Modern Asynchronous JavaScript

Chapter 2: Enhancing Custom Iterators With Generators

Overview of today's presentation

Jack:

- What is a generator
- What is the purpose of a generator
- The iteration protocols

Mandy:

- Difference between a *synchronous* iterator and a *synchronous* generator
- Difference between an *asynchronous* iterator and an *asynchronous* generator

Giuseppe:

- Differences between a custom iterator and generator
- How generators make your life easier

Questions & Quiz

What is a Generator?

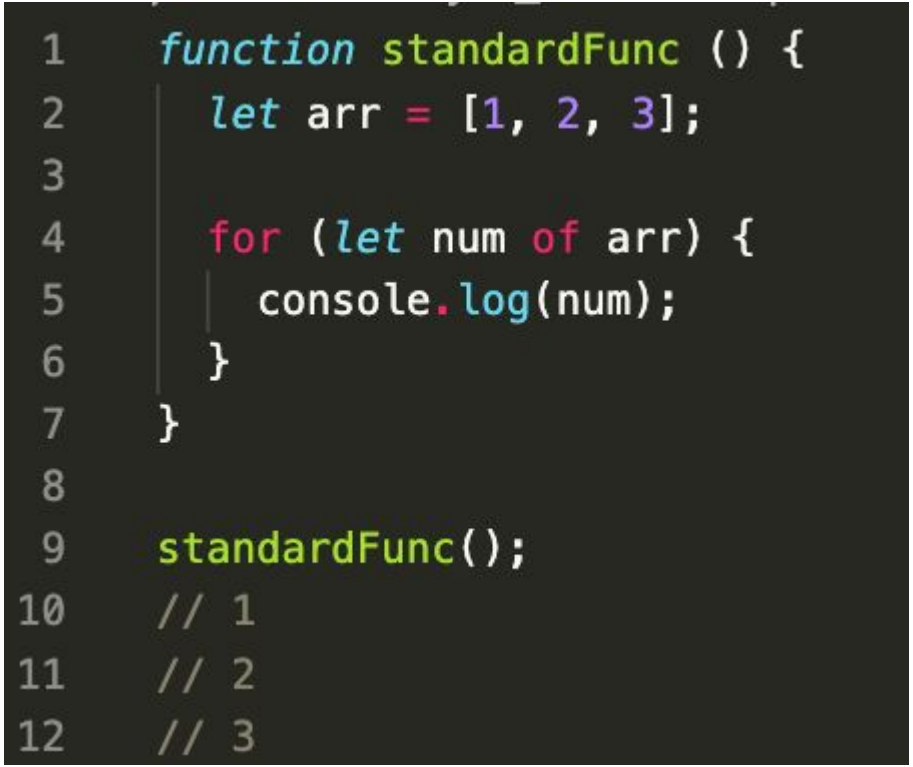
A generator function allows us to define an iterative algorithm by writing a single function **whose execution is not continuous**.

They also return a generator object, which is iterable.

```
1  function* generator () {
2    let arr = [1, 2, 3];
3
4    for (let num of arr) {
5      yield num;
6    }
7  }
8
9  let iterator = generator();
10 console.log(iterator); // Object [Generator] {}
```

Continuous execution

Normally, we expect JavaScript functions to have continuous execution.



```
1   function standardFunc () {
2     let arr = [1, 2, 3];
3
4     for (let num of arr) {
5       console.log(num);
6     }
7   }
8
9   standardFunc();
10  // 1
11  // 2
12  // 3
```

Non-continuous execution

Generator functions however, allow for non-continuous execution.

```
1   function* generatorFunc() {
2     let arr = [1, 2, 3];
3
4     for (let num of arr) {
5       yield num;
6     }
7   }
8
9   let iterator = generatorFunc();
10  console.log(iterator.next()); // { value: 1, done: false }
11  console.log(iterator.next()); // { value: 2, done: false }
12  console.log(iterator.next()); // { value: 3, done: false }
13  console.log(iterator.next()); // { value: undefined, done: true }
```

```
1  function* generator() {
2    yield 1;
3    yield 2;
4  }
5
6  let iterator = generator();
7  iterator.next(); // { value: 1, done: false }
8  iterator.next(); // { value: 2, done: false }
9  iterator.next(); // { value: undefined, done: true }
```

