', 'world'])];
// takes dynamic no. of values and returns array...
console.log(arrSpread);
```

- constructors of Map and Set

```
// EX5: Map constructor - standard key/value pairings...
const testMap = new Map([[false, '0'], [true, '1']]);
// maps false to 0 &c.
console.log(testMap);
// use standard Map methods - e.g. get and set
console.log(testMap.get(false));
// use iterator methods
console.log(testMap.keys());
console.log(testMap.values());
// then iterate over returns from iterator method
console.log(Array.from(testMap.keys())); // returns expected array containing map values...
```

and we can do the same with `Set`

```
// Set constructor
const testSet = new Set(['hello', 'world']);
```

We also have default iterables provided by *Promise* methods such as `Promise.all` and `Promise.race`, and `yield*` for generator iterables.

The nature of iterable The nature of iterability may be defined as follows,

- data consumers - JS has various language constructs, which consume data, e.g.
  - `for-of` loops over values
  - spread `(...)` operator inserts values into Arrays or function calls
  - ...
- data sources - consume values from a variety of data sources, including iterating element of an array, key/value entries in a Map or simply the characters in a string

So, ES6 introduces an interface pattern for `Iterable` - **data consumers use it, data sources implement it...**

n.b. JS does not have interfaces, `Iterable` is more akin to a convention for general usage.

source A value is `iterable` if it provides the symbol `Symbol.iterator` - this returns the so-called iterator.

This iterator is an object that returns values via its method, `next()` - this iterates one method call at a time...

consumption We may consume the data using the iterator to retrieve values...

e.g.

```
// create array
const testArr = ['hello', 'world'];
// create iterator for array object - works due to Array default iterator...
const testIterator = testArr[Symbol.iterator]();
```

Then, we may call the `next()` method for the iterator we just created,

```
// call next on iterator
testIterator.next();
// returns value and done boolean...
{value: 'hello', done: false}
// call next on iterator
testIterator.next();
// returns value and done boolean...
{value: 'world', done: false}
// call next on iterator
testIterator.next();
// returns value and done boolean...
{value: undefined, done: true}
```

Same pattern as ES6 generator - `next()` is called one extra time to return `true` boolean for `done` .

The protocol for iterable and iterators defines the iteration as sequential - iterator will return one value at a time…

Iterable values Iterable values include,

- Arrays

Default Array object is iterable for each element. We may use a standard loop construct, e.g.

```
for (const val of ['hello', 'world']) {
  console.log(val);
}

//returns expected values
$ hello
$ world
```

- Strings

String data type is iterable - shows that one of the primitive values is iterable. This is achieved by *coercing* all values to objects before the iterator method is accessed.

```
for (const char of 'hello') {
  console.log(char);
}
```

- Maps

Iterable over their entries - entry encoded as key/value pair, an Array with two elements.

Entries are always iterated over in the same order they were added to the map.

n.b. Map keys must be unique…

```
 const testMap = new Map();
 testMap.set('student1', {quiz1: 'B', quiz2: 'B-'});
 testMap.set('student2', {quiz1: 'B-'});
 // iterate over map
 for (const [key,value] of testMap) {
   console.log(`key = ${key} & value = ${value}`);
 }
```

- Sets

Iterable over their elements - iterated over in the same order as they were added.

```
const testSet = new Set();
testSet.add('B');
testSet.add('B-');
// iterate over set
for (const val of testSet) {
  console.log(val);
}
```

Similar in design and usage to Maps, but a Set does not include a key/value pair, only values.

n.b. values must be unique…

- DOM data structures

In JS, most DOM data structure will eventually by fully iterable. This functionality is currently a work in progress following ES6, e.g.

```
for (const node of document.querySelectorAll('p')) {
  ...
}
```

- iterable computed data

Iterable content may also be consumed from data computed dynamically.

All major ES6 data structures, e.g. Arrays, Typed Arrays, Maps, & sets, have three built-in methods, which return iterable objects.

```
const testArr = ['hello', 'world'];
for (const pair of testArr.entries()) {
  console.log(pair);
  // outputs
  // [ 0, 'hello' ]
  // [ 1, 'world' ]
}
```

n.b. plain objects are not iterable by default…

iterate over properties We may create a custom *helper* function to iterate over properties, e.g.

```
/*
* HELPER FN:
* - iterate object properties v.1
* - use iterator to traverse the Array with the property keys
*/
function objectEntries(data) {
    /*
    * `Reflect` is a built-in object - provides methods interceptible JS operations
    * - not a constructor (unlike most global objects)
    * - all properties and methods are static (like the `Math` object)
    */
    // intercept current parameter for properties - wrap keys with iterator
    let iter = Reflect.ownKeys(data)[Symbol.iterator]();

    console.log(Reflect.ownKeys(data)); //logs `student1`, `student2`...

