CSCI-4208: Developing Advanced Web Applications

Week 4: Lecture 6 & 7

JavaScript - Data Structures

Overview: JavaScript - Data Structures

1. Arrays
2. Stacks & Queues
3. Sets
4. Maps
5. Iterator objects
6. Spread operator & Destructuring
7. Traversal methods (Arrays)

1. Arrays

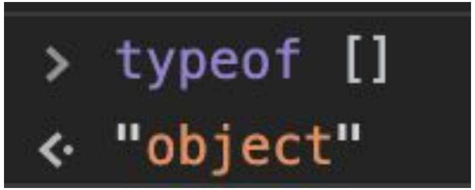
- Arrays: In JS
- Arrays: Instantiation
- Arrays: Properties
- Arrays: Accessers (Getters)
- Arrays: Mutaters (Setters)
- Arrays: Transforms (Array ↔ String)
- Arrays: Transforms (Array ↔ Array)

1. Arrays: In JS

- An array is an Indexed Collection i.e. an ordered list.

index	0	1	2	3
value	'a'	'b'	'c'	'd'

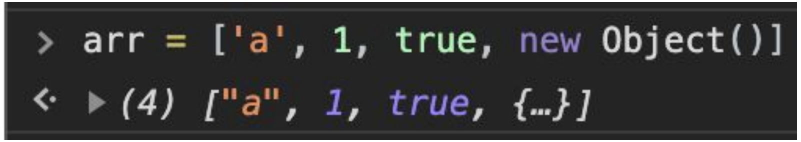
- Arrays are objects, with methods to traverse and set/get its values



```
> typeof []
<· "object"
```

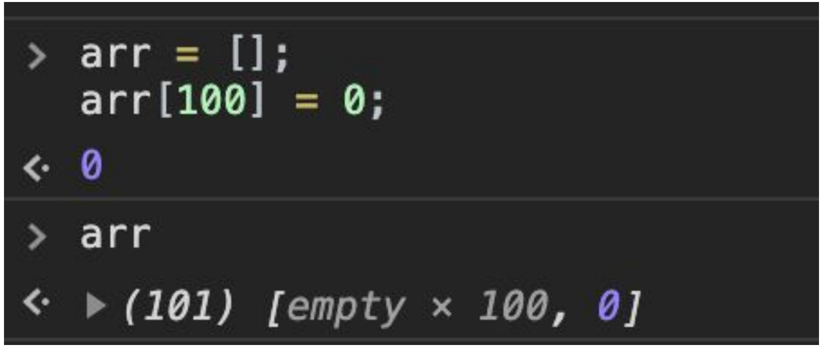
1. Arrays: In JS

- The types of its elements are not fixed.



```
> arr = ['a', 1, true, new Object()]
<· ▸ (4) ["a", 1, true, {…}]
```

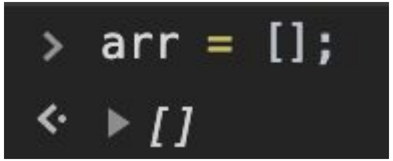
- Arrays are not guaranteed to be dense.



```
> arr = [];
  arr[100] = 0;
<· 0
> arr
<· ▸ (101) [empty × 100, 0]
```

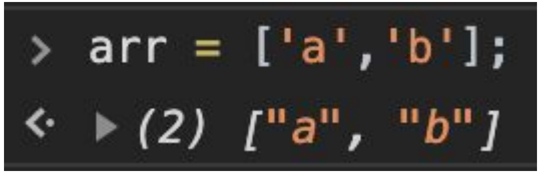
1. Arrays: Instantiate

- List Initializer *(no parameters)*



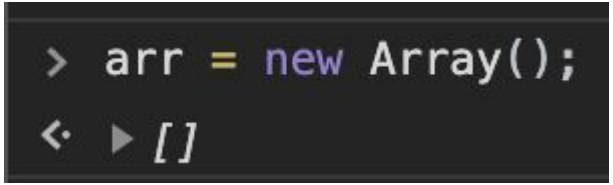
```
> arr = [];
<· ▶ []
```

- List Initializer *(parameters)*



```
> arr = ['a','b'];
<· ▶ (2) ["a", "b"]
```

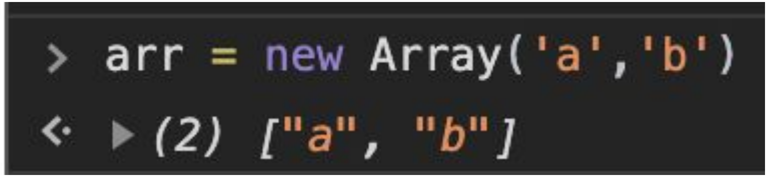
- Constructor *(no parameters)*



```
> arr = new Array();
<· ▶ []
```

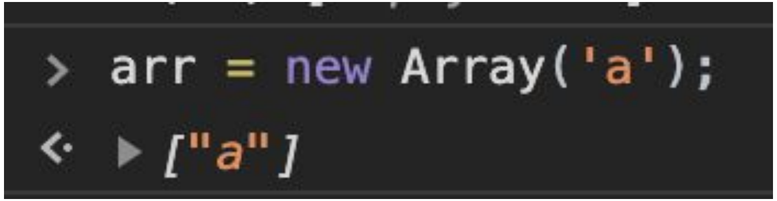
1. Arrays: Instantiate

- Constructor → multiple parameters



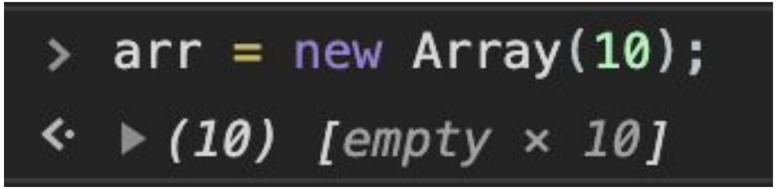
```
> arr = new Array('a','b')
<· ▶(2) ["a", "b"]
```

- Constructor → single parameters



```
> arr = new Array('a');
<· ▶["a"]
```

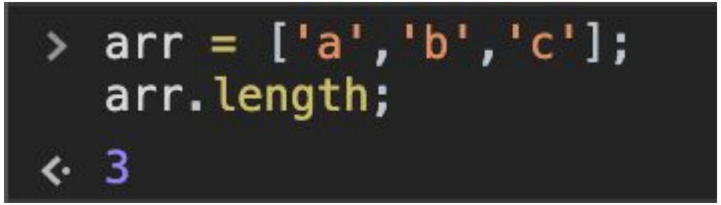
Beware:
(single int)



```
> arr = new Array(10);
<· ▶(10) [empty × 10]
```

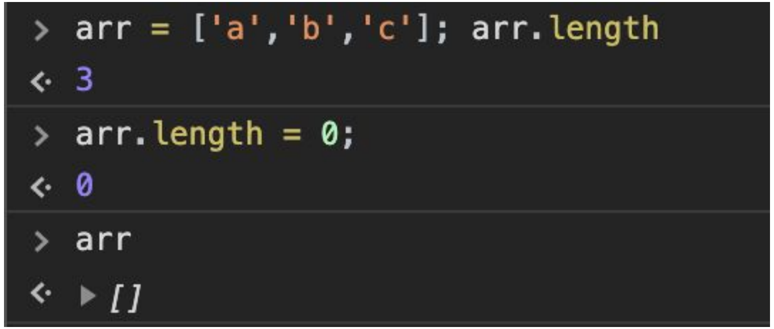
1. Arrays: Properties

- Length



```
> arr = ['a','b','c'];
  arr.length;
<· 3
```

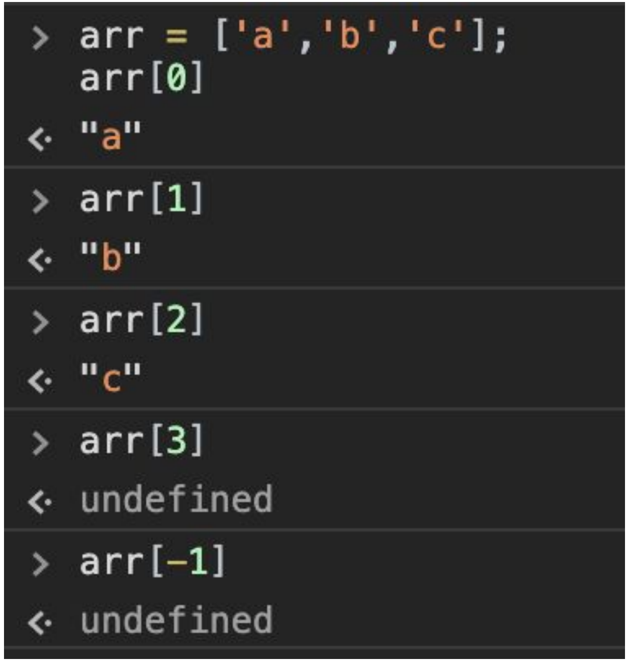
- The length of a JavaScript array is not fixed. It can grow, shrink, or be reassigned



```
> arr = ['a','b','c']; arr.length
<· 3
> arr.length = 0;
<· 0
> arr
<· ▶ []
```

1. Arrays: Accessors (Getters)

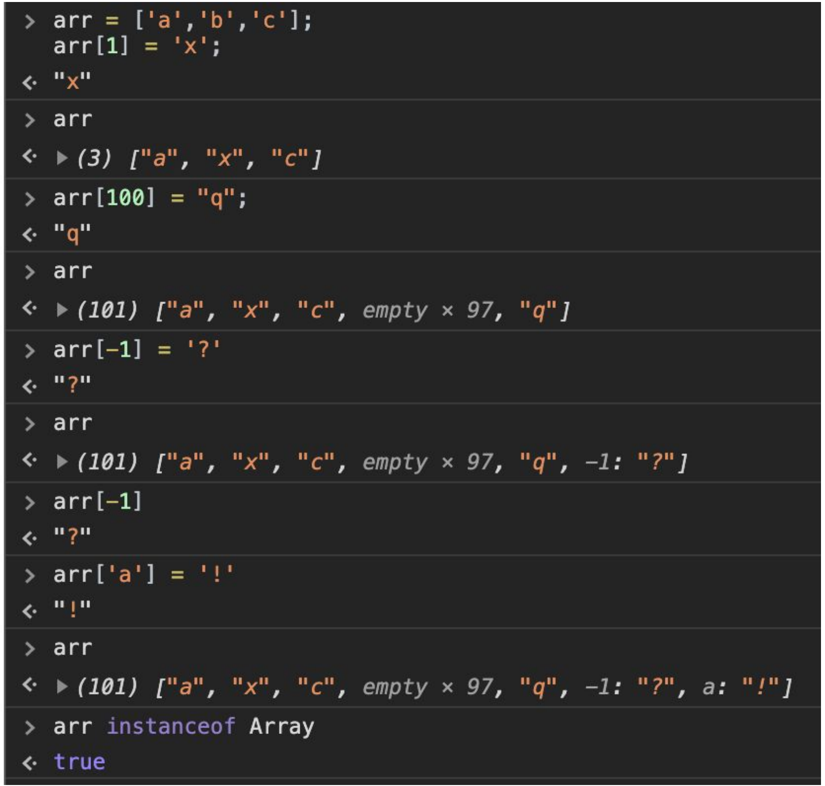
- Indexing into Arrays



```
> arr = ['a','b','c'];
  arr[0]
<· "a"
> arr[1]
<· "b"
> arr[2]
<· "c"
> arr[3]
<· undefined
> arr[-1]
<· undefined
```

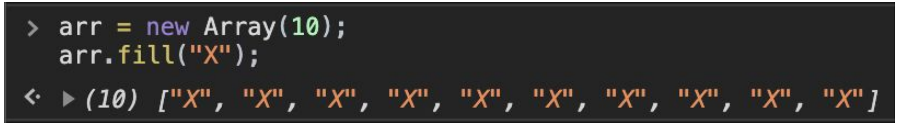
1. Arrays: Mutaters (Setters)

- Set into Array with index



```
> arr = ['a','b','c'];
  arr[1] = 'x';
<· "x"
> arr
<· ▶ (3) ["a", "x", "c"]
> arr[100] = "q";
<· "q"
> arr
<· ▶ (101) ["a", "x", "c", empty × 97, "q"]
> arr[-1] = '?'
<· "?"
> arr
<· ▶ (101) ["a", "x", "c", empty × 97, "q", -1: "?"]
> arr[-1]
<· "?"
> arr['a'] = '!'
<· "!"
> arr
<· ▶ (101) ["a", "x", "c", empty × 97, "q", -1: "?", a: "!"]
> arr instanceof Array
<· true
```

- Set into Array with fill

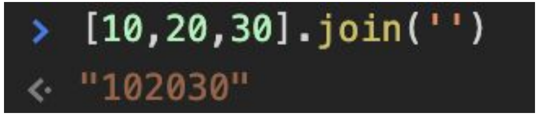


```
> arr = new Array(10);
  arr.fill("X");
<· ▶ (10) ["X", "X", "X", "X", "X", "X", "X", "X", "X", "X"]
```

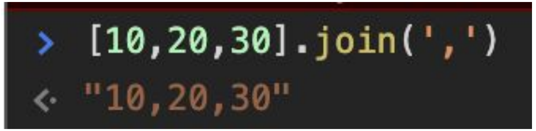
Note: (-1, 'a') are not indexes into Array

