Promises, Iterators, and Generators

Level 6

Iterators

Level 6 – Section 2

What We Know About Iterables So Far

Arrays are **iterable** objects, which means we can use them with *for...of*.

```
let names = ["Sam", "Tyler", "Brook"];

for(let name of names){
  console.log( name );
}
```

```
> Sam
> Tyler
> Brook
```

Plain JavaScript objects are **not iterable,** so they **do not work** with *for...of* out-of-the-box.

```
let post = {
  title: "New Features in JS",
  replies: 19
};

for(let p of post){
  console.log(p);
}
```

```
> TypeError: post[Symbol.iterator] is not a function
```

Iterables Return Iterators

Iterables return an **iterator** object. This object knows how to **access items from a collection** 1 at a time, while **keeping track of its current position** within the sequence.

```
let names = ["Sam", "Tyler", "Brook"];

for(let name of names){
   console.log( name );
}
```

What's really happening behind the scenes

```
let iterator = names[Symbol.iterator]();

let firstRun = iterator.next();
let name = firstRun.value;

let secondRun = iterator.next();
let name = secondRun.value;

let thirdRun  = iterator.next();
let name = thirdRun.value;

let fourthRun = iterator.next();
```

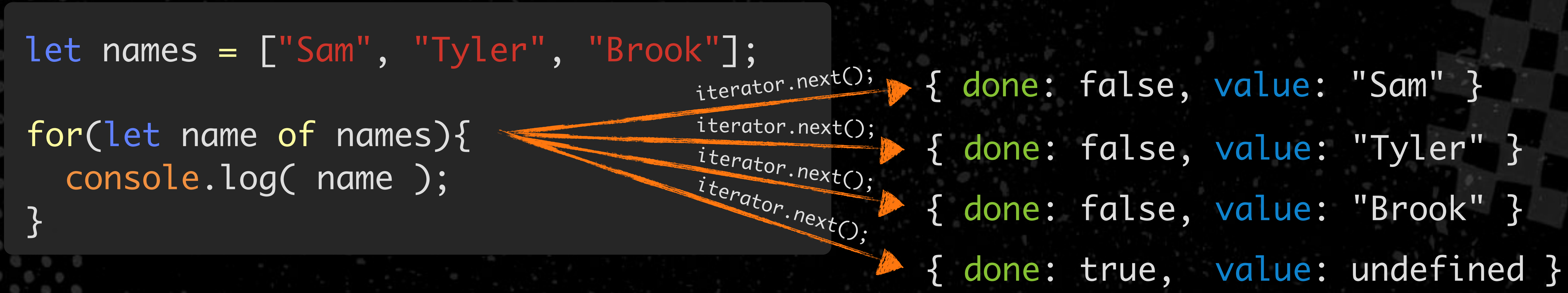
```
{ done: false, value: "Sam" }

{ done: false, value: "Tyler" }

{ done: false, value: "Brook" }
```

Breaks out of the loop when done is true

```
{ done: true, value: undefined }
```

Understanding the next Method

Each time *next()* is called, it returns an object with **2** specific properties: *done* and *value.*



```
let names = ["Sam", "Tyler", "Brook"];

for(let name of names){
  console.log( name );
}
```

```
iterator.next();  { done: false, value: "Sam" }
iterator.next();  { done: false, value: "Tyler" }
iterator.next();  { done: false, value: "Brook" }
iterator.next();  { done: true,  value: undefined }
```

Here's how values from these 2 properties work:

done (boolean)

- Will be *false* if the iterator is able to return a value from the collection
- Will be *true* if the iterator is past the end of the collection

value (any)

- Any value returned by the iterator. When *done* is *true*, this returns *undefined*.

The First Step Toward an Iterator Object

An iterator is an object with a *next* property, returned by the result of calling the *Symbol.iterator* method.

```
let post = {
  title: "New Features in JS",
  replies: 19
};

post[Symbol.iterator] = function(){

  let next = () => {

  }

  return { next };
};
```

Iterator object

```
for(let p of post){
  console.log(p);
}
```

```
> Cannot read property 'done' of undefined
```

Different error message... We are on the right track!

Navigating the Sequence

We can use *Object.keys* to build an array with property names for our object. We'll also use a counter (*count*) and a boolean flag (*isDone*) to help us navigate our collection.

```
let post = { //... };

post[Symbol.iterator] = function(){

  let properties = Object.keys(this);
  let count = 0;
  let isDone = false;

  let next = () => {

  }

  return { next };
};
```

Returns an array with property names

Allows us to access the properties array by index

Will be set to true when we are done with the loop

Returning done and value

We use *count* to keep track of the sequence and also to fetch values from the *properties* array.

```
let post = { //... };

post[Symbol.iterator] = function(){

  let properties = Object.keys(this);
  let count = 0;
  let isDone = false;

  let next = () => {
    if(count >= properties.length){
      isDone = true;
    }
    return { done: isDone, value: this[properties[count++]] };
  }
  return { next };
};
```

Ends the loop after reaching the last property

Fetches the value for the next property

++ only increments count after it's read

this refers to the post object

Running Our Custom Iterator

We've successfully made our plain JavaScript object **iterable**, and it can now be used with *for…of*.

```
let post = {
  title: "New Features in JS",
  replies: 19
};

post[Symbol.iterator] = function(){
  //...
  return { next };
};

for(let p of post){
  console.log(p);
}
```

```
> New Features in JS
> 19
```

Iterables With the Spread Operator

Objects that comply with the iterable protocol can also be used with the **spread operator**.

```
let post = {
  title: "New Features in JS",
  replies: 19
};

post[Symbol.iterator] = function(){
  //...
  return { next };
};

let values = [...post];
console.log( values );
```

```
> ['New Features in JS', 19]
```

Groups property values and returns an array

Iterables With Destructuring

Lastly, **destructuring** assignments will also work with iterables.

```
let post = {
  title: "New Features in JS",
  replies: 19
};

post[Symbol.iterator] = function(){
  //...
  return { next };
};

let [title, replies] = post;
console.log( title );
console.log( replies );
```

```
> New Features in JS
> 19
```