JavaScript Iterables, Iterators, and Generators

Iterators

- Objects that have a **`next`** method
- Used to visit elements in a sequence
  - even in a lazy manner
- Returns an object with **`value`** and **`done`** properties
  - it's best to return a new object from each call
- If end of sequence has been reached, **`done`** will be true
  - can omit otherwise
  - for infinite sequences, **`done`** never becomes true
- Whether **`value`** has meaning when **`done`** is **`true`** depends on the iterator
  - but the for-of loop, spread operator, and destructuring will ignore this value
  - can omit **`value`** property

Iterables

- Objects that have a method whose name is the value of `Symbol.iterator`
- That method returns an iterator
- An object can be both an iterable and an iterator

  `obj[Symbol.iterator]() === obj`
  and `obj` has a `next` method

Iterable/Iterator Example

```
let fibonacci = {
  [Symbol.iterator]() {
    let prev = 0, curr = 1;
    return {
      next() {
        [prev, curr] = [curr, prev + curr];
        return {value: curr};
      }
    };
  }
};

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

stops iterating when **done** is true which never happens here

```
1
2
3
5
8
13
21
34
55
89
```

skipping initial values of 0 and 1 and starting at the second 1

Iterable Objects ...

- Objects from these builtin classes are iterable
  - `Array` - over elements
  - `Set` - over elements
  - `Map` - over key/value pairs as `[key, value]`
  - DOM `NodeList` - over `Node` objects (when browsers add support)
- Primitive strings are iterable
  - over Unicode code points

... Iterable Objects ...

- 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

for arrays, keys are indices;
for sets, keys are same as values

objects returned are both iterators and iterable

- Custom objects can be made iterable
  - by adding `Symbol.iterator` method

… Iterable Objects

- **Ordinary objects** such as those created from object literals are **not iterable**
  - when this is desired, use `Map` class instead **or** write a function like the following

this serves as an example of how to implement an iterator

to exclude symbol keys, use `Object.getOwnPropertyNames(obj)`

```
function objectEntries(obj) {
  let index = 0;
  let keys = Reflect.ownKeys(obj); // gets both string and symbol keys
  return { // the iterable and iterator can be same object
    [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);
}
```

```
// Using a generator
function* objectEntries(obj) {
  let keys = Reflect.ownKeys(obj);
  for (const key of keys) yield([key, obj[key]]);
}
```

can get an iterable for keys in an object with `Reflect.enumerate(obj);`

Iterable Consumers ...

- **`for-of` loop**
  - `for (const value of someIterable) { ... } // iterates over all values`
- spread operator
  - can add all values from iterable into a new array
    - `let arr = [firstElem, ...someIterable, lastElem];`
  - can use all values from iterable as arguments to a function, method, or constructor call
    - `someFunction(firstArg, ...someIterable, lastArg);`
- positional destructuring
  - `let [a, b, c] = someIterable; // gets first three values`

… Iterable Consumers

- `Set` constructor takes an iterable over values
- `Map` constructor takes an iterable over key/value pairs

- `Promise` methods `all` and `race` take an iterable over promises

- In a generator, `yield*` yields all values in an iterable one at a time
  - will make sense after generators are explained

Generators

- Special kind of iterator that is also iterable
- Can be paused and resumed via multiple return points, each specified using **`yield`** keyword
  - **`yield`** keyword can only be used in generator functions
  - each **`yield`** is hit in a separate call to **`next`** method
  - to yield a single value, **`yield`** ***`value;`***
  - to yield each value returned by an iterable one at a time, **`yield*`** ***`iterable;`***
    - can obtain an iterable by calling another generator function (see next slide)
- Exit by
  - running off end of function
  - returning a specific value using **`return`** keyword
  - throwing an error

`done` will be **`true`** after any of these and will remain **`true`**

Generator Functions

- Return a generator
- Defined with "`function*`" instead of "`function`"
- Can define "generator methods" in class definitions
  - precede method name with `*`

Generator Methods

called on a generator object returned by a generator function

typically these methods are not used directly

- **`next(value)`** method
  - gets next value, similar to iterator **`next`** method
  - takes optional argument, but not on first call
    - specifies value that the **`yield`** hit in this call will return at start of processing for next call
- **`return(value)`** method
  - terminates generator from the outside just as if the generator returned the specified value
  - returns **`{value: value; done: true}`**
- **`throw(error)`** method
  - throws error inside generator at **`yield`** where execution paused
  - if generator catches error and yields a value, generator is not terminated yet
  - otherwise generator is terminated

Basic Generator

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

let myGen = myGenFn(); // 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()) {
  console.log(n); // 1, then 2, not 3
}
```

without **return** statement in **myGenFn**, this disappears

Infinite Generator

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

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

```
1
2
3
5
8
13
21
34
55
89
```

```
// Iterables can be
// implemented with generators.
let fib = {
  * [Symbol.iterator]() {
    let [prev, curr] = [0, 1];
    while (true) {
      [prev, curr] = [curr, prev + curr];
      yield curr;
    }
  }
};

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

`Array` Methods

- It would be nice if many `Array` methods could be used on any iterable
- `includes` - determines whether a collection contains a given value
- `indexOf` - finds index of first occurrence of a given value
- `lastIndexOf` - finds index of last occurrence of a given value
- `find` - finds first element that meets some condition
- `findIndex` - finds 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 new collection of elements that meet a condition
- `map` - generates 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 final result of applying a given function to previous result and next element

star-it

- 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
- Only `filter` and `map` make sense for infinite sequences
- This code provides good examples of
  working with iterables and generators!
- At https://github.com/mvolkmann/star-it

Test Setup

```
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);
  }
  // Traverses all descendants of this TreeNode
  // deep-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; // 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;
      }
    }
  }
}
```

Runtime Assertions

```
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.call(value);
  assertIsIterator(iterator);
}
```

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();
```

infinite sequence warning:
won't return if all values satisfy predicate

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);
```

infinite sequence warning:
won't return if no value satisfies predicate

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);
```

infinite sequence warning:
won't return if no value satisfies predicate

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);
```

infinite sequence warning:
won't return

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();
```

infinite sequence warning:
won't return if
value is not found

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);
```

infinite sequence warning:
won't return if
value is not found

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);
```

infinite sequence warning:
won't return

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);
```

infinite sequence warning:
won't return

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();
```

infinite sequence warning:
won't return if no value satisfies predicate

skip

```
// 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 in order to fully utilize for-of loops, the spread operator, and destructuring