1. Arrays: Transforms (Array ↔ String)

- *Array → String:* `join(v)`

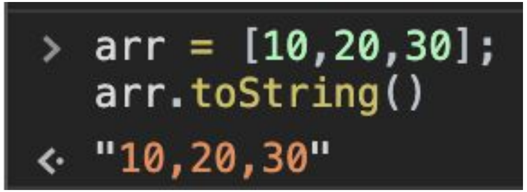


```
> [10,20,30].join('')
<· "102030"
```



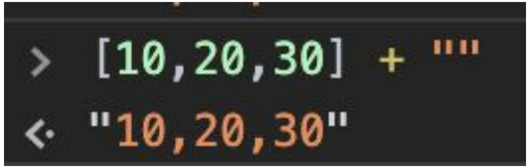
```
> [10,20,30].join(',')
<· "10,20,30"
```

- *Array → String:* `toString()`



```
> arr = [10,20,30];
  arr.toString()
<· "10,20,30"
```

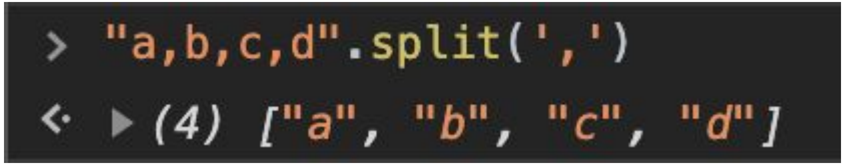
- *Array → String:* `+`



```
> [10,20,30] + ""
<· "10,20,30"
```

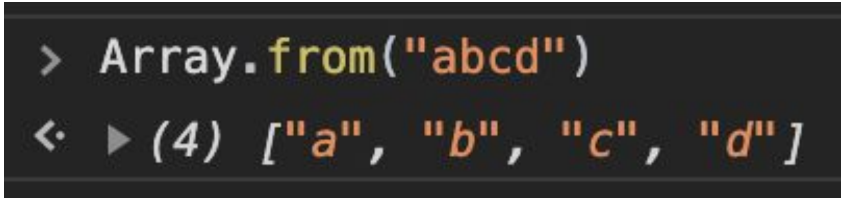
1. Arrays: Transforms (Array ↔ String)

- *String → Array:* `split()`



```
> "a,b,c,d".split(',')
<· ▶ (4) ["a", "b", "c", "d"]
```

- *String → Array:* `from()`

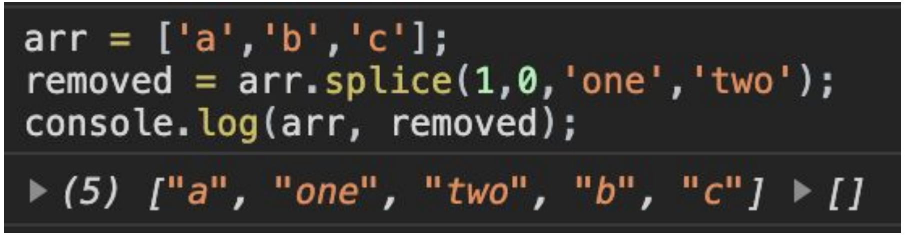


```
> Array.from("abcd")
<· ▶ (4) ["a", "b", "c", "d"]
```

1. Arrays: Transforms (Array ↔ Array)

splice

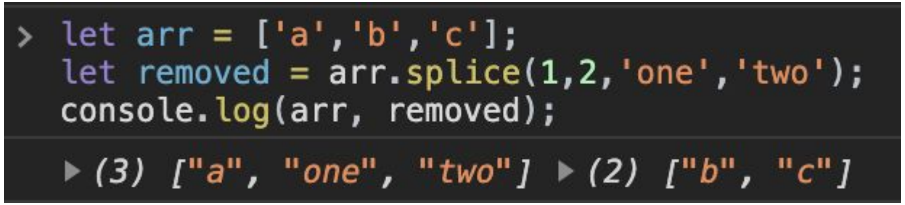
add items to array by index, no deletes.
params: index, deletes, items



```
arr = ['a','b','c'];
removed = arr.splice(1,0,'one','two');
console.log(arr, removed);

▶ (5) ["a", "one", "two", "b", "c"] ▶ []
```

add items to array by index, with deletes.
params: index, deletes, items



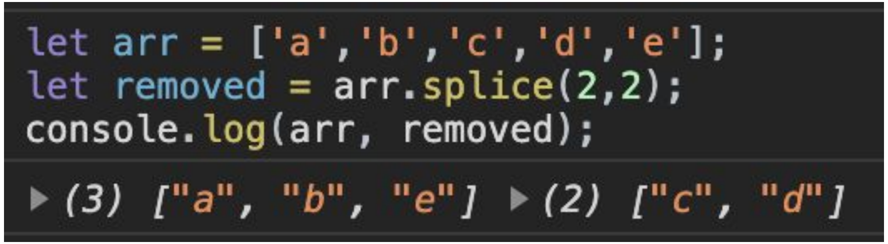
```
> let arr = ['a','b','c'];
  let removed = arr.splice(1,2,'one','two');
  console.log(arr, removed);

  ▶ (3) ["a", "one", "two"] ▶ (2) ["b", "c"]
```

1. Arrays: Transforms (Array ↔ Array)

splice

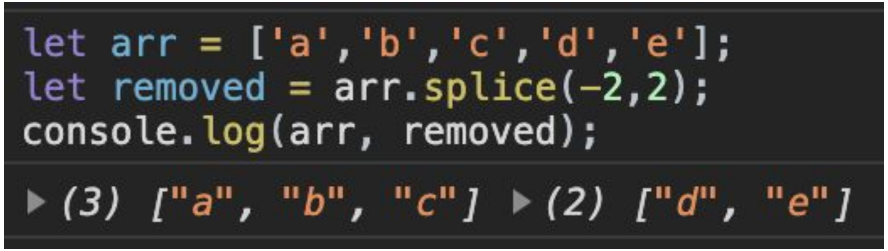
remove items by index.
params: index, deletes



```
let arr = ['a','b','c','d','e'];
let removed = arr.splice(2,2);
console.log(arr, removed);
```

```
▶(3) ["a", "b", "e"] ▶(2) ["c", "d"]
```

remove items by negative index.
params: index, deletes



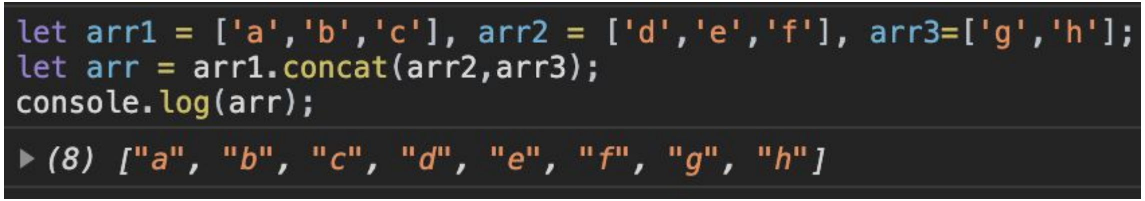
```
let arr = ['a','b','c','d','e'];
let removed = arr.splice(-2,2);
console.log(arr, removed);
```

```
▶(3) ["a", "b", "c"] ▶(2) ["d", "e"]
```

1. Arrays: Transforms (Array ↔ Array)

concat

merge arrays:
Note: concat can take any number of args



```
let arr1 = ['a','b','c'], arr2 = ['d','e','f'], arr3=['g','h'];
let arr = arr1.concat(arr2,arr3);
console.log(arr);
```

```
▶ (8) ["a", "b", "c", "d", "e", "f", "g", "h"]
```

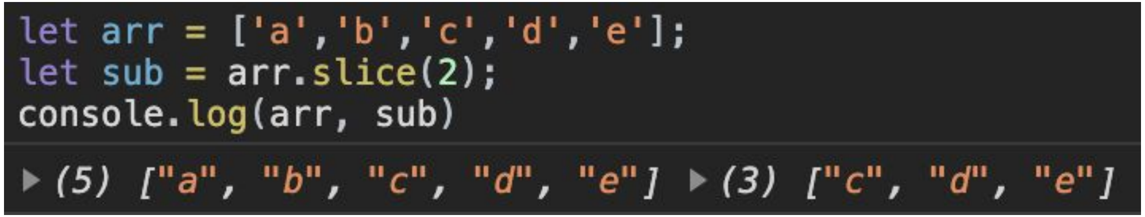
spread operator

arr = [...arr1, ...arr2, ...arr3]

1. Arrays: Transforms (Array ↔ Array)

slice

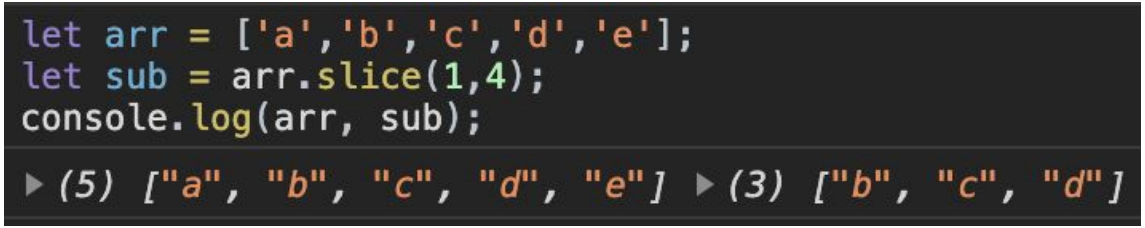
get subarray containing items starting at index



```
let arr = ['a','b','c','d','e'];
let sub = arr.slice(2);
console.log(arr, sub)
```

```
▶ (5) ["a", "b", "c", "d", "e"] ▶ (3) ["c", "d", "e"]
```

get subarray containing items starting at index and ending at index



```
let arr = ['a','b','c','d','e'];
let sub = arr.slice(1,4);
console.log(arr, sub);
```

```
▶ (5) ["a", "b", "c", "d", "e"] ▶ (3) ["b", "c", "d"]
```

2. Stacks & Queues

- Define Stacks & Queues
- push/pop
- shift/unshift
- FIFO vs FILO Operations

2. Stacks & Queues: Definitions

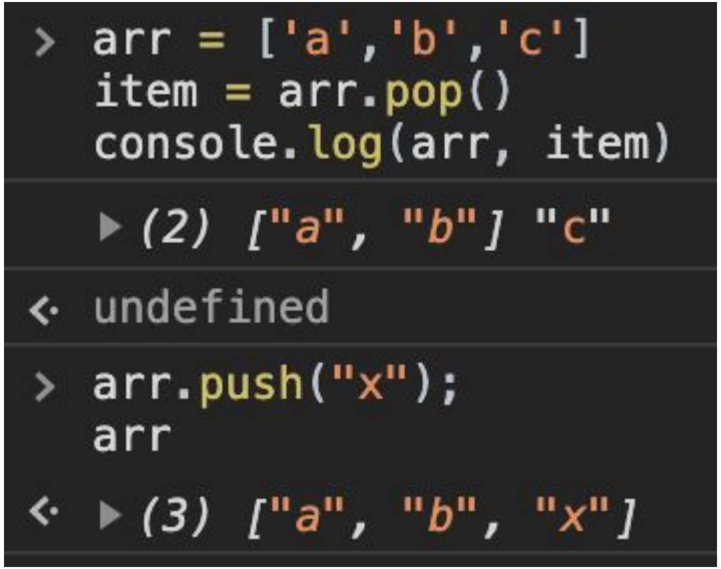
- Stacks: a linear **data structure** which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). There are many real-life examples of a **stack**. An example of pancakes.

- Queues: a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first.

- Difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

2. Stacks & Queues: Push & Pop

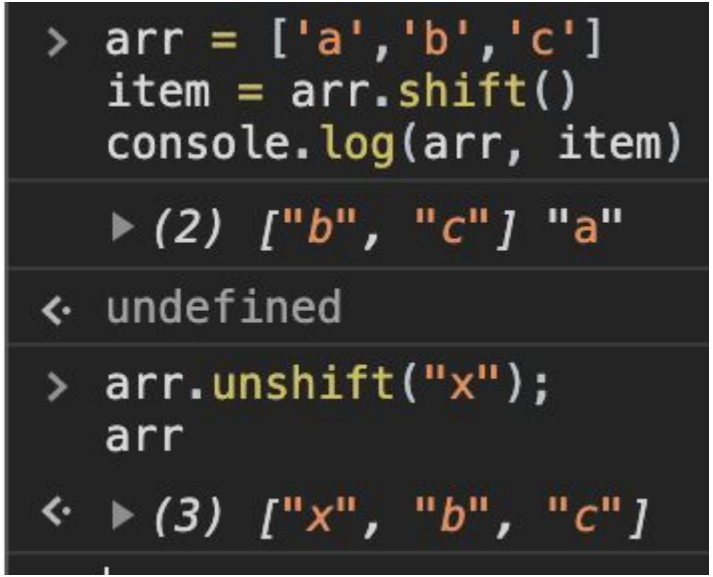
- `push(v)` - inserts item at end of list
- `pop(v)` -removes & returns item from end of list



```
> arr = ['a','b','c']
  item = arr.pop()
  console.log(arr, item)
  ▸ (2) ["a", "b"] "c"
<· undefined
> arr.push("x");
  arr
<· ▸ (3) ["a", "b", "x"]
```

2. Stacks & Queues: Shift & Unshift

- shift(v) - inserts item at start of list
- unshift(v) -removes & returns item from start of list



```
> arr = ['a','b','c']
  item = arr.shift()
  console.log(arr, item)

  ▶ (2) ["b", "c"] "a"

<· undefined

> arr.unshift("x");
  arr

<· ▶ (3) ["x", "b", "c"]
```