Line 6: `generatorFunc()` (returns generator object)

Line 7: `iterator.next()` (encounters `yield`. Yields value and saves state)

Line 8: `iterator.next()` (encounters `yield`. Yields value and saves state)

Line 9 `iterator.next()` (calling `next` has no effect as there are no more values to yield. Repeated calls will return the same result)

Other features & things to be aware of

```
1  function* generatorFunc() {
2    let name = yield 'What is your name?';
3    yield `Hello ${name}`;
4    return 'My name is Hal';
5    yield "I'm sorry Pete, I'm afraid I can't do that"; // unreachable
6  }
7
8  let iterator = generatorFunc();
9  console.log(iterator.next()); // { value: 'What is your name?', done: false }
10 console.log(iterator.next('Pete')); // { value: 'Hello Pete', done: false }
11 console.log(iterator.next()); // { value: 'My name is Hal', done: true }
12 console.log(iterator.next()); // { value: undefined, done: true }
```

- The argument we pass in becomes the value of `yield`
- A `return` statement will pass its value and set `done` to `true`. Any code after the return is unreachable

Iteration Protocols

1. The Iterator Protocol
2. The Iterable Protocol

Iterator Protocol

An object is said to conform to the iterator protocol when it implements some interface that can determine:

1. If there are any elements left, and
2. What the next element is

To satisfy this, the `next()` method returns an object containing the properties `value` and `done`

Iterable Protocol

An object is said to conform to the iterable protocol, when it contains a `[Symbol.iterator]` property, which returns an object that conforms to the iterator protocol

Review from Chapter 1
Example 1: Synchronous Custom Iterator

```
const collection = {
  a: 10,
  b: 20,
  c: 30,

  [Symbol.iterator]() {

    let i = 0;
    const values = Object.keys(this);

    return {

      next: () => {
        return {
          value: this[values[i++]],
          done: i > values.length
        }
      }

    }

  }

};
```

As an iterable, the object must have a **Symbol.iterator** method. The **Symbol.iterator** method returns a plain object that contains a **next** property.

The **next()** method returns the iteration result of the object. The method **next()** returns an object with two properties: **value** and **done**.

The **value** property holds the value returned by the iterator. The **done** property holds **true** when there is no value to return and the iteration is complete.

New material from Chapter 2
Example 1: Synchronous Generator

```
1  const collection = {
2    a: 10,
3    b: 20,
4    c: 30,
5
6    [Symbol.iterator]: function* () {
7
8      for (let key in this) {
9        yield this[key];
10     }
11
12   }
13
14 };
```

The object also has an **Symbol.iterator** method. The method is defined with an **asterisk (*)** to indicating it's a generator function.

There is a **for...in loop** inside the generator function to iterate over the collection object's properties.

With each iteration, the **yield** keyword halts the loop's execution and returns the **value** to the caller.

Synchronous Iterator vs. Synchronous Generator

```
const collection = {
  a: 10,
  b: 20,
  c: 30,

  [Symbol.iterator]() {

    let i = 0;
    const values = Object.keys(this);

    return {
      next: () => {
        return {
          value: this[values[i++]],
          done: i > values.length
        }
      }
    }
  }
};
```

```
const collection = {
  a: 10,
  b: 20,
  c: 30,

  [Symbol.iterator]: function* () {

    for (let key in this) {
      yield this[key];
    }

  }
};
```

```
const iterator = collection[Symbol.iterator]();

console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());

// OUTPUT:
{ value: 10, done: false }
{ value: 20, done: false }
{ value: 30, done: false }
{ value: undefined, done: true }
```

Review from Chapter 1
Example 2: Custom Asynchronous Iterator

```
1  const srcArr = [
2    'https://eloux.com/async_js/examples/1.json',
3    'https://eloux.com/async_js/examples/2.json',
4    'https://eloux.com/async_js/examples/3.json',
5  ]
6
7
8  srcArr[Symbol.asyncIterator] = function() {
9    let i = 0;
10
11   return {
12
13     async next() {
14       if (i === srcArr.length) {
15         return {
16           done: true
17         };
18       }
19
20       const url = srcArr[i++];
21       const response = await fetch(url);
22
23       if (!response.ok) {
24         throw new Error ('Unable to retrieve URL: ' + url);
25       }
26
27       return {
28         value: await response.json(),
29         done: false
30       }
31
32     }
33
34   }
35 }
```

Assign a function to the **Symbol.asyncIterator** property of `srcArr` array.

