

# LS Async Seminar

Introduction & Chapter 01: Custom Iterators

Tyler Hurley, Nirab Pant, Steve Gontzes

November 7th, 2021

# Today's Material

- Brief Introduction to Symbols
- Custom Synchronous Iterators
  - Iterators and Iterables
  - Using *[Symbol.iterator]* to create custom for...of behavior
- Asynchronous Execution
  - Promises, Fetch and XHR APIs, Async/Await Keywords
- Custom Asynchronous Iterators

# Brief Introduction to Symbols

# Symbols are Unique

- Symbols are a recently added primitive data type that provide developers with a **guaranteed unique value**.

```
let hello = Symbol("hello");  
let newHello = Symbol("hello");  
  
console.log(hello === newHello);  
// false, symbols are guaranteed to be unique,  
// regardless of description
```

# Symbols Help Avoid Key Collision

```
let hello = Symbol("hello");

let obj = {
  hello: "world"
};

obj.hello = "string property!";
// example of key collision, reassigns existing property
obj[hello] = "new world";
// creates new key, value pair with the symbol stored in `hello` as its key

console.log(obj);
// {
//   hello: 'string property!',
//   [Symbol(hello)]: 'new world'
// }
```

# Well-Known Symbols

- Throughout this discussion, we will be taking advantage of several well-known symbols such as *Symbol.iterator* and *Symbol.asyncIterator* which are used to set the default iterator/async iterator for a given iterable
- Well-known symbols are stored as properties on the Symbol constructor function which is why you can access them using *Symbol.iterator* and *Symbol.asyncIterator*.

```
>> Object.getOwnPropertySymbols(Array.prototype)
<  Array [ Symbol(Symbol.iterator), Symbol("Symbol.unscopables") ]
    0: Symbol(Symbol.iterator)
    1: Symbol("Symbol.unscopables")
    length: 2
    ▶ <prototype>: Array []

>> Object.getOwnPropertyNames(Symbol)
<  Array(18) [ "for", "keyFor", "prototype", "isConcatSpreadable",
"iterator", "match", "replace", "search", "species", "hasInstance", ... ]
    0: "for"
    1: "keyFor"
    2: "prototype"
    3: "isConcatSpreadable"
    4: "iterator"
    5: "match"
    6: "replace"
    7: "search"
    8: "species"
    9: "hasInstance"
   10: "split"
   11: "toPrimitive"
   12: "toStringTag"
   13: "unscopables"
   14: "asyncIterator"
   15: "matchAll"
   16: "length"
   17: "name"
    length: 18
    ▶ <prototype>: Array []
```

# Custom Synchronous Iterators

# Iterables and Iterators

An iterable is an object:

- that allows its values to be looped over in a *for...of* construct
- with either `[Symbol.iterator]` or `[Symbol.asyncIterator]` method implemented that returns an iterator object that can loop through the values

1: Iterable object

2: `[Symbol.iterator]` method

3: Iterator object

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



# Synchronous Iterators

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

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

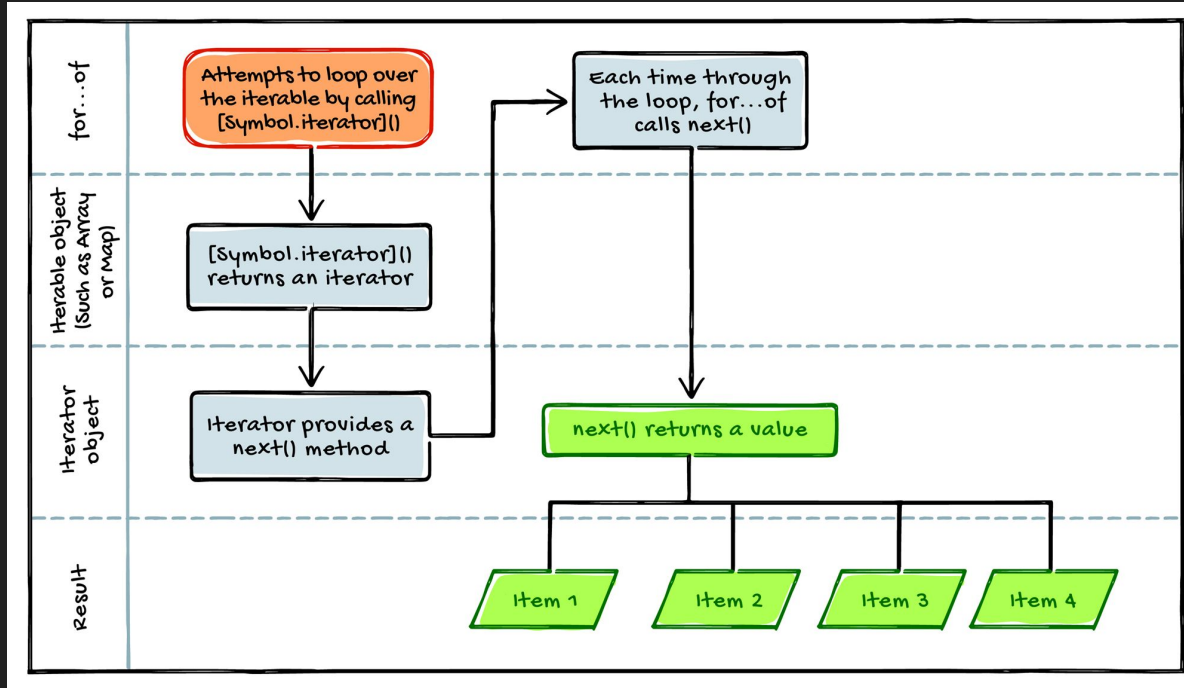
console.log(iterator.next());
// => {value: 10, done: false}
console.log(iterator.next());
// => {value: 20, done: false}
console.log(iterator.next());
// => {value: 30, done: false}
console.log(iterator.next());
// => {value: undefined, done: true}
```

-----

