Introduction to JavaScript Collections

Created	@January 1, 2022 7:04 PM
Type	Lecture
Week	2
Lecture #	1
Lecture URL	https://youtu.be/3fv1ejKqiUU
Notion URL	https://21f1003586.notion.site/Introduction-to-JavaScript-Collections-70d952a1b02b44fe85d20c3599d17ade

Basic Arrays

- Collection of objects of any type
  - Can be a mixed type too
    - Numbers, Strings, Objects, Functions
- Element access
- Length
- Holes
- Iteration

Iteration

- Go over all the elements in a collection
- Concepts:
  - **Iterable** → An object whose contents can be accessed sequentially
  - **Iterator** → Pointer to the next element
- Iterable objects:
  - Array
  - String
  - Map
  - Set

    - Browser DOM — tree structure
- Objects:
    - `object.keys()`
    - `object.entries()`
    - These are some helper functions

Iterations and Transformations

- Functions that take functions as input
- `map, filter, find`
    - Apply a callback function over each element of array
- Elements of functional programming
    - Create a transformation chain

Callback → Function passed in to another function, to be called back for some purpose ***[IMPORTANT]***

Other Collections

- Maps → Proper dictionaries instead of objects
- WeakMaps
- Sets

More advanced stuff → use only when needed

Destructuring

- Simple syntax to split an array into multiple variables
- Easier to pass and collect arguments, etc.
- Also possible for objects

Generators

- Functions that `yield` values one at a time
- Computed iterables
- Dynamically generate iterators

Basic array

```
let x = [1, 2, 3, 4];
console.log(`${typeof(x)}: ${x} with length = ${x.length}`);
console.log(x[0]);
```

Output

```
object: 1,2,3,4 with length = 4
1
```

Mixed element array

```
let y = [1, 'b', a => a + 1];
console.log(`${typeof(y)}: ${y} with length = ${y.length}`);
console.log(x.length, y.length);
```

Output

```
object: 1,b,a => a + 1 with length = 3
4 3
```

Delete

```
x = [];
console.log(`${typeof(x)}: ${x} with length = ${x.length}`);
```

Holes in the array

This creates holes in the array

We have changed the size of array without adding any element, so the elements are undefined

```
y.length = 10;
console.log(`${typeof(y)}: ${y} with length = ${y.length}`);
```

JavaScript Collections - Iterations and Destructuring

Created	@January 2, 2022 1:00 AM
Type	Lecture
Week	2
Lecture #	2
Lecture URL	https://youtu.be/WUehmp1bwaY
Notion URL	https://21f1003586.notion.site/JavaScript-Collections-Iterations-and-Destructuring-aba4f319800a43d0807ae1411517f05d

Iteration

```
let x = [1, 'b', a => a + 1];
x.length = 5;

for(let i = 0; i < x.length; ++i) {
  console.log(`x: ${x[i]} of type ${typeof(x[i])}`);
}
```

Output

```
x: 1 of type number
x: b of type string
x: a => a + 1 of type function
x: undefined of type undefined
x: undefined of type undefined
```

`in` iteration

The `in` keyword skips the undefined entries

```
// "i" here takes the index values
for(const i in x) {
  console.log(`x: ${x[i]} of type ${typeof(x[i])}`);
}
```

Output

```
x: 1 of type number
x: b of type string
x: a => a + 1 of type function
```

`of` iteration

The `of` keyword also skips the undefined entries

```
// "i" here takes the element values
for(const i in x) {
  console.log(`x: ${i} of type ${typeof(i)}`);
}
```

Output

```
x: 0 of type string
x: 1 of type string
x: 2 of type string
```

Are objects iterable?

No, unless they implement the iterable protocol

```
let x = {'a': 1, 'b': 'alpha', 'c': [3, 2, 1]};

// The following loop will work
// "in" operator iterates over the list of keys in that object
for(const i in x) {
  console.log(`${i}: ${x[i]}`)
}

// The following loop will NOT work
// "of" operator tries to iterate over the values of the object
// That iteration is NOT defined
for(const i of x) {
  console.log(i)
}

// Using this loop we can iterate over the object, sort of
for(const [k, v] of Object.entries(x)) {
  console.log(`{k}: {v}`)
}
```

Create an array with holes

```
// The following code creates an empty array
let x = new Array(5);
x[1] = 10;
x[3] = 'hello';

for(const [k, v] of x.entries()) {
  console.log(`Index ${k}, value: ${v} of type ${typeof(v)}`);
}
```