Symbol.asyncIterator method returns an object that contains an **async next()** method.

The **async next()** method fetches data asynchronous from the URL provided by `srcArr` array.

On lines **15 - 16**, **async next()** returns a Promise when the end of the array is reached. Equivalent to ***Promise.resolve({ done: true })***

Otherwise on lines **27 - 30**, **async next()** returns a Promise which resolves to an object with the properties **value** and **done.** The **value** property references the data fetched and **done** references **false**.

The **async next()** always returns a promise because it's a **async** function.

New material from Chapter 2
Example 2: Asynchronous Generator

```
1  const srcArr = [
2   'https://eloux.com/async_js/examples/1.json',
3   'https://eloux.com/async_js/examples/2.json',
4   'https://eloux.com/async_js/examples/3.json',
5  ]
6
7  srcArr[Symbol.asyncIterator] = async function*() {
8    let i = 0;
9
10   for (const url of this) {
11     const response = await fetch(url);
12
13     if (!response.ok) {
14       throw new Error('Unable to retrieve URL: ' + response.status);
15     }
16
17     yield response.json();
18
19   }
20
21 };
```

Assign a function to the **Symbol.asyncIterator** property of `srcArr` array.

The method is defined with an **asterisk (*)** to indicate it's a generator function.

Lines **10 - 19** there is a **for...of loop** to fetch the data from each url asynchronously using **await**.

We do not need to define an **async next()** method. The **yield keyword** returns the value to the function's caller. When **next()** is called, a promise is returned.

Line 17 is equivalent to yield **await** response.json() since yield calls **await** implicitly.

Asynchronous Iterator vs. Asynchronous Generator

```
const srcArr = [
  'https://eloux.com/async_js/examples/1.json',
  'https://eloux.com/async_js/examples/2.json',
  'https://eloux.com/async_js/examples/3.json',
]

srcArr[Symbol.asyncIterator] = function() {
  let i = 0;

  return {

    async next() {
      if (i === srcArr.length) {
        return {
          done: true
        };
      }

      const url = srcArr[i++];
      const response = await fetch(url);

      if (!response.ok) {
        throw new Error ('Unable to retrieve URL: ' + url);
      }

      return {
        value: await response.json(),
        done: false
      }
    }
  };
};
```

```
const srcArr = [
  'https://eloux.com/async_js/examples/1.json',
  'https://eloux.com/async_js/examples/2.json',
  'https://eloux.com/async_js/examples/3.json',
]

srcArr[Symbol.asyncIterator] = async function*() {
  let i = 0;

  for (const url of this) {
    const response = await fetch(url);

    if (!response.ok) {
      throw new Error('Unable to retrieve URL: ' + response.status);
    }

    yield response.json();
  }
};
```

```
const iterator = srcArr[Symbol.asyncIterator]();

iterator.next().then(result => {
  console.log(result.value.firstName);
});

iterator.next().then(result => {
  console.log(result.value.firstName);
});

iterator.next().then(result => {
  console.log(result.value.firstName);
});

// OUTPUT:
John
Anna
Peter
```

Real world example of Asynchronous generator

Using an Async generator to iterate over paginated data.
Paginated data is data in small chunks rather than delivered all at once. For example, the forum posts on launch school is paginated data. There's roughly 10 post on a page, and you need to click "next" to load more forum posts.

In the textbook example, "**gen_ex03.js**" on page 17.
A generator is used to make multiple network requests to the GitHub API and retrieve commits for a repository. The API sends a response for the last 30 commits, but in order to retrieve 90, we can use a generator and on each iteration we fetch the next batch of commits.