2. Stacks & Queues: FIFO vs FILO Operations

- unshift + shift (Stack behavior from start)
- pop + push (Stack behavior from end)
- unshift + pop (Queue behavior from direction: start→ end)
- shift + push (Queue behavior from direction: end→ start)

3. Sets:

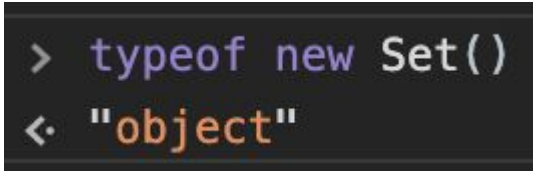
- Sets: In JS
- Sets: Methods Overview
- Sets: Instantiation
- Sets: Properties
- Sets: Accessors (Getters)
- Sets: Mutaters (Setters)
- Sets: Transforms (Set ↔ List)
- Maps: Transforms (Set ↔ String)

3. Sets: In JS

- A set is a Keyed Collection where each item in list is unique.

keys / values	'a'	'b'	'c'	'd'

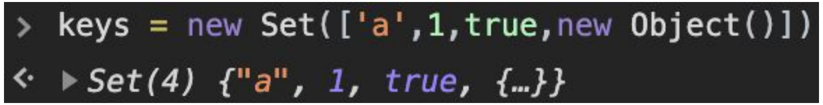
- Sets are objects, with methods to traverse and set/get its values



```
> typeof new Set()
<· "object"
```

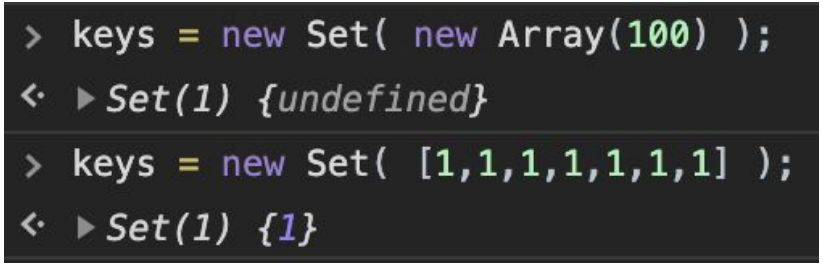
3. Sets: In JS

- The types of its elements are not fixed.



```
> keys = new Set(['a',1,true,new Object()])
<· ▶Set(4) {"a", 1, true, {…}}
```

- Sets are guaranteed to be dense
  & guarantees that each item is unique



```
> keys = new Set( new Array(100) );
<· ▶Set(1) {undefined}
> keys = new Set( [1,1,1,1,1,1,1] );
<· ▶Set(1) {1}
```

3. Sets: Methods Overview

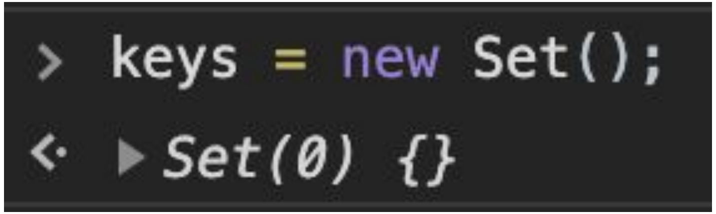
- `Set()` (constructor)

- `size` (accesser)
- `has(v)` (accesser)

- `clear()` (mutater)
- `add(v)` (mutater)
- `delete(v)` (mutater)

3. Sets: Instantiation

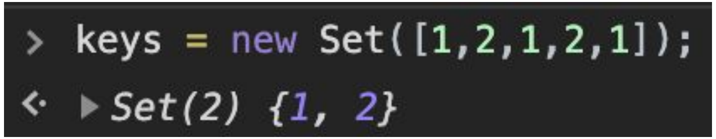
- Constructor *(no parameters)*



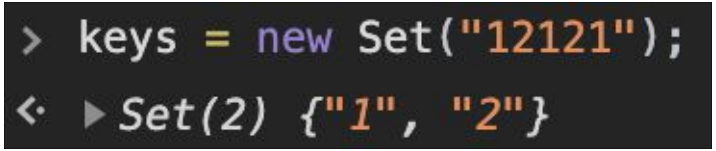
```
> keys = new Set();
<· ▸ Set(0) {}
```

- *Constructor (parameter)*

  param: must be an iterable object



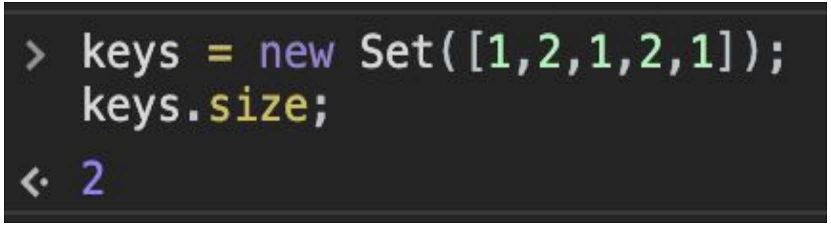
```
> keys = new Set([1,2,1,2,1]);
<· ▸ Set(2) {1, 2}
```



```
> keys = new Set("12121");
<· ▸ Set(2) {"1", "2"}
```

3. Sets: Properties

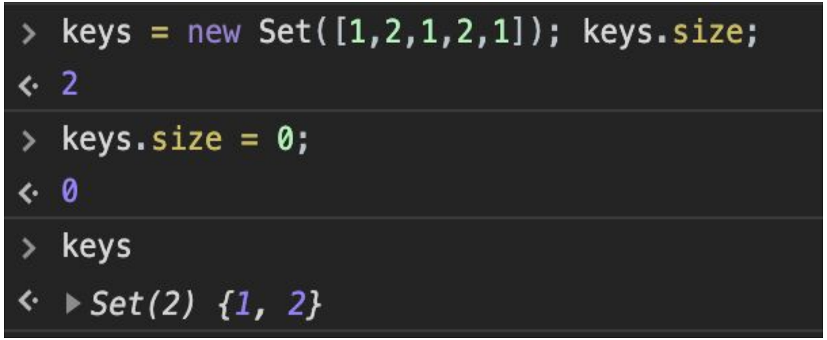
- Size



```
> keys = new Set([1,2,1,2,1]);
  keys.size;
<· 2
```

- The size of a set is not fixed.
  It can grow & shrink,
  but it cannot be reassigned.

  Note: clear() to empty a set.

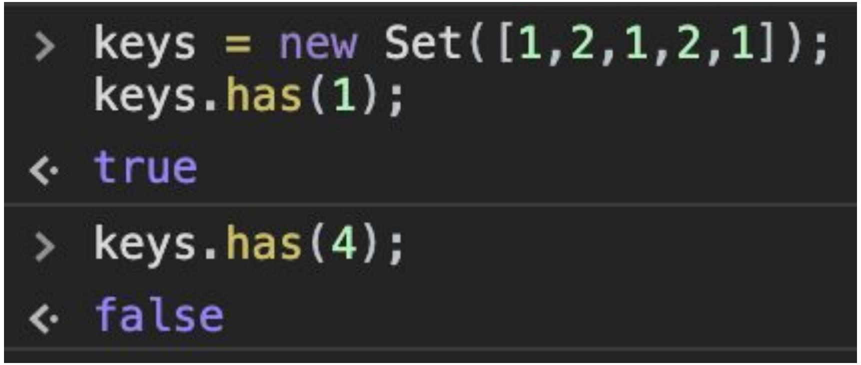


```
> keys = new Set([1,2,1,2,1]); keys.size;
<· 2
> keys.size = 0;
<· 0
> keys
<· ▶Set(2) {1, 2}
```

3. Sets: Accessers (Getters)

- Accessing values from a Set. Sets are not ordered so no indexing into them!

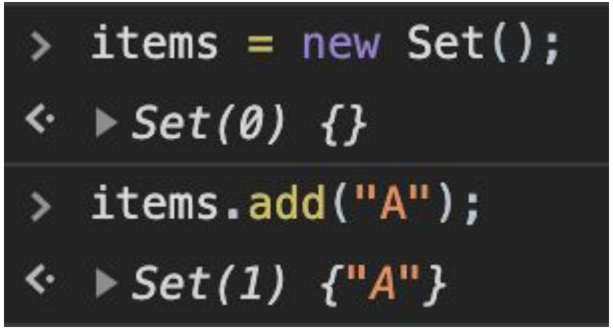
`has(v)`: returns true/false if the set contains that value



```
> keys = new Set([1,2,1,2,1]);
  keys.has(1);
<· true
> keys.has(4);
<· false
```

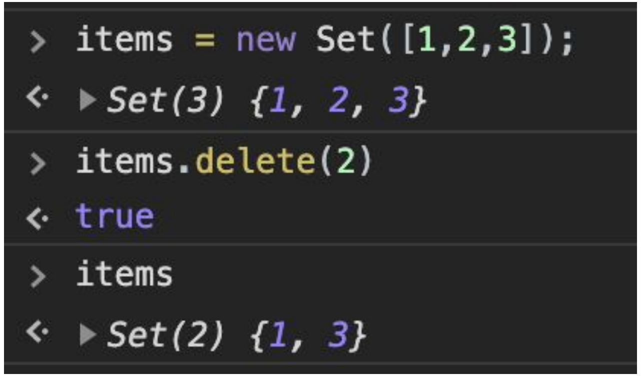
3. Sets: Mutaters (Setters)

- add(v): Add item to a set



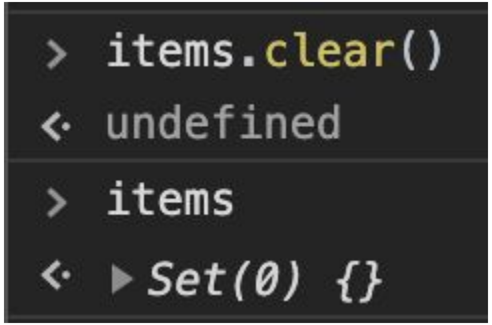
```
> items = new Set();
<· ▸ Set(0) {}
> items.add("A");
<· ▸ Set(1) {"A"}
```

- delete(v): Remove item from a set



```
> items = new Set([1,2,3]);
<· ▸ Set(3) {1, 2, 3}
> items.delete(2)
<· true
> items
<· ▸ Set(2) {1, 3}
```

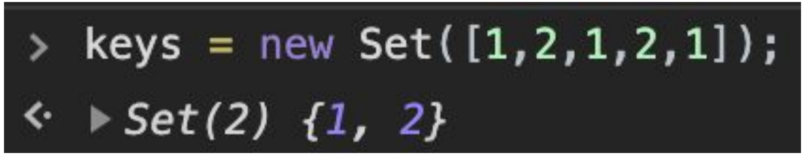
- clear():
  Remove all items from a set



```
> items.clear()
<· undefined
> items
<· ▸ Set(0) {}
```

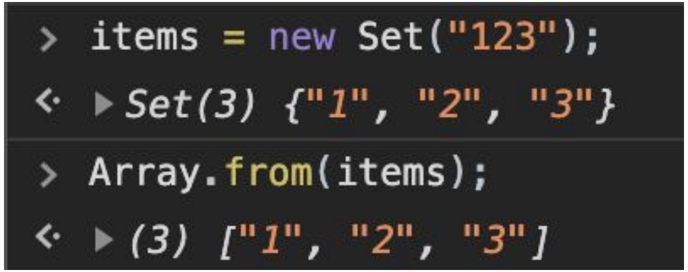
3. Sets: Transform (Set ↔ List)

- Set() constructor transforms (List → Set)



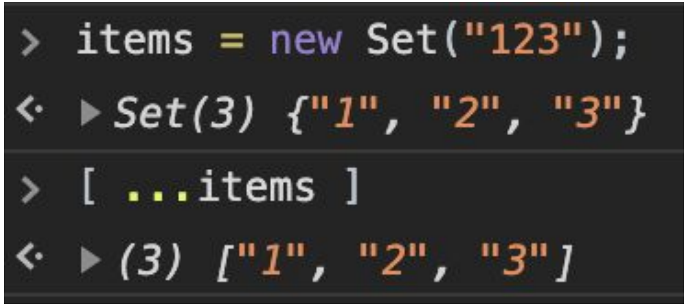
```
> keys = new Set([1,2,1,2,1]);
<· ▸Set(2) {1, 2}
```

- Array.from() function transforms (Set → List)



```
> items = new Set("123");
<· ▸Set(3) {"1", "2", "3"}
> Array.from(items);
<· ▸(3) ["1", "2", "3"]
```

- Spread operator transforms (Set → List)



```
> items = new Set("123");
<· ▸Set(3) {"1", "2", "3"}
> [ ...items ]
<· ▸(3) ["1", "2", "3"]
```

4. Maps:

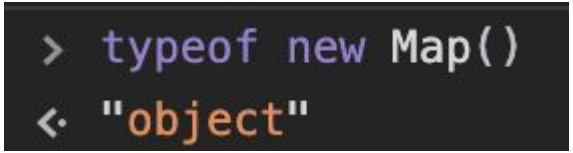
- Maps: In JS
- Maps: Methods Overview
- Maps: Instantiation
- Maps: Properties
- Maps: Accessors (Getters)
- Maps: Mutaters (Setters)
- Maps: Transforms (Map ↔ Array)
- Maps: Transforms (Map ↔ Object)