    // for function call - return local scope `this` wrapped in its own iterator...
    return {
        [Symbol.iterator]() {
            return this;
        },
        next() {
            // destructure return from keys iterator - returns `student1`, `student2`...
            let { done, value: key } = iter.next();
            console.log({ done, value: key });
            // if no more keys - return end of outer iterator...
            if (done) {
                return { done: true };
            }
            // otherwise - return the key & the value for the key in the passed data...
            return { value: [key, data[key]] };
        }
    };
}
```

Iteration language constructs Iteration protocol is supported by default in the following ES6 language constructs,

- destructuring via Array pattern (works for any iterable)

```
// EX1.1: destructuring via Array pattern
const testArray = ['book1', 'book2', 'book3'];
const [a,b] = testArray;
console.log([a,b]); // logs: ['book1', 'book2']

// EX1.2: destructuring via Array pattern
const testSet = new Set();
testSet.add('book1');
testSet.add('book2');
testSet.add('book3');
const [c,d] = testSet;
console.log([c,d]); // logs: ['book1', 'book2']
const [e,...f] = testSet;
console.log([e, ...f]); // logs: ['book1', 'book2', 'book3'] due to spread operator for 'f'
```

- `for-of` loop

```
// EX2: for-of loop
for (const val of testSet) {
  console.log(`set value = ${val}`); // logs: 'set value = book1' &c.
}
```

- `Array.from()` - converts iterables and array-like values (e.g. basic object) to Arrays (also available for typed arrays)

```
// EX3: Array.from() - converts iterables to Arrays
const testMap = new Map();
testMap.set('student1', {quiz1: 'B', quiz2: 'B-'});
testMap.set('student2', {quiz1: 'B-'});
/*
* log Array.from
* [ [ 'student1', { quiz1: 'B', quiz2: 'B-' } ],
  [ 'student2', { quiz1: 'B-' } ] ]
*/
const logArrayFrom = Array.from(testMap)
console.log(logArrayFrom[1][1]); // logs {quiz1: 'B-'}
```

- spread operator - inserts values of an iterable into an Array

```
// EX4: spread operator - inserts values of an iterable into an Array
const testArray2 = ['book3', 'book4'];
const spreadArray = ['book1', 'book2', ...testArray2, 'book5'];
console.log(spreadArray); // logs: ['book1', 'book2', 'book3', 'book4', 'book5']
```

- Maps - constructor iterates over [key,value] pairs into a Map

```
const getStudent = testMap.get('student1');
console.log(getStudent); // logs: {quiz1: 'B', quiz2: 'B-'}
```

- Sets - constructor iterates over elements into a Set

```
// EX6: Sets - constructor iterates over elements into a Set
const hasBook1 = testSet.has('book1');
const hasBook4 = testSet.has('book4');
// logs: 'testSet has book1: true and book2: false'
console.log(`testSet has book1: ${hasBook1} and book4: ${hasBook4}`);
```

- Promises - `Promise.all` and `Promise.race`

```
Promise
  .all([
    fetch('./assets/items.json'),
    fetch('./assets/notes.json')
  ])
  .then(responses =>
    Promise.all(responses.map(res => res.json()))
  ).then (json => {
    console.log(json);
  });
```

```
Promise
  .race([
    fetch('./assets/items.json'),
    fetch('./assets/notes.json')
  ])
  .then(responses => {
    return responses.json()
  })
  .then(res => console.log(res));
```

- `yield*` - operator available only inside a generator, yields all items iterated over by an iterable

```
/*
* EX8: `yield*`
* - operator available only inside a generator,
*   yields all items iterated over by an iterable
*/
function* yieldAllGeneratorValues(iterable) {
  yield* iterable;
}

console.log(yieldAllGeneratorValues(testArray));
```

Implementing iterables Manual implementation of iterators, compared with built-in ES6 examples such as *generators*.

Iteration protocol may be considered as follows,

`Iterable: traversable data structure | Iterator: pointer for traversing an iterable`

`[Symbol.iterator] ------------- RETURNS ------------> next()`

Object becomes *iterable* if it has a method (own or inherited) whose key is `Symbol.iterator` . This method must return an iterator.

This iterator object will iterate over the items *inside* the iterator via its method `next()` .

A basic, dummy iterable is as follows,

```
// EX1: basic custom iterable - returns hard-coded...
const iterable = {
  [Symbol.iterator]() {
    let step = 0;
    const iterator = {
      next() {
        if (step <= 2) {
          step++;
        }
        switch(step) {
          case 1:
            return {value: 'dance', done: false};
          case 2:
            return {value: 'sing', done: false};
          default:
            return {value: undefined, done: true};
        }
      }
    };
    return iterator;
  }
};

for (const val of iterable) {
  console.log(val);
}
```

In the `return`, `value` holds the actual item's value, and `done` specifies a boolean for the end of the iterator.

A custom implementation with passed parameters will be more common. The iterator will be added to the passed parameters thereby creating an iterable object.