A Generator Iterable has itself as Iterator

```
const aGeneratorObject = function* () {
 yield 1;
 yield 2;
 yield 3;
}();

console.log(typeof aGeneratorObject.next);
// "function", because it has a next method, so it's an iterator

console.log(typeof aGeneratorObject[Symbol.iterator]);
// "function", because it has an @@iterator method, so it's an iterable

console.log(aGeneratorObject[Symbol.iterator]() === aGeneratorObject);
// true, because its @@iterator method returns itself (an iterator), so it's an
well-formed iterable
```

An Iterable that has itself as Iterator

```
function objectEntries(obj) {
 let index = 0;
 let propKeys = Reflect.ownKeys(obj);

 return {
   [Symbol.iterator]() {
     return this;
   },
   next() {
     if (index < propKeys.length) {
       let key = propKeys[index];
       index++;
       return { value: [key, obj[key]] };
     } else {
       return { done: true };
     }
   }
 };
}
```

```
let obj = {};
const iterable = objectEntries(obj);
console.log(iterable[Symbol.iterator]() === iterable);
// true
```

Iterables which can iterate only once VS Iterables which can iterate many times

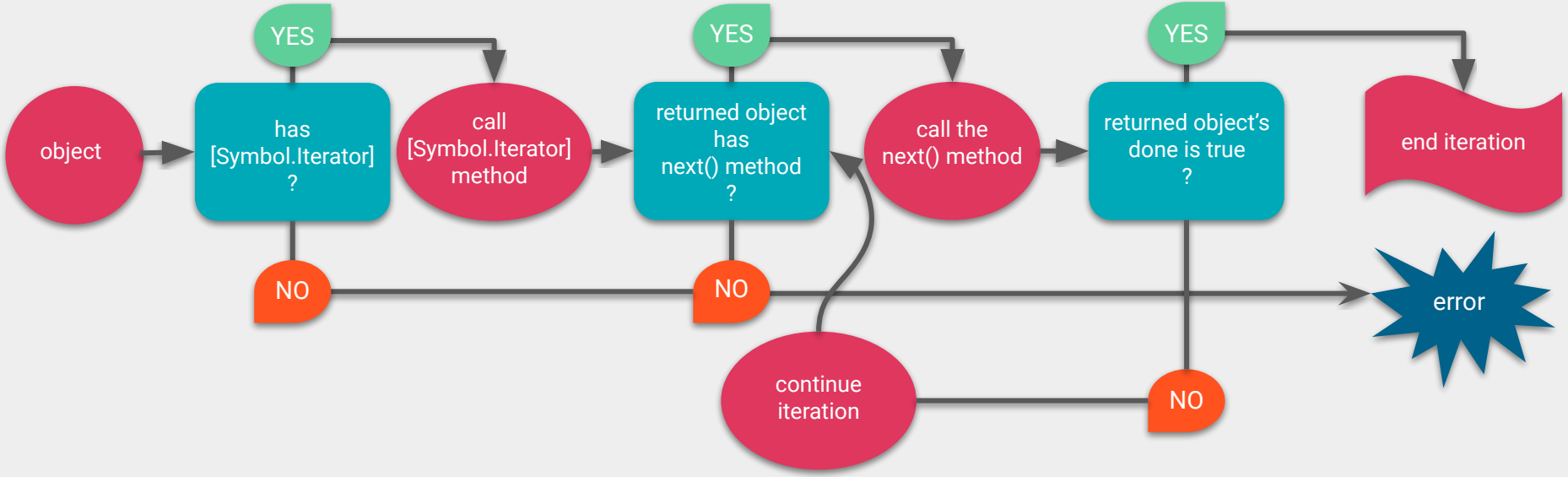
```
function* elements() {
   yield 'a';
   yield 'b';
}

function iterateTwice(iterable) {
   for (const x of iterable) {
       console.log(x);
   }
   for (const x of iterable) {
       console.log(x);
   }
}
```

```
iterateTwice(['a', 'b']);
// Output:
// a
// b
// a
// b

iterateTwice(elements());
// Output:
// a
// b
```

From MDN

Iterables which can iterate only once (such as Generators) customarily return this from their @@iterator method, whereas iterables which can be iterated many times must return a new iterator on each invocation of @@iterator.