4. Maps: In JS

- A map is a Keyed Collection where each key is unique & each key pairs to a value

keys	'a'	'b'	'c'	'd'
values	10	true	'cat'	3.5

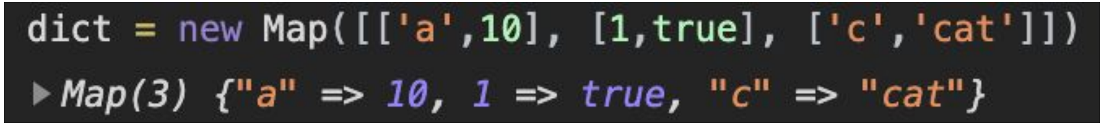
- Maps are objects, with methods to traverse and set/get its keys,values



```
> typeof new Map()
<· "object"
```

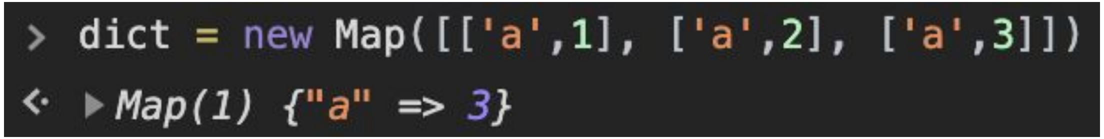
4. Maps: In JS

- The types of its elements are not fixed.



```
dict = new Map([['a',10], [1,true], ['c','cat']])
▶Map(3) {"a" => 10, 1 => true, "c" => "cat"}
```

- Maps are guaranteed to be dense & keys are guaranteed to be unique



```
> dict = new Map([['a',1], ['a',2], ['a',3]])
<· ▶Map(1) {"a" => 3}
```

4. Maps: Methods Overview

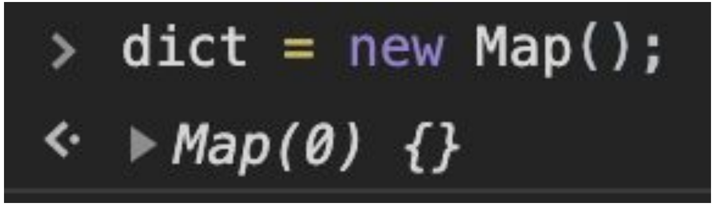
- Map() (constructor)

- size (accessor)
- has(k) (accessor)
- get(k) (accessor)

- set(k,v) (mutater)
- delete(k) (mutater)
- clear() (mutater)

4. Maps: Instantiation

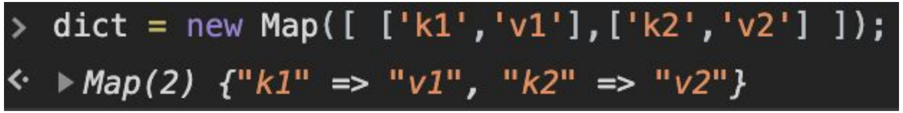
- Constructor *(no parameters)*



```
> dict = new Map();
<· ▶ Map(0) {}
```

- *Constructor (parameter)*

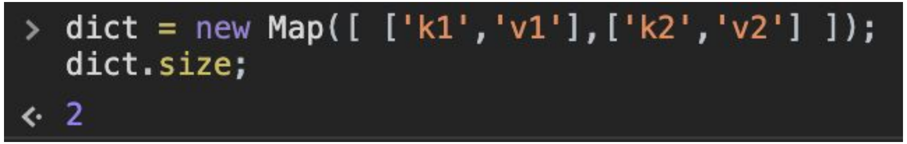
  param: must be an iterable object



```
> dict = new Map([ ['k1','v1'],['k2','v2'] ]);
<· ▶ Map(2) {"k1" => "v1", "k2" => "v2"}
```

4. Maps: Properties

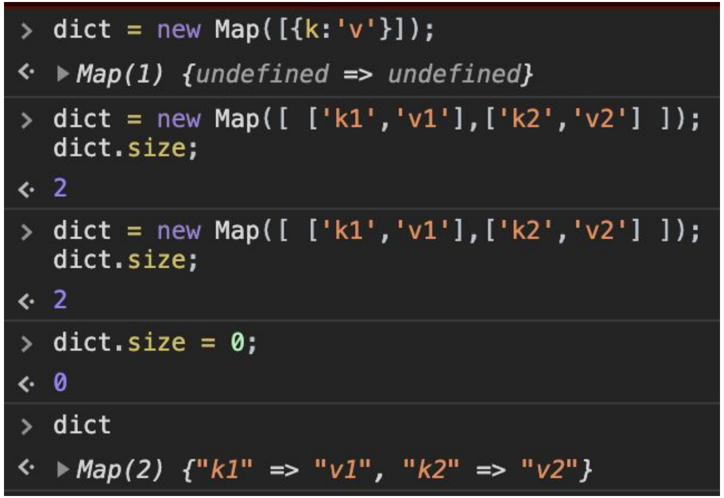
- `size`



```
> dict = new Map([ ['k1','v1'],['k2','v2'] ]);
  dict.size;
<· 2
```

- The size of a map is not fixed.
  It can grow & shrink,
  but it cannot be reassigned.

  Note: clear() to empty a map.

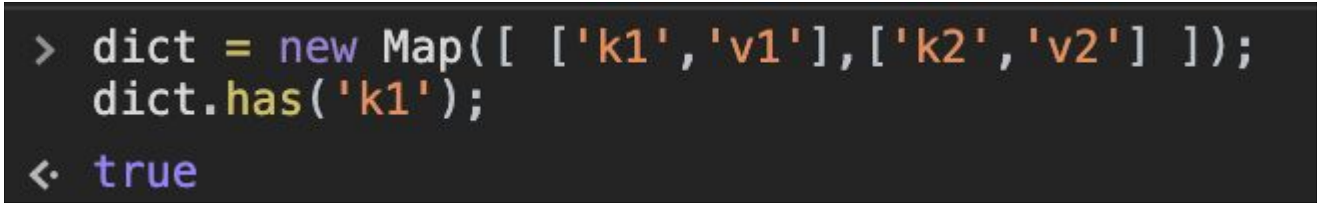


```
> dict = new Map([{k:'v'}]);
<· ▸Map(1) {undefined => undefined}
> dict = new Map([ ['k1','v1'],['k2','v2'] ]);
  dict.size;
<· 2
> dict = new Map([ ['k1','v1'],['k2','v2'] ]);
  dict.size;
<· 2
> dict.size = 0;
<· 0
> dict
<· ▸Map(2) {"k1" => "v1", "k2" => "v2"}
```

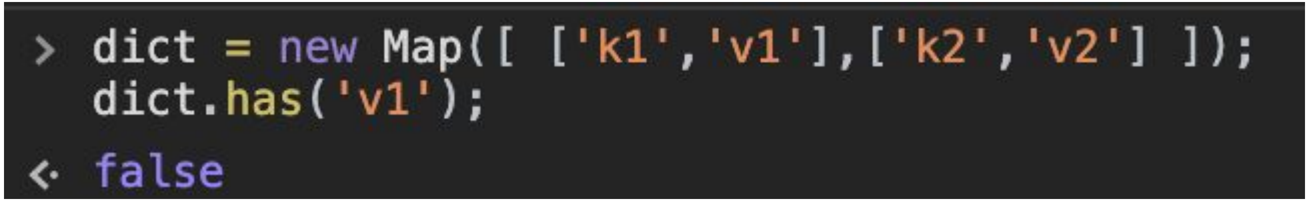
4. Maps: Accessers (Getters)

- Accessing values from a Map. Maps associate a key to each value.

- `has(k)`: returns true/false if the map contains that key



```
> dict = new Map([ ['k1','v1'],['k2','v2'] ]);
  dict.has('k1');
<· true
```

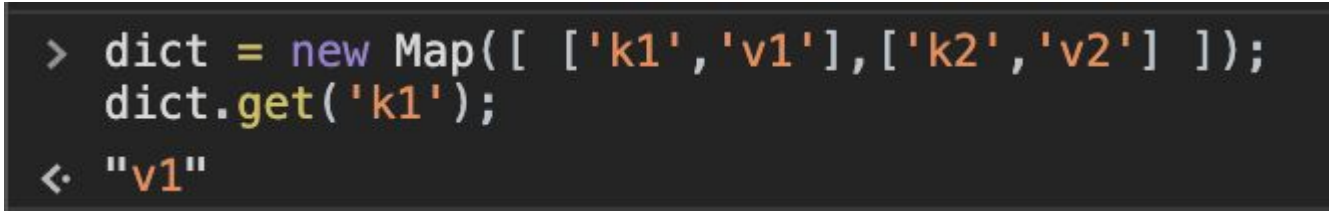


```
> dict = new Map([ ['k1','v1'],['k2','v2'] ]);
  dict.has('v1');
<· false
```

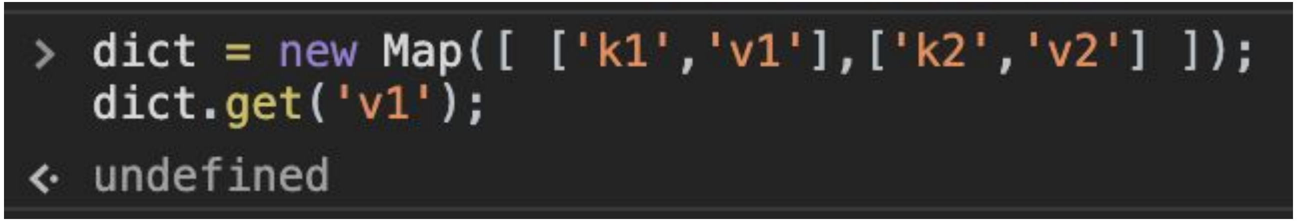
4. Maps: Accessers (Getters)

- Accessing values from a Map. Maps associate a key to each value.

- `get(k)`: returns the value associated with a key



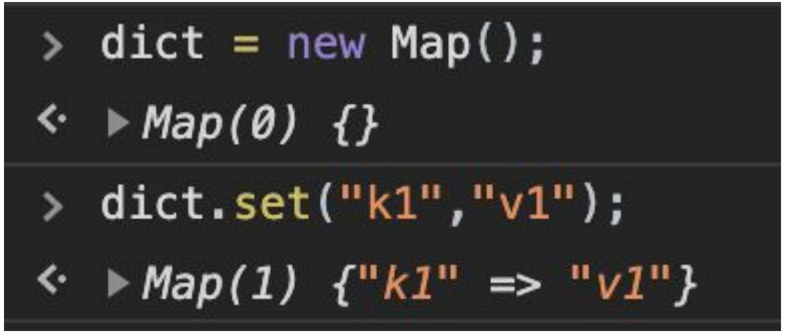
```
> dict = new Map([ ['k1','v1'],['k2','v2'] ]);
  dict.get('k1');
<· "v1"
```



```
> dict = new Map([ ['k1','v1'],['k2','v2'] ]);
  dict.get('v1');
<· undefined
```

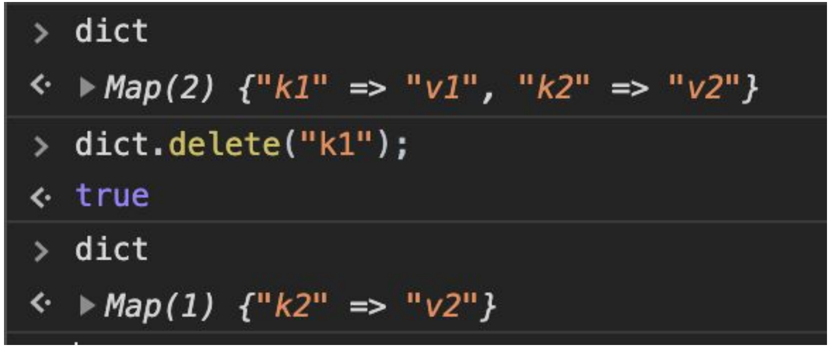
4. Maps: Mutaters (Setters)

- set(k,v): Set a key-value pair



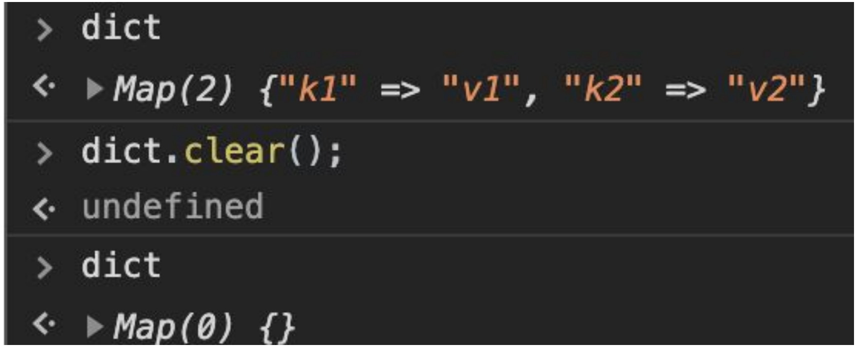
```
> dict = new Map();
<· ▶Map(0) {}
> dict.set("k1","v1");
<· ▶Map(1) {"k1" => "v1"}
```

- delete(k): Remove a key-value pair



```
> dict
<· ▶Map(2) {"k1" => "v1", "k2" => "v2"}
> dict.delete("k1");
<· true
> dict
<· ▶Map(1) {"k2" => "v2"}
```

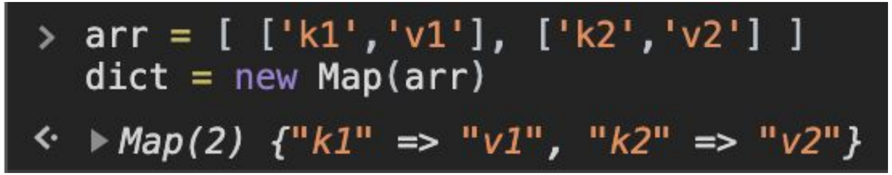
- clear():
  Remove all key-value pairs



```
> dict
<· ▶Map(2) {"k1" => "v1", "k2" => "v2"}
> dict.clear();
<· undefined
> dict
<· ▶Map(0) {}
```