Output

```
Index 0, value: undefined of type undefined
Index 1, value: 10 of type number
Index 2, value: undefined of type undefined
Index 3, value: hello of type string
Index 4, value: undefined of type undefined
```

Now, when we use the `in` operator to iterate over the array, it skips the `undefined` elements

```
for(const i in x) {
  console.log(`Index ${i}, value: ${x[i]} of type ${typeof(x[i])}`);
}
```

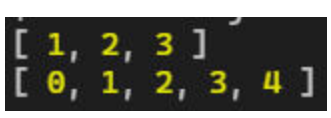
Output

Spread (ᵔᴥᵔ)

```
let x = [1, 2, 3];
let y = [0, ...x, 4];

console.log(x);
console.log(y)
```

Output



The spread operator `...` allows an iterable, like array or string, to be expanded in places where zero or more arguments (in functions) or elements (in arrays) are expected. Source

Iteration and Transformation

`find()`

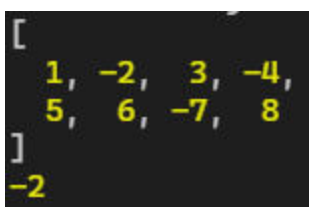
The `find()` method returns the value of the first element in the provided array that satisfies the provided testing function

If no values satisfy the function, `undefined` is returned

```
let x = [1, -2, 3, -4, 5, 6, -7, 8];
let y = x.find(i => i < 0);

console.log(x);
console.log(y);
```

Output



`filter()`

The `filter()` method ***creates a new array*** with all the elements that pass the testing condition implemented by the provided function

```
console.log(x.filter(i => i < 0));
```

Output

`map()`

The `map()` method ***creates a new array*** populated with the results of calling a provided function on every element in the calling array

```
console.log(x.map(i => i > 0 ? '+' : '-'));
```

Output

`reduce()`

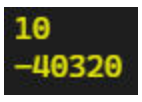
The `reduce()` method executes a user-supplied "reducer" callback function on each element of the array, in order, passing in the return value from the calculation on the preceding element

The final result of running the reducer across all elements of the array is a single value

Here, `a` is the accumulator, and `0` (or `1`) is the initial value, and the step written before them is the statement to be executed on every single element of the array, and their result is passed to the next element in the `a` variable

```
console.log(x.reduce((a, i) => a + i, 0));
console.log(x.reduce((a, i) => a * i, 1));
```

Output



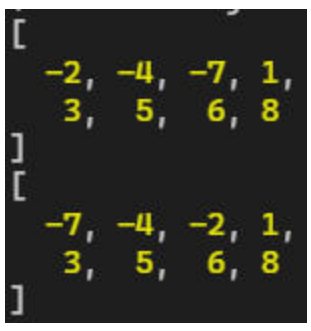
`sort()`

The `sort()` function sorts the elements of an array in place and returns the sorted array

The default sorting order is ascending, built upon converting the elements into strings, the comparing their sequences of UTF-16 code units value

```
console.log(x.sort());                // INCORRECT
console.log(x.sort((a, b) => a - b));  // CORRECT
```

Output



Destructuring

```
let x = [1, 2, 3];
let [a, b] = x;
console.log(x);
console.log(a);
console.log(b);
```

Output

Object Destructuring

```
const person = {
  firstName: 'Albert',
  lastName: 'Pinto',
  age: 25,
  city: 'Mumbai'
};

const {firstName: fn, city: c} = person;
console.log(fn);
```

```
console.log(c);

const {lastName, age} = person;
console.log(lastName);
console.log(age);
```

Output

Here, we are destructuring the object and take various keys and put their values in some variables

Another way to destructure

```
const {firstName, ...rem} = person;
console.log(firstName);
console.log(rem);
```

Output

JavaScript - Modularity

Created	@January 2, 2022 6:35 PM
Type	Lecture
Week	2
Lecture #	3
Lecture URL	https://youtu.be/On0u6m2LXeE
Notion URL	https://21f1003586.notion.site/JavaScript-Modularity-45167e804dae43c2a58fc37f5cbbf74e

Modules

- Quite crucial for re-usability of code
- Collect related functions, objects, values together
- `export` values for use by other scripts
- `import` values from other scripts, packages

Ways of implementing

- script → directly include the script inside the browser
- CommonJS → Introduced for server side modules
  - synchronous load: server blocks till module is loaded
- AMD → Asynchronous Module Definition
  - browser side modules

ECMAScript 6 and above:

- ES6 modules
  - Both servers and browsers
  - Asynchronous load

npm

- Node Package Manager
- Node:
  - command line interface for JS

    - Mainly used for backend code, can also be used for testing
- npm can also be used to package modules to the frontend
    - “Bundle”managers → webpack, rollup, etc

To import a JS module in HTML file

```
<script type="module" src="file.js"></script>
```

Direct one line export

```
export const c = 300000000;
```

Basic import

NOTE: This file must be included as `type="module"` in the HTML `script` tag

```
import {c} from './path/to/file.js';
console.log(c);
```

Explicit named exports

```
function sq(x) {
  return x * x;
}

export const c = 299792458;
export function energy(m) {
  return m * sq(c);
}
```

Similarly, to use the above export

```
import {c, energy} from './path/to/file.js';
console.log(c);
console.log(energy(10));
```

Renaming imports

```
import {c as speedOfLight} from './path/to/file.js';
console.log(speedOfLight);
```

Another way to write exports

```
function sq(x) {
  return x * x;
}

const speedOfLight = 299792458;
function e(m) {
  return m * sq(speedOfLight);
}

export {
  speedOfLight as c,
  e as energy
};
```

Default export

```
const c = 299792458;
export default function(m) {
  return m * c * c;
}
```

How to import a default export?

```
import energy from './path/to/file.js';
console.log(energy(100));
```

When we import a variable, we get a read-only view of that variable, we cannot modify it's value

However, we can call a function (supposedly imported) that can modify the value of that variable

Objects

- Everything is an object in JS
- Object literals
  - Assign values to named parameters in object
- Object methods
  - Assign functions that can be called on an object
- Special variable `this`
- Function methods
  - `call(), apply(), bind()`
- `Object.keys(), values(), entries()`
  - Use as dictionary
  - Iterators

Prototype based inheritance

- Object can have a "prototype"
- Automatically get properties of parent
- Single inheritance track

Class

- Better syntax — still prototype based inheritance
- constructor must explicitly call `super()`
- Multiple inheritance or Mixins
  - Complex to implement

Basic object literals and methods

```
let xx = {'a': 5, 'b': 'hello'};
console.log(xx);
xx.add = function(x, y) {
  return x + y;
}
console.log(xx);
console.log(`xx is of type ${typeof(xx)}`);
console.log(`xx.add is of type ${typeof(xx.add)}`);
console.log(`Evaluate the function xx.add(3, 5) gives ${xx.add(3, 5)}`);
```

Output

```
{ a: 5, b: 'hello' }
{ a: 5, b: 'hello', add: [Function (anonymous)] }
xx is of type object
xx.add is of type function
Evaluate the function xx.add(3, 5) gives 8
```

What is `this`?

`this` refers to the object it belongs to

Analogous to `self` in Python

```
xx.f = function(x) {
  return this.a + x;
}

console.log(xx.f(100));
```

Output

`105`

Copying an object

```
let x = {a: 1, b: 2};
let y = x;
x.a = 3;
```

In the previous case, `y` will point to the same memory location as `x`, hence changing anything in `x` will reflect in `y` also, and vice-versa

So, how to copy objects?

```
let z = {...x};
x.a = 5;
```

In this case, `z` gets its own copy of the object that `x` had

JavaScript - Function methods, Prototypes & Classes

Created	@January 2, 2022 11:20 PM
Type	Lecture
Week	2
Lecture #	4
Lecture URL	https://youtu.be/fCyF0OK2FAw
Notion URL	https://21f1003586.notion.site/JavaScript-Function-methods-Prototypes-Classes-c487f0c36c694de6aa3a997d76cdc793

`get` and `set` properties

```
let user = {
  first: 'Albert',
  last: 'Pinto',
  get full() {
    return this.first + ' ' + this.last;
  },
  set full(f) {
    const parts = f.split(' ');
    this.first = parts[0];
    this.last = parts[1];
  }
};

console.log(user.full);
user.full = 'Gabbar Singh';
console.log(`Now ${user.first} and ${user.last}`);
```

Output

```
Albert Pinto
Now Gabbar and Singh
```

Call the method of an object in context of the same or another object

```
let xx = {
  'a': 5,
```

```
  'b': 'hello',
  'add': function(x, y) {
    return x + y + this.a;
  }
};
let z = xx.add;
console.log(z.call("", 3, 4));  // NaN
console.log(z.call(xx, 3, 4));  // 12
```

`apply()`

This spreads the arguments

If there are extra arguments, they will be ignored

```
console.log(z.apply(xx, [1, 2, 3, 4]));
```