Iteration inside a for...of statement



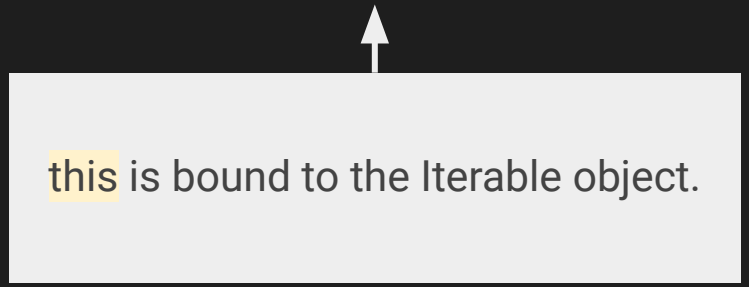
Whether an iterable produces a new iterator or not, matter when you iterate over the same iterable multiple times.

```
function getIterator(iterable) {
   return iterable[Symbol.iterator]();
}
let iterable = ['a', 'b'];
console.log(getIterator(iterable) === getIterator(iterable)); // false

function* elements() {
   yield 'a';
   yield 'b';
}
iterable = elements();
console.log(getIterator(iterable) === getIterator(iterable)); // true
```

What about *this*?

```
const collection = {
 a: 10,
 b: 20,
 c: 30,
 [Symbol.iterator]: function*() {
   for (let key in this) {
     // yield this[key];
     yield this;
   }
 }
};

const iterator = collection[Symbol.iterator]();
console.log(iterator.next());
```

```
{
  value: {
    a: 10,
    b: 20,
    c: 30,
    [Symbol(Symbol.iterator)]: [GeneratorFunction: [Symbol.iterator]]
  },
  done: false
}
```



Arrow functions change the way *this* is bound

```
const collection = {
 a: 10,
 b: 20,
 c: 30,
 [Symbol.iterator]() {
   let i = 0;
   const values = Object.keys(this);
   return {
     next: () => {
       return {
         value: this,
         done: i > values.length
       }
     }
   };
 }
};
```

```
const collection = {
 a: 10,
 b: 20,
 c: 30,
 [Symbol.iterator]: function() {
   let i = 0;
   const values = Object.keys(this);
   let nextBound = function() {
     return {
       value: this,
       done: i > values.length
     }
   }.bind(this);
   return {
     next: nextBound,
   };
 }
};
```

```
const iterator = collection[Symbol.iterator]();
console.log(iterator.next());
```

```
{
  value: {
    a: 10,
    b: 20,
    c: 30,
    [Symbol(Symbol.iterator)]: [Function: [Symbol.iterator]]
  },
  done: false
}
```

Generators Makes implementation of an Iterator easier

- Generators maintain their internal state
- We can use the yield* expression to delegate to another generator or iterable object
- Using Generators allows for less cluttered syntax

But

If you need to use an object that has properties other than next
Or
If the iteration logic is complex

Using generators will not help much and could even add more complexity

The convenience of using a Generator to implement an Iterator

```
function objectEntries(obj) {
 let index = 0;
 let propKeys = Reflect.ownKeys(obj);

 return {
   [Symbol.iterator]() {
     return this;
   },
   next() {
     if (index < propKeys.length) {
       let key = propKeys[index];
       index++;
       return { value: [key, obj[key]] };
     } else {
       return { done: true };
     }
   }
 };
}
```

```
function* objectEntries(obj) {
 const propKeys = Reflect.ownKeys(obj);

 // Reflect.ownKeys method returns an Iterable
 // so we can use for ... of to loop on the returned object
 for (const propKey of propKeys) {
   yield [propKey, obj[propKey]];
 }
}
```

The implementation of the Iterator requires us to manage the state of the index variable created through a closure.

Using a Generator allows us to use a loop and yield a value. We don't need to specify a condition to stop the iteration.

Questions?

Quiz time!

1. What is stored in the variable `generator` after line 9 is executed?
2. What are the results of the `console.log` statements on lines 10, 11, 12, and 13?

```
1   function* peopleGenerator() {
2     let people = ['Mandy', 'Giuseppe', 'Jack'];
3
4     for (let person of people) {
5       yield person;
6     }
7   }
8
9   let generator = peopleGenerator();
10  console.log(generator.next().value);
11  console.log(generator.next().value);
12  console.log(generator.next().value);
13  console.log(generator.next().value);
```

1. The generator object
2. Mandy, Giuseppe, Jack, and `undefined`

```
1   function* peopleGenerator() {
2     let people = ['Mandy', 'Giuseppe', 'Jack'];
3
4     for (let person of people) {
5       yield person;
6     }
7   }
8
9   let generator = peopleGenerator();   // Object [Generator] {}
10  console.log(generator.next().value); // Mandy
11  console.log(generator.next().value); // Giuseppe
12  console.log(generator.next().value); // Jack
13  console.log(generator.next().value); // undefined
```