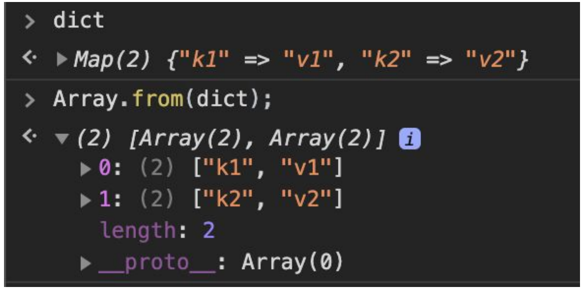
4. Maps: Transform (Map ↔ Array)

- `Map()` constructor
  transforms (Array → Map)



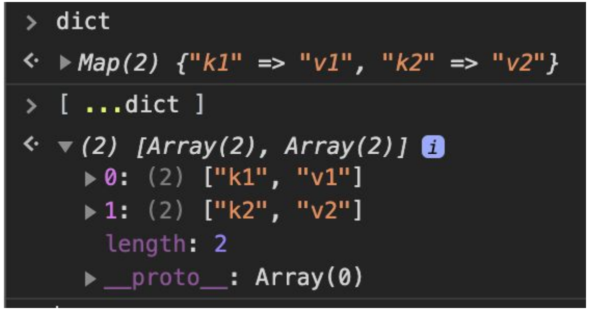
```
> arr = [ ['k1','v1'], ['k2','v2'] ]
  dict = new Map(arr)
<· ▶ Map(2) {"k1" => "v1", "k2" => "v2"}
```

- `Array.from()` function
  transforms (Map → Array)



```
> dict
<· ▶ Map(2) {"k1" => "v1", "k2" => "v2"}
> Array.from(dict);
<· ▼ (2) [Array(2), Array(2)] i
     ▶ 0: (2) ["k1", "v1"]
     ▶ 1: (2) ["k2", "v2"]
       length: 2
     ▶ __proto__: Array(0)
```

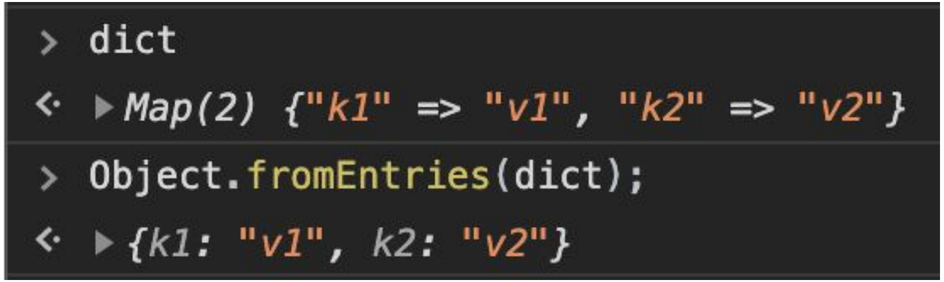
- `...` Spread operator
  transforms (Map → Array)



```
> dict
<· ▶ Map(2) {"k1" => "v1", "k2" => "v2"}
> [ ...dict ]
<· ▼ (2) [Array(2), Array(2)] i
     ▶ 0: (2) ["k1", "v1"]
     ▶ 1: (2) ["k2", "v2"]
       length: 2
     ▶ __proto__: Array(0)
```

4. Maps: Transform (Map ↔ Object)

- `Object.fromEntries()` transforms (Map → Object)



```
> dict
<· ▶Map(2) {"k1" => "v1", "k2" => "v2"}
> Object.fromEntries(dict);
<· ▶{k1: "v1", k2: "v2"}
```

5. Iterator objects

- Iterators: In JS
- Iterators: Methods Overview
- Iterators: next()
- Iterators: Instantiation & Usage

5. Iterator objects: In JS

Explanation:

- Iterator objects iterate across a sequence.
- Iterator objects may return the value at its current position
- Iterator objects track their status within the sequence.

Advantages:

- Multiple iterator objects may be instantiated on a single sequence
- Iterator objects do not mutate the state of the sequence
- All iterators get values with its next() method regardless of underlying data structure

5. Iterators: Methods Overview

- keys() (factory method, from: arrays, sets, maps)
- values() (factory method, from: arrays, sets, maps)
- entries() (factory method, from: arrays, sets, maps)

- next() (accessor/mutator)

Note: A factory method constructs & returns a new instance of an object

5. Iterators: next()

- Iterator objects only have a single method: `next()`

- `next()` method returns an object with two properties:
  - `value:` The current element tracked by the iterator
  - `done:` A boolean value indicating whether there are more items in sequence

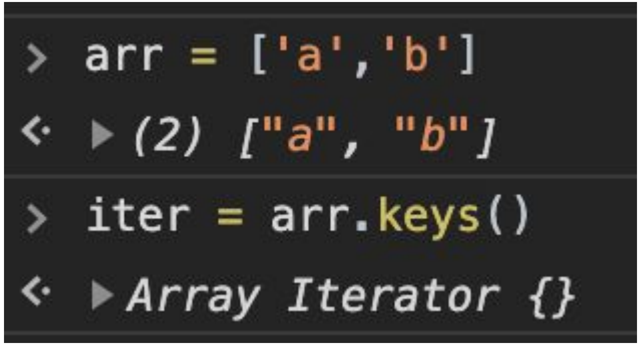
- `next()` is both an Accessor & Mutator method
  - accessor: returns data from the iterator
  - mutator: updates the state of the iterator, *i.e. moves to the next current element in sequence*

- `next()` easily accesses data from sequences
  - avoids concerns for the underlying data structure of sequence
  - avoids concerns for size of sequence (i.e. always returns an object with value & done)

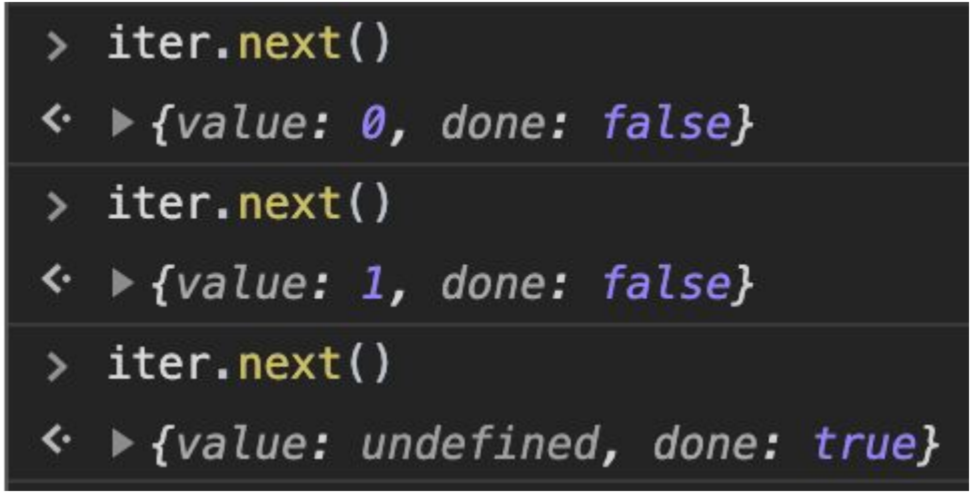
5. Iterators: Instantiation & Usage - keys()

`keys()`: returns an iterator to access the sequence's keys

Array



```
> arr = ['a','b']
<· ▶(2) ["a", "b"]
> iter = arr.keys()
<· ▶Array Iterator {}
```

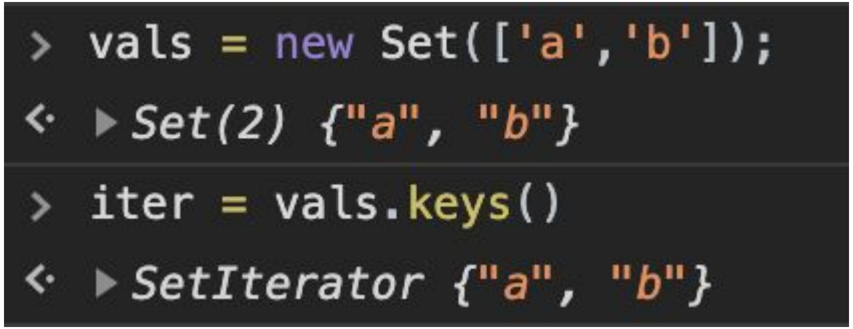


```
> iter.next()
<· ▶{value: 0, done: false}
> iter.next()
<· ▶{value: 1, done: false}
> iter.next()
<· ▶{value: undefined, done: true}
```

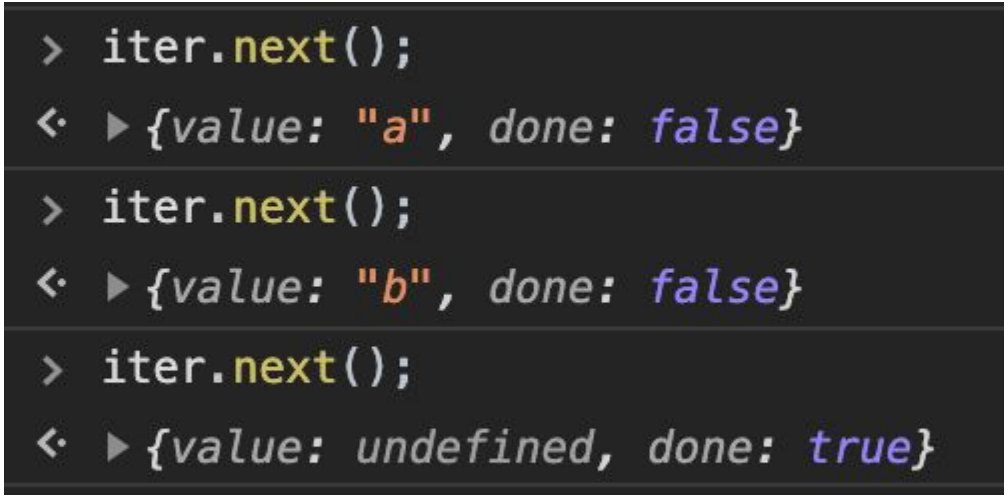
5. Iterators: Instantiation & Usage - keys()

`keys():` returns an iterator to access the sequence's keys

Set



```
> vals = new Set(['a','b']);
<· ▶ Set(2) {"a", "b"}
> iter = vals.keys()
<· ▶ SetIterator {"a", "b"}
```

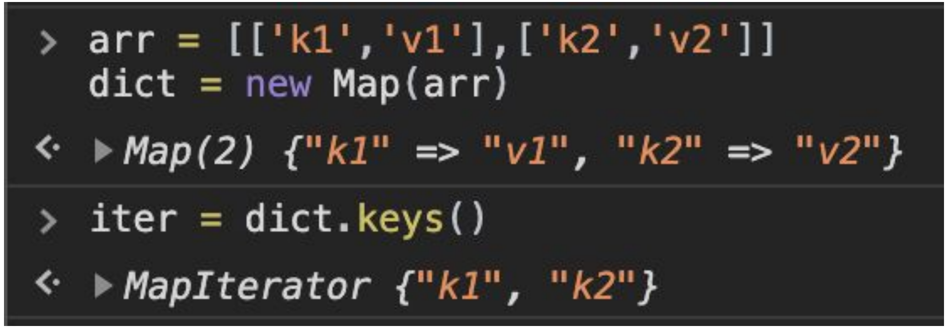


```
> iter.next();
<· ▶ {value: "a", done: false}
> iter.next();
<· ▶ {value: "b", done: false}
> iter.next();
<· ▶ {value: undefined, done: true}
```

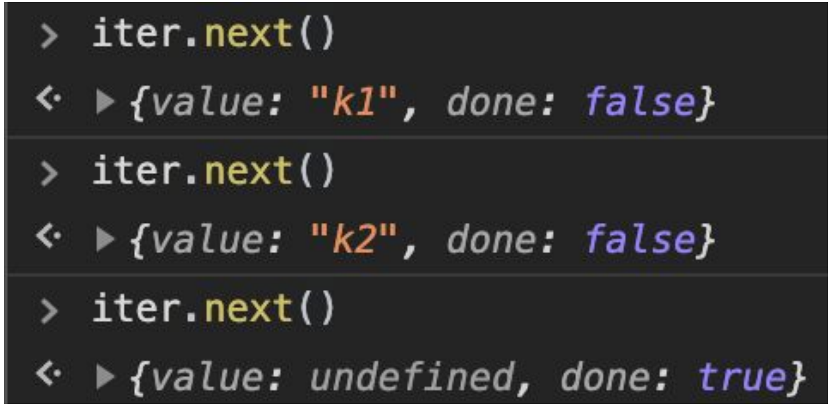
5. Iterators: Instantiation & Usage - keys()

`keys()`: returns an iterator to access the sequence's keys

Map



```
> arr = [['k1','v1'],['k2','v2']]
  dict = new Map(arr)
<· ▶ Map(2) {"k1" => "v1", "k2" => "v2"}
> iter = dict.keys()
<· ▶ MapIterator {"k1", "k2"}
```