```
for (const value of collection) {
  console.log(value);
}

// logs:
// => 10
// => 20
// => 30
```

# Iterating an Iterable with `for..of`



## A simplified view of the implementation of the “for...of” construct

```
function myForOfFunc(collection, func) {  
  let iterator = collection[Symbol.iterator]();  
  let current = iterator.next();  
  while (!current.done) {  
    func(current.value);  
    current = iterator.next();  
  }  
}  
  
function displayItem(item) {  
  console.log(item);  
}
```

```
const collection = {  
  a: 10,  
  b: 20,  
  c: 30,  
  [Symbol.iterator]() {...}  
}  
  
myForOfFunc(collection, displayItem);  
  
// logs  
// ⇒ 10  
// ⇒ 20  
// ⇒ 30
```

## Understanding synchronous iterators forwards and backwards...

```
const array = [1, 2, 3];

array[Symbol.iterator] = function() {
  let count = this.length;
  return {
    next: () => {
      return {
        value: this[--count],
        done: count < 0,
      }
    }
  }
}
```

```
for (const item of array) {
  console.log(item);
}

// logs
// => 3
// => 2
// => 1

console.log([...array])
// [ 3, 2, 1 ]

console.log(array);
// [ 1, 2, 3, [Symbol(Symbol.iterator)]:
// [Function (anonymous)] ]
```

## Checking if an Object is Iterable

```
function isIterable(object) {  
  return typeof object[Symbol.iterator] === "function";  
}
```

```
isIterable("this is a string")  
// ⇒ true
```

```
isIterable([1,2,3,4])  
// ⇒ true
```

```
isIterable({first: 'gold', second: 'silver', third: 'bronze'})  
// ⇒ false
```

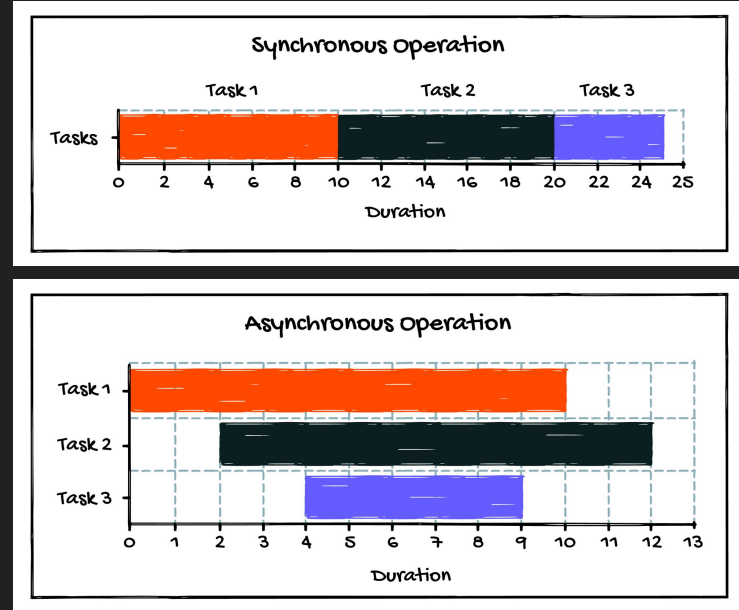
# Asynchronous Execution

# Why is Asynchronous Execution Important?

# Asynchronous Execution

*“The concept of asynchrony determines whether a task can start executing before another task is finished.*

*In a synchronous execution, the program pauses until the current task is completed before moving to the next task. But in an asynchronous execution, the program continues executing even when the previous operation hasn't finished yet.” - Kelhini*





How do we write asynchronous code?

# Promises

- A *Promise* is an object that guarantees an eventual value of an asynchronous operation upon successful completion or a reason for failure otherwise
- Unlike the callback model, with *Promises*, we can now work with returned objects, simplifying syntax and allowing chaining (instead of nesting)
- Promises can have 3 states:
  - Pending
  - Fulfilled
  - Rejected

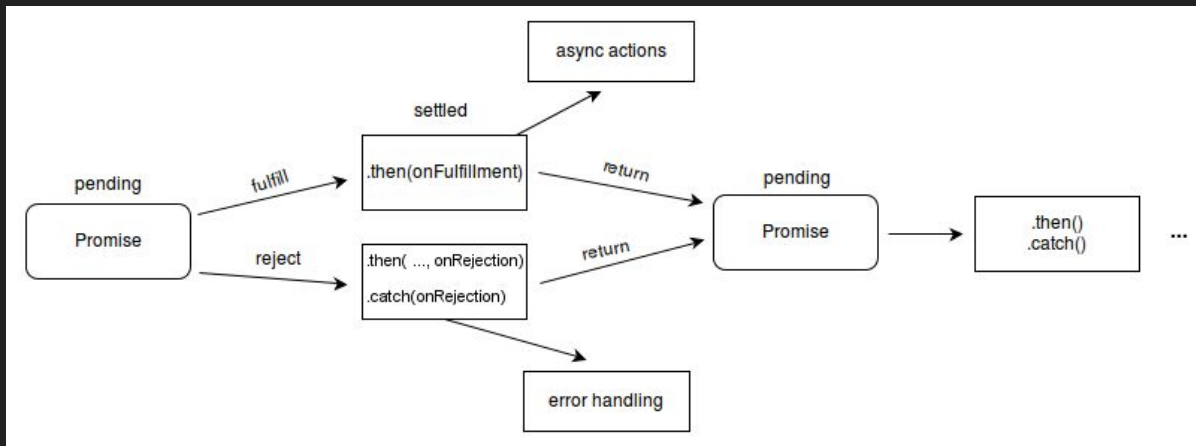


Diagram credit: [MDN](#)

## XMLHttpRequest API