`bind()`

CLOSURE

A powerful way to create new functions from existing ones

```
let z2 = z.bind(xx, 2);  // Here, x takes the value 2
console.log(z2(3));      // And, y takes the value 3

// Output
10
```

Prototypes

Kinda like inheritance

```
const x = {a: 1, inc: function() {this.a++;}};
console.log(x);
const y = {__proto__: x, b: 2};
console.log(y);
console.log(y.a);
y.inc();
console.log(y.a);
```

Output

Classes

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  describe() {
    return `${this.name} makes a sound ${this.sound}`;
  }
}
let x = new Animal('Jerry');
console.log(x.describe());

class Dog extends Animal {
  constructor(name) {
    super(name);
    this.sound = 'borrk';
  }
}
let d = new Dog('Spike');
console.log(d.describe());
```

Output

```
Jerry makes a sound undefined
Spike makes a sound borrk
```

JavaScript - Asynchrony

Created	@January 3, 2022 10:10 AM
Type	Lecture
Week	2
Lecture #	5
Lecture URL	https://youtu.be/DT1NE0pUfeQ
Notion URL	https://21f1003586.notion.site/JavaScript-Asynchrony-809b84df29a944278b99ce8cc65fed66

Function calls

- Function is like a “branch”
  - but must save present state so we can return
- Call stack
  - Keep track of chain of functions called up to now
  - Pop back up out of stack

- `main()` on stack — current - calls `f()`
- `f()` goes on stack — calls `g()`
- `g()` goes on stack — calls `h()`
- `h()` goes on stack — executes
- return from `h` → pop into `g`
- return from `g` → pop into `f`
- return from `f` → pop into `main`

A video on call stack: Philip Roberts: Help, I’m stuck in an event-loop. on Vimeo

Call stack visualization: http://latentflip.com/loupe/

Event Loop and Task Queue

- Task Queue: *store next task to execute*
  - Tasks are pushed into queue by events (clicks, input, network, etc.)
- Event Loop
  - Wait for call stack to become empty
  - Pop task out of queue and push it onto stack, start executing
- Run-to-completion
  - Guarantee from JavaScript runtime
  - Each task will run to completion before next task is invoked

Asynchronous programming in JavaScript • JavaScript for impatient programmers (ES2021 edition) (exploringjs.com)

Callbacks

- Long running code
  - it will block the execution till it finishes
- Push the long running code into a separate “thread” or “task”
  - Let the main code proceed
  - Call it back when completed (or failed?)

Example: reading files — synchronous

```
const fs = require('fs');

try {
  const data = fs.readFileSync('./path/to/file.txt', 'utf8');
  console.log(data);
} catch(err) {
  console.error(err);
}
```

Example: reading files — asynchronous

```
const fs = require('fs');

fs.readFile('./path/to/file.js', 'utf8', (err, data) => {
  if(err) {
    console.error(err);
    return;
  }

  console.log(data);
});
```

Asynchronous code

- Very powerful — allows JS to have high performance even though it is single threaded
- Can be difficult to comprehend
- Promises, async function calls, etc.
  - Important and useful concepts

Some useful videos

- The Async Await Episode I Promised by Fireship
- Closures Explained in 100 Seconds // Tricky JavaScript Interview Prep by Fireship
- In general, watch Fireship religiously

JavaScript - JSON API

Created	@January 3, 2022 11:36 AM
Type	Lecture
Week	2
Lecture #	6
Lecture URL	https://youtu.be/1pSu5k1XxmI
Notion URL	https://21f1003586.notion.site/JavaScript-JSON-API-4fab954d247146e19abd45ae27f86b5d

JSON

- JavaScript Object Notation
  - For serialization, communication
- Notation is frozen
  - The spec is now final, ~~*written in stone*~~
  - It means even problem cases will remain
- Usage through JSON API

JSON API

- Global namespace object `JSON`
- Main methods
  - `JSON.stringify()`
    - Converts an object into a string
  - `JSON.parse()`
    - Tries to convert a string into an object

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  describe() {
    return `${this.name} makes a sound ${this.sound}`;
  }
}
```

```
class Cat extends Animal {
  constructor(name) {
    super(name);
    this.sound = 'Meow';
  }
  static fromJson(o) {
    c = new Cat(o.name);
    c.sound = o.sound;
    return c;
  }
}


let c = new Cat('Tom');
console.log(c.describe());
```

Now, to test stringify and parse

```
let p = JSON.stringify(c);
console.log(c);
console.log(p);
```

```
let cc = Cat.fromJSON(JSON.parse(p));
console.log(cc.describe());
```