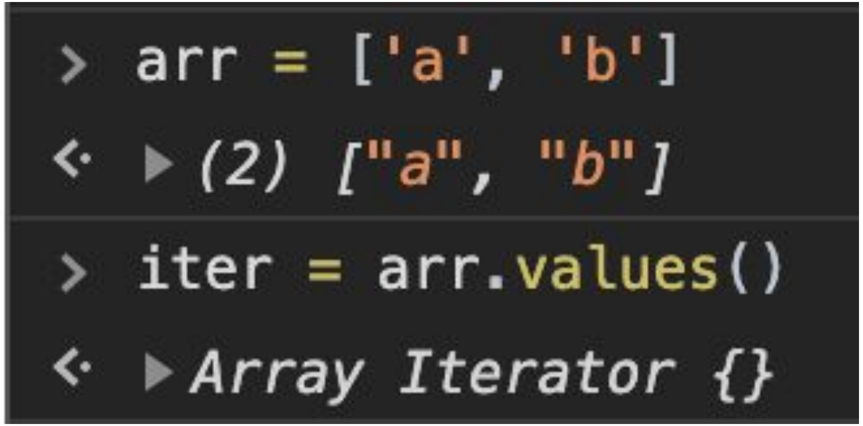


```
> iter.next()
<· ▶ {value: "k1", done: false}
> iter.next()
<· ▶ {value: "k2", done: false}
> iter.next()
<· ▶ {value: undefined, done: true}
```

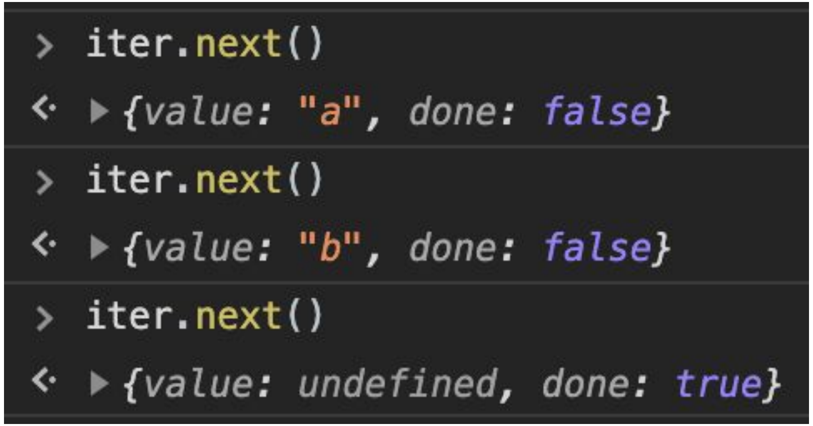
5. Iterators: Instantiation & Usage - values()

`values()`: returns an iterator to access the sequence's values

Array



```
> arr = ['a', 'b']
<· ▶ (2) ["a", "b"]
> iter = arr.values()
<· ▶ Array Iterator {}
```

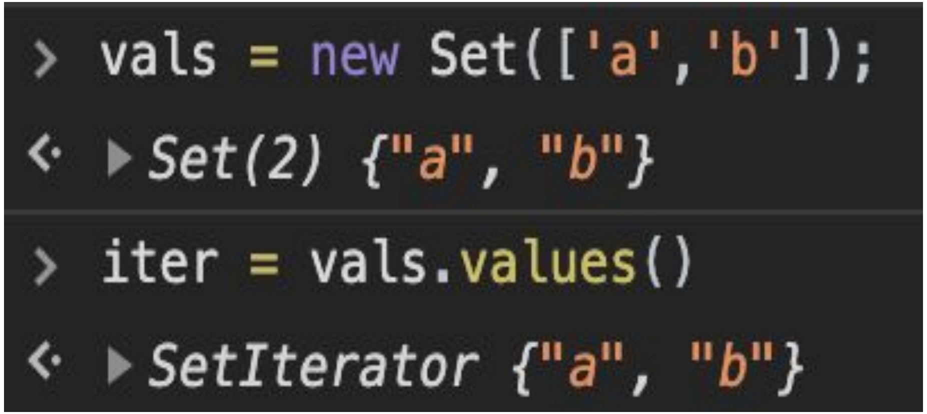


```
> iter.next()
<· ▶ {value: "a", done: false}
> iter.next()
<· ▶ {value: "b", done: false}
> iter.next()
<· ▶ {value: undefined, done: true}
```

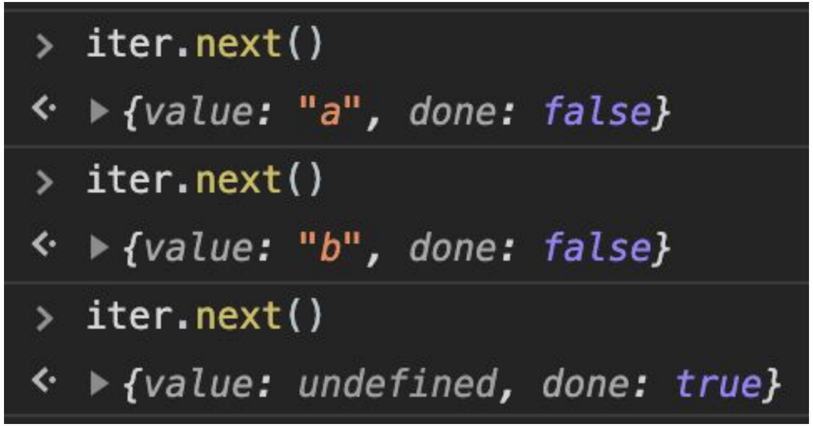
5. Iterators: Instantiation & Usage - values()

`values()`: returns an iterator to access the sequence's values

Set



```
> vals = new Set(['a','b']);
<· ▶ Set(2) {"a", "b"}
> iter = vals.values()
<· ▶ SetIterator {"a", "b"}
```

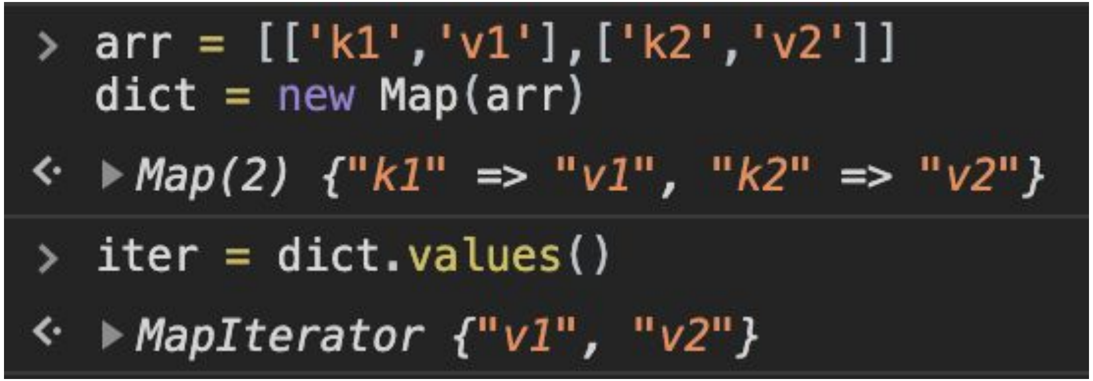


```
> iter.next()
<· ▶ {value: "a", done: false}
> iter.next()
<· ▶ {value: "b", done: false}
> iter.next()
<· ▶ {value: undefined, done: true}
```

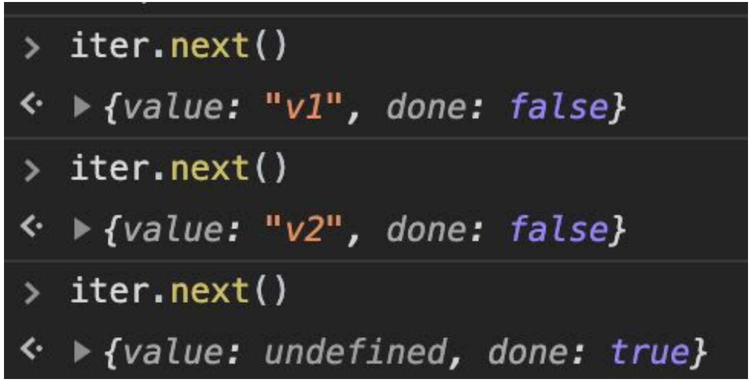
5. Iterators: Instantiation & Usage - values()

`values()`: returns an iterator to access the sequence's values

Map



```
> arr = [['k1','v1'],['k2','v2']]
  dict = new Map(arr)
<· ▶Map(2) {"k1" => "v1", "k2" => "v2"}
> iter = dict.values()
<· ▶MapIterator {"v1", "v2"}
```

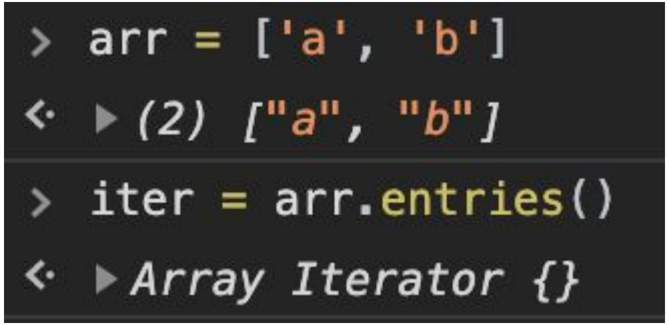


```
> iter.next()
<· ▶{value: "v1", done: false}
> iter.next()
<· ▶{value: "v2", done: false}
> iter.next()
<· ▶{value: undefined, done: true}
```

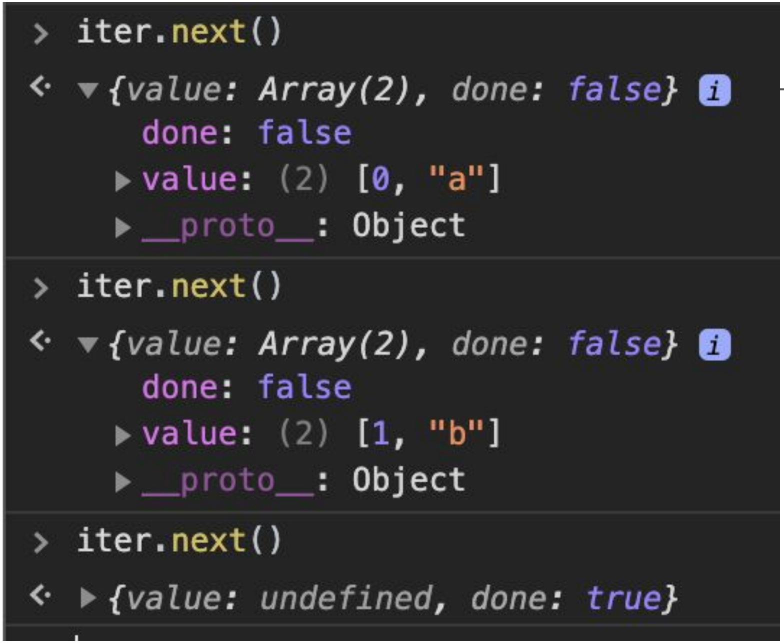
5. Iterators: Instantiation & Usage - entries()

`entries()`: returns an iterator to access the sequence's key-value pairs

Array



```
> arr = ['a', 'b']
<· ▶ (2) ["a", "b"]
> iter = arr.entries()
<· ▶ Array Iterator {}
```

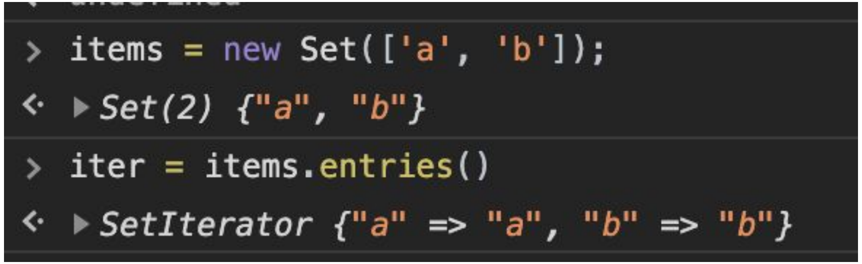


```
> iter.next()
<· ▼ {value: Array(2), done: false} i
       done: false
     ▶ value: (2) [0, "a"]
     ▶ __proto__: Object
> iter.next()
<· ▼ {value: Array(2), done: false} i
       done: false
     ▶ value: (2) [1, "b"]
     ▶ __proto__: Object
> iter.next()
<· ▶ {value: undefined, done: true}
```

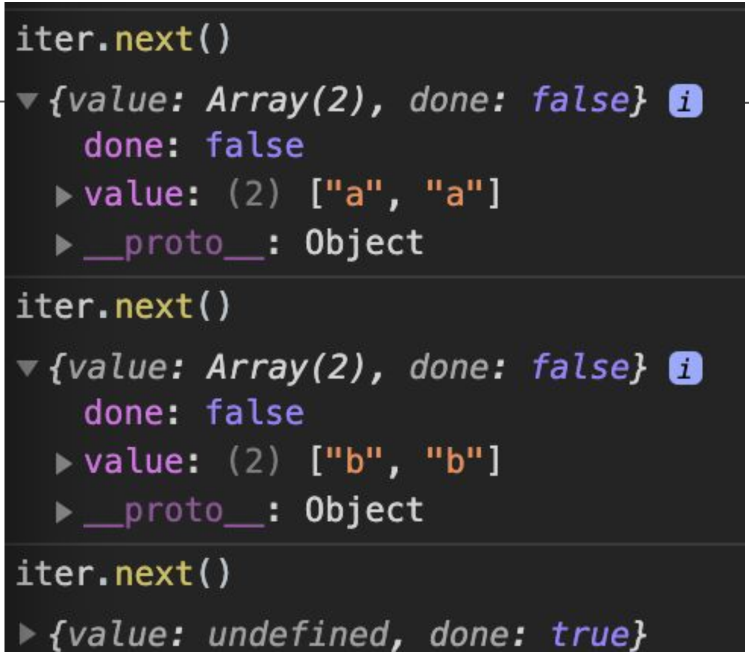
5. Iterators: Instantiation & Usage - entries()