```
const request = new XMLHttpRequest();
request.open('GET', '/examples/1.json');
request.responseType = 'json';

request.addEventListener('load', event => {
  // process request.response
});

request.addEventListener('error', event => {
  console.log('Error! ' + request.responseText);
});

request.send();
```

## Fetch API (promise based)

```
const promise =
  fetch('/examples/1.json');

promise.then((result) => {
  // process result
}, (error) => {
  console.log(error);
});
```

# Promises

```
function fetchExample() {  
  const promise =  
    fetch('/examples/1.json');  
  
  return promise  
    .then((result) => result.json())  
    .then((json) => console.log(json))  
    .catch((error) => {  
      console.error('Caught: ' +  
error.message);  
    });  
}  
  
fetchExample();
```

# Async / Await

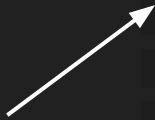
```
async function fetchExample() {  
  const promise = fetch('/examples/1.json');  
  
  try {  
    let result = await promise;  
    let json = await result.json();  
    console.log(json);  
  } catch (error) {  
    console.error('Caught: ' + error.message);  
  }  
}  
  
fetchExample();
```

# Custom Asynchronous Iterators

# Asynchronous Iterators

## Synchronous Iterator

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



Asynchronous Iterator: returns a *Promise* object instead of an ordinary object

```
[Symbol.asyncIterator]() {
  let i = 0;
  const keys = Object.keys(this);
  return {
    next: () => {
      return new Promise((resolve, reject) => {
        setTimeout(() => {
          resolve({
            value: this[keys[i++]],
            done: i > keys.length
          });
        }, 1000);
      });
    }
  };
}
```

```

const collection = {
  a: 10,
  b: 20,
  c: 30,
  [Symbol.asyncIterator]() {
    let i = 0;
    const keys = Object.keys(this);
    return {
      next: () => {
        return new Promise((resolve, reject) => {
          setTimeout(() => {
            resolve({
              value: this[keys[i++]],
              done: i > keys.length
            });
          }, 1000);
        });
      }
    };
  }
};

```

```

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

iterator.next().then(result => console.log(result));
// => {value: 10, done: false}

iterator.next().then(result => console.log(result));
// => {value: 20, done: false}

iterator.next().then(result => console.log(result));
// => {value: 30, done: false}

iterator.next().then(result => console.log(result));
// => {value: undefined, done: true}

```

## Retrieving URLs Separately

```
const srcArr = [url1, url2, url3];

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

      };

    }

  };

};
```

Note: “for... await... of” construct needs to be used within an *async* function -- use IIFE!

```
(async function() {

  try {

    for await (const item of srcArr) {

      console.log(item);

    }

  } catch (error) {

    console.log("Caught: " + error.message);

  }

})();

// logs
// ⇒ { firstName: 'John', lastName: 'Doe' }
// ⇒ { firstName: 'Anna', lastName: 'Smith' }
// ⇒ { firstName: 'Peter', lastName: 'Jones' }
```



# Promise.resolve()