```
// EX1: params are NOT iterable by default - need to create iterable over passed params...
function iterateOver(...data) {
  // define counter for iterator
  let index = 0;
  // define iterable wrapper for passed data params...
  const iterable = {
    [Symbol.iterator]() {
      // define iterator for looping over data
      const iterator = {
        next() {

        }
      };
      return iterator;
    }
  };
  return iterable;
}
```

This will then be called using one of the available output options for an iterable object,

```
for (const val of iteratorOver('do', 're', 'mi', 'fa', 'so', 'la', 'te')) {
  console.log(val);
}
```

closing iterators The optional iterator method, `return()` , allows an iterator to clean up if it wasn't iterated over to the expected end.

This method will also *close* the iterator.

In `for-of` , we may also cause premature termination using one of the following,

- break
- continue (e.g. continue an outer loop from an inner loop - inner loop will `break`)
- throw (catch errors &c.)
- return

Likewise, many built-in iterators will close if they are not completely *drained*,

- `for-of`
- `yield*`
- Destructuring
- `Array.from()`
- `Map()` , `Set()` , `WeakMap()` , `WeakSet()`
- `Promise.all()` , `Promise.race()`

Helper functions Helper functions may also be developed to return iterables, in addition to more standard iterable data structures.

A common requirement is an iterable for the properties of an object,

```
function objectEntries(obj) {
  let iter = Reflect.ownKeys(obj)[Symbol.iterator]();

  return {
    [Symbol.iterator]() {
      return this;
    },
    next() {
      let {done, value:key} = iter.next();
      if (done) {
        // if done is true, explicit undefined value is not necessary
        return { done: true };
      }
      return {value: [key, obj[key]]};
    }
  };
}
```

Example usage is as follows,

```
const obj = {book1: 'the glass bead game', book2: 'hannibal\'s footsteps',
                          book3: 'a good year'};

for (const [key, value] of objectEntries(obj)) {
```

```
  console.log(`${key}: ${value}`);
}
```

However, we might want to output property values for nested objects, e.g.

```
const obj = {
  library: 'waldzell',
  archive: {
    book1: 'the glass bead game',
    book2: 'hannibal\'s footsteps',
    book3: 'a good year'
  }
  details: {
    location: 'mariafels',
    director: 'knoefels',
    access: 'restricted'
  }
}
```

Iterables and recursion We might also create a factory function for creating iterable objects. We may wrap the initial parent object in an iterable with iterator, and then check for a child object as the iterator loops over parent object.

```
/******
* FNs: test iterator functions
*/
// FN: return iterable with iterator for plain object
function objectIterable(obj) {

  // create iterable object from passed target param - define function for iterator
  obj[Symbol.iterator] = function () {
    // define keys from passed object
    const keys = Object.keys(obj);
    return {
      // define iterator next() method
      next() {
        // done boolean set to true with no more keys from passed object
        const done = keys.length === 0;
        // set key to first element - remove element
        const key = keys.shift();
        // return object - done boolean and value array from each item in passed object
        return {
          done,
          value: [key, obj[key]]
        }
      }
    };
  }
  // return new iterable object for passed param object
  return obj;

}
```

This function becomes the initial factory for creating an iterable object. Regardless of depth, we may create an iterable object using a recursive function, e.g.

```
// FN: recursive function for creating object iterables...
// pass plain object - iterable with iterator returned
function createObjIterator(obj) {

  // create iterable object with iterator
  const iter = objectIterable(obj);

  // iterate over object iterable - check key & val
  for ([key, val] of iter) {
    // check child property value - if object, invoke recursion...
    if (typeof val === 'object') {
        // call self to invoke recursion...
        createObjIterator(val);
    } else {
      //console.log(`${key} - ${val}`);
      // add key/val pair to array...
      archiveArr.push([key, val]);
    }
  }

}
```

Then, we may simply test as follows,

```
createObjIterator(archive);

//console.log(objMap);
console.log(archiveArr);
```

References

- MDN - JS - Symbol
- MDN - JS - Symbol.iterator