`entries()`: returns an iterator to access the sequence's key-value pairs

Set



```
> items = new Set(['a', 'b']);
<· ▸Set(2) {"a", "b"}
> iter = items.entries()
<· ▸SetIterator {"a" => "a", "b" => "b"}
```

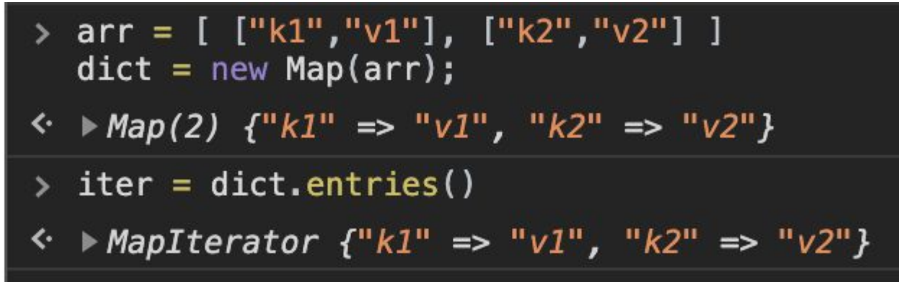


```
iter.next()
▾{value: Array(2), done: false} i
   done: false
  ▸value: (2) ["a", "a"]
  ▸__proto__: Object
iter.next()
▾{value: Array(2), done: false} i
   done: false
  ▸value: (2) ["b", "b"]
  ▸__proto__: Object
iter.next()
▸{value: undefined, done: true}
```

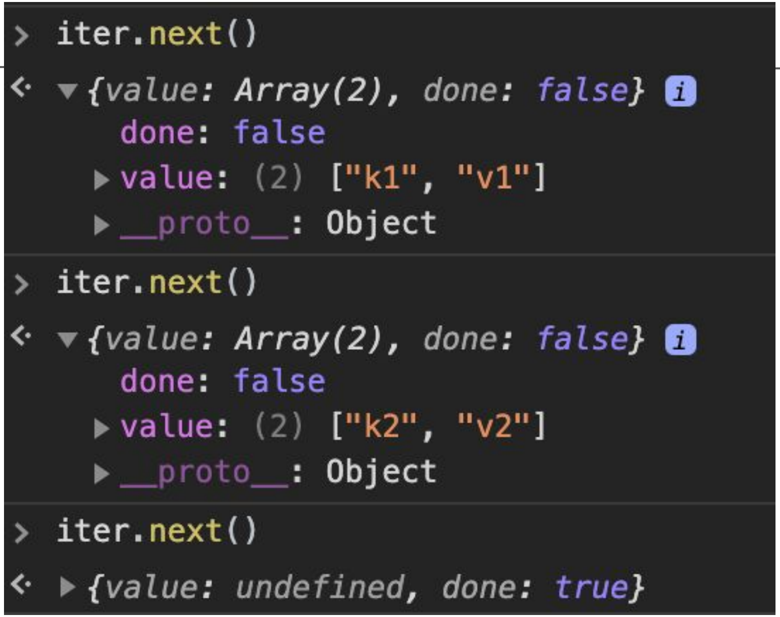
5. Iterators: Instantiation & Usage - entries()

`entries()`: returns an iterator to access the sequence's key-value pairs

Map



```
> arr = [ ["k1","v1"], ["k2","v2"] ]
  dict = new Map(arr);
<· ▶Map(2) {"k1" => "v1", "k2" => "v2"}
> iter = dict.entries()
<· ▶MapIterator {"k1" => "v1", "k2" => "v2"}
```



```
> iter.next()
<· ▼{value: Array(2), done: false} ℹ
      done: false
    ▶ value: (2) ["k1", "v1"]
    ▶ __proto__: Object
> iter.next()
<· ▼{value: Array(2), done: false} ℹ
      done: false
    ▶ value: (2) ["k2", "v2"]
    ▶ __proto__: Object
> iter.next()
<· ▶{value: undefined, done: true}
```

6. Spread operator & Destructuring

- Destructuring Explained
- Spread operator to declare functions
- Spread operator on arrays/objects
- Spread operator to invoke functions
    - Use cases: Math min/max

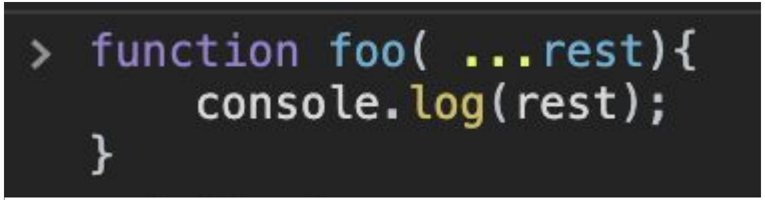
6. Spread operator & Destructuring: Explained

- Destructure assignment: unpack values from arrays, or properties from objects, into distinct variables.

- Spread operator: used when multiple elements from an object or array need to be dereferenced & included together.

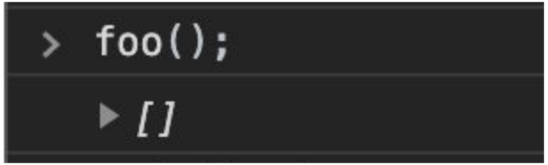
6. Spread operator & Destructuring: Declare Functions

declare function with spread

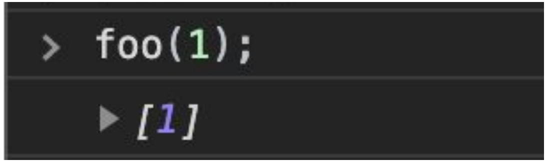


```
> function foo( ...rest){
      console.log(rest);
  }
```

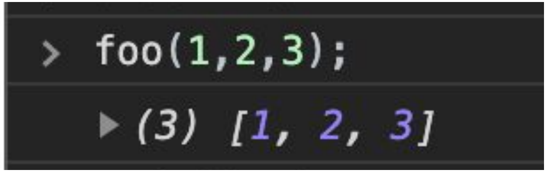
invoke function with varying length params



```
> foo();
  ▶ []
```



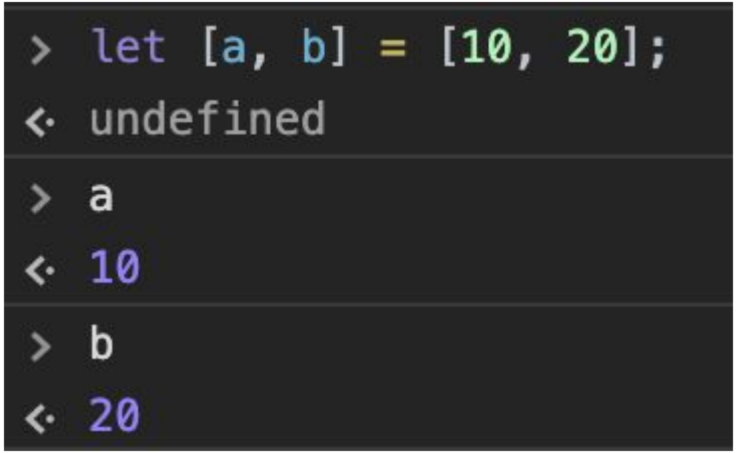
```
> foo(1);
  ▶ [1]
```



```
> foo(1,2,3);
  ▶ (3) [1, 2, 3]
```

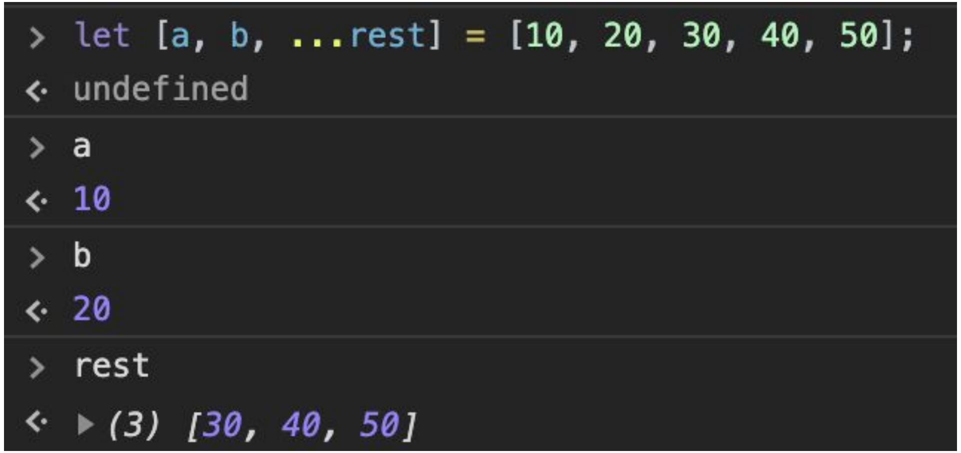
6. Spread operator & Destructuring: Arrays/Objects

Array Destructuring



```
> let [a, b] = [10, 20];
<· undefined
> a
<· 10
> b
<· 20
```

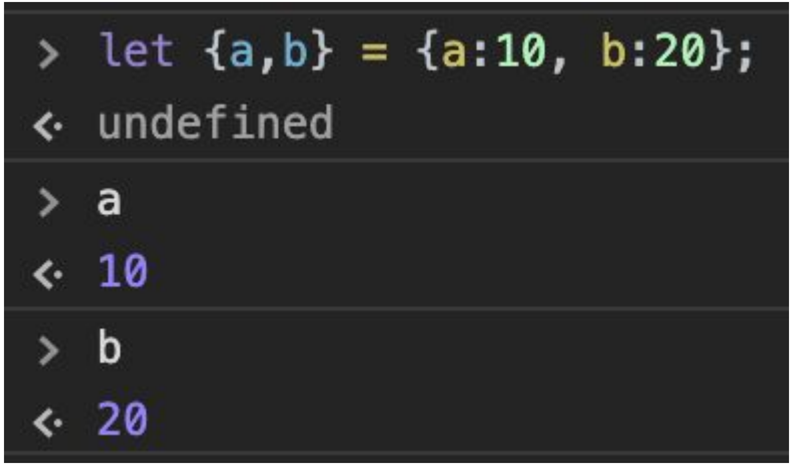
Array Destructuring & spread operator *(left)*



```
> let [a, b, ...rest] = [10, 20, 30, 40, 50];
<· undefined
> a
<· 10
> b
<· 20
> rest
<· ▶(3) [30, 40, 50]
```

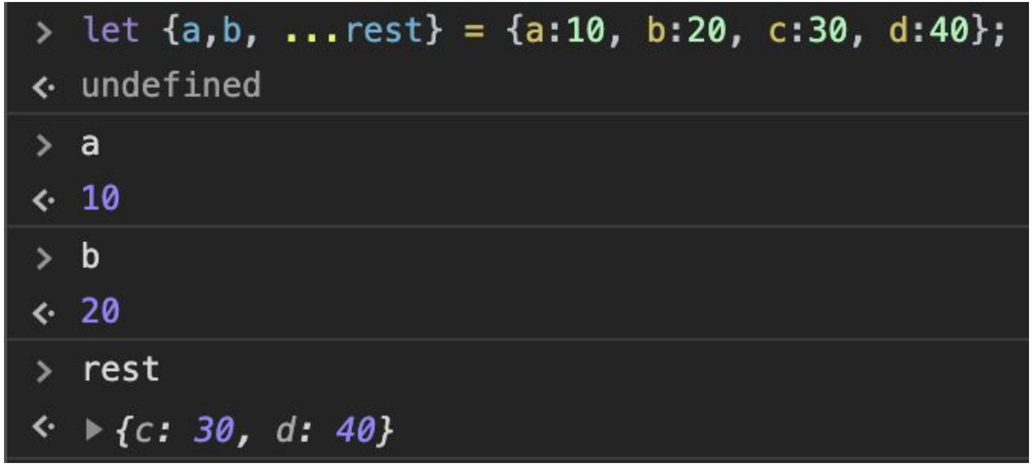
6. Spread operator & Destructuring: Arrays/Objects

Object Destructuring



```
> let {a,b} = {a:10, b:20};
<· undefined
> a
<· 10
> b
<· 20
```

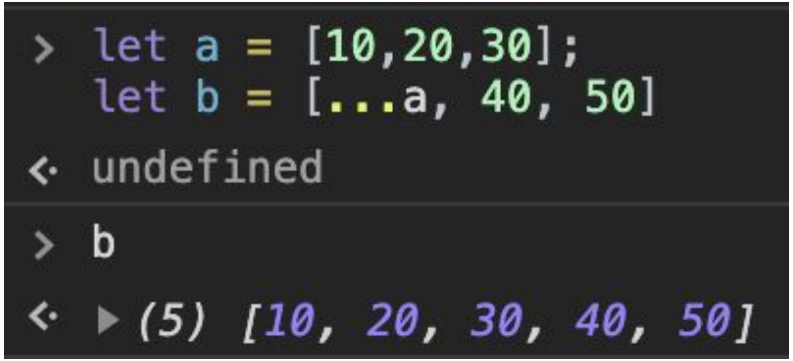
Object Destructuring & spread operator *(left)*



```
> let {a,b, ...rest} = {a:10, b:20, c:30, d:40};
<· undefined
> a
<· 10
> b
<· 20
> rest
<· ▶{c: 30, d: 40}
```

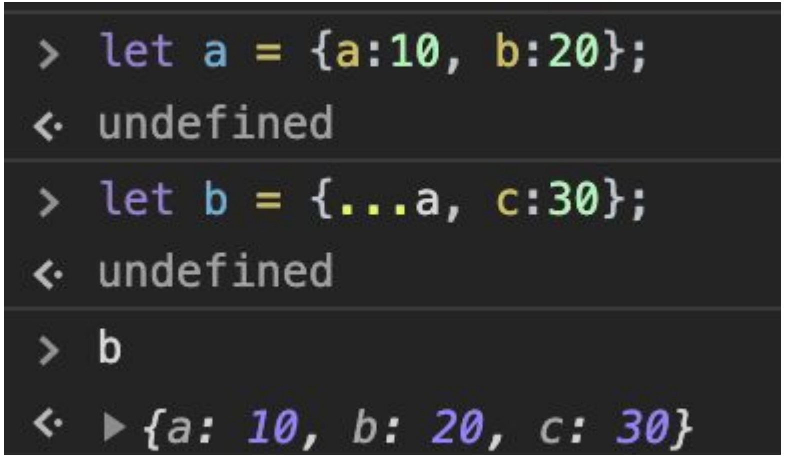
6. Spread operator & Destructuring: Arrays/Objects

Array Destructuring *(right)*



```
> let a = [10,20,30];
  let b = [...a, 40, 50]
<· undefined
> b
<· ▶ (5) [10, 20, 30, 40, 50]
```

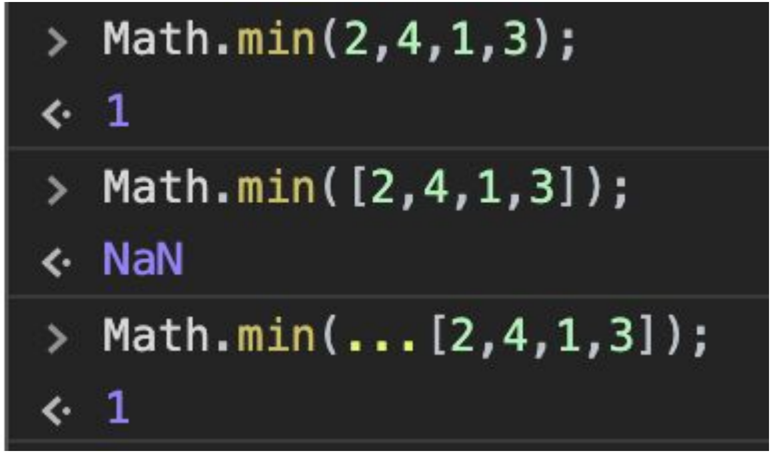
Object Destructuring *(right)*



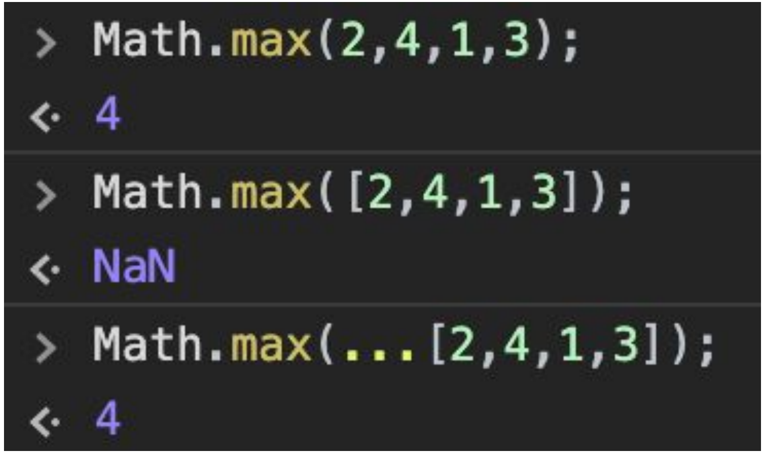
```
> let a = {a:10, b:20};
<· undefined
> let b = {...a, c:30};
<· undefined
> b
<· ▶ {a: 10, b: 20, c: 30}
```

6. Spread operator & Destructuring: Invoke Functions

- Use Case: Get Min/Max values from array with Math object



```
> Math.min(2,4,1,3);
<· 1
> Math.min([2,4,1,3]);
<· NaN
> Math.min(...[2,4,1,3]);
<· 1
```



```
> Math.max(2,4,1,3);
<· 4
> Math.max([2,4,1,3]);
<· NaN
> Math.max(...[2,4,1,3]);
<· 4
```

7. Traversal methods - Arrays - Overview

- Searchers
  - indexOf
  - lastIndexOf
  - includes
  - find
  - findIndex
- Iterators
  - forEach
  - map
  - flatMap
- Sorters
  - sort
  - reverse
- Simplifiers
  - flat
  - filter
  - reduce
  - reduceRight
- Existential Quantifiers
  - every
  - some

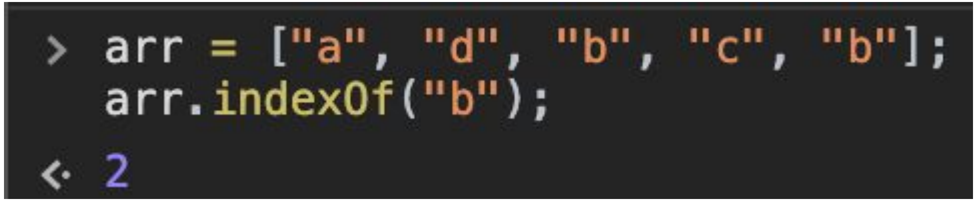
7. Array methods: Searchers - Overview

First Order Functions *(values as parameters)*	High Order Functions *(functions as parameters)*

`indexOf(v)`
`lastIndexOf(v)`
`includes(v)`
	`find( fx→x )`
	`findIndex( fx→x )`

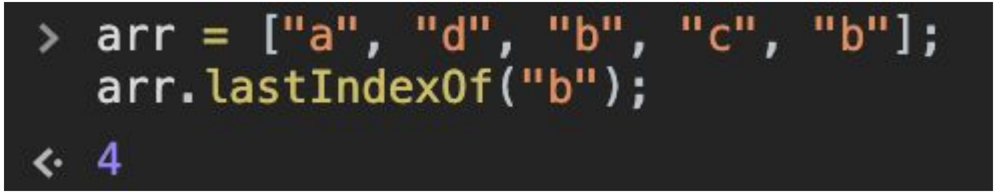
7. Array methods: Searchers - Examples

`indexOf(v)`
returns first index of item in array



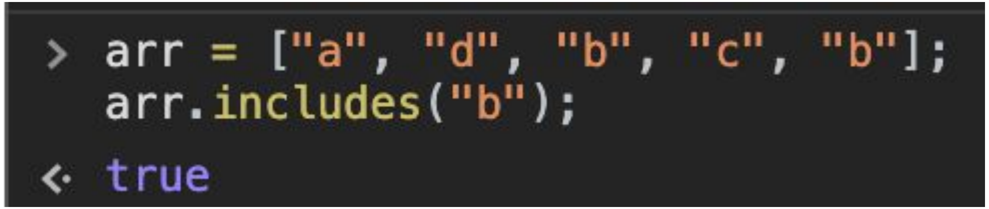
```
> arr = ["a", "d", "b", "c", "b"];
  arr.indexOf("b");
<· 2
```

`lastIndexOf(v)`
returns last index of item in array



```
> arr = ["a", "d", "b", "c", "b"];
  arr.lastIndexOf("b");
<· 4
```

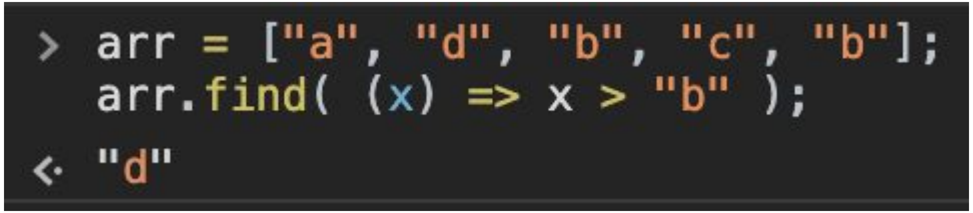
`includes(v)`
returns boolean if item is in array



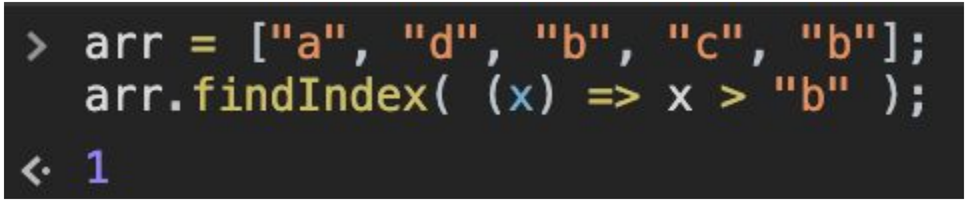
```
> arr = ["a", "d", "b", "c", "b"];
  arr.includes("b");
<· true
```

7. Array methods: Searchers - Examples

Anonymous function as parameter

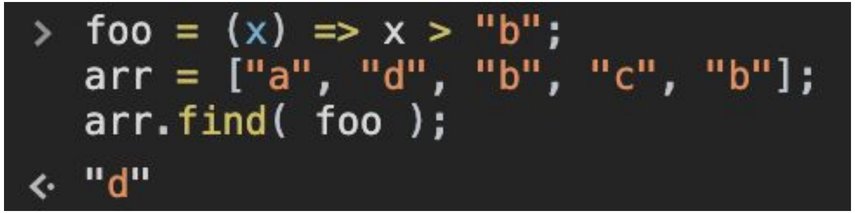


```
> arr = ["a", "d", "b", "c", "b"];
  arr.find( (x) => x > "b" );
<· "d"
```

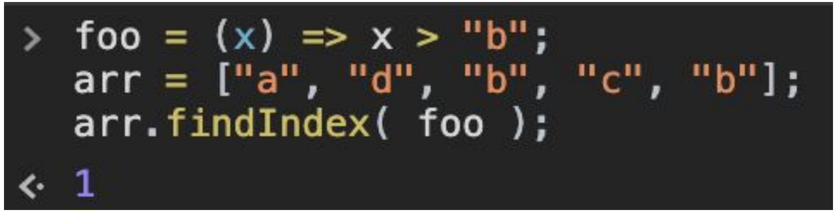


```
> arr = ["a", "d", "b", "c", "b"];
  arr.findIndex( (x) => x > "b" );
<· 1
```

Named function as parameter



```
> foo = (x) => x > "b";
  arr = ["a", "d", "b", "c", "b"];
  arr.find( foo );
<· "d"
```



```
> foo = (x) => x > "b";
  arr = ["a", "d", "b", "c", "b"];
  arr.findIndex( foo );
<· 1
```

`find(fx→x)`
returns first item that meets condition from given function

`findIndex(fx→x)`
returns first index of item that meets condition from given function

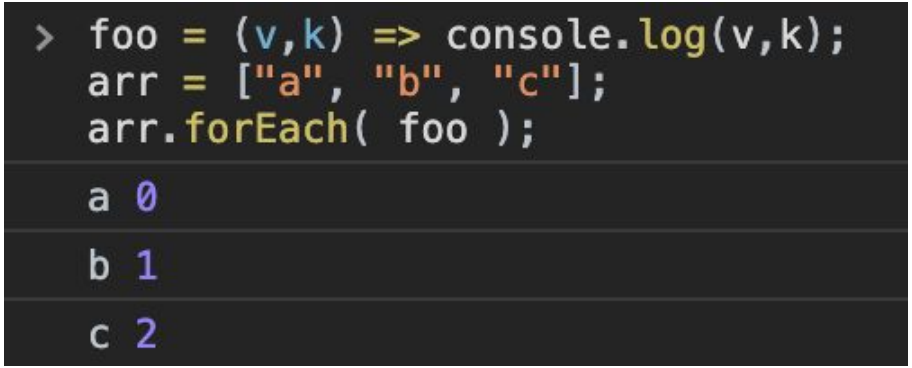
7. Array methods: Iterators - Overall

First Order Functions *(values as parameters)*	High Order Functions *(functions as parameters)*

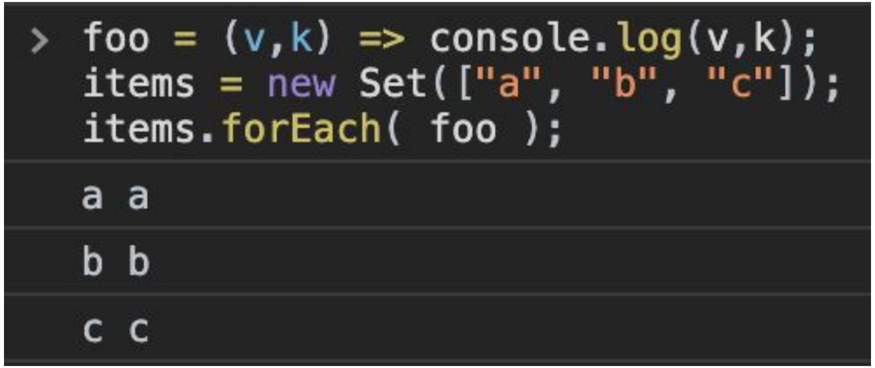
`None`	`forEach( fx→x )`
	`map( fx→x )`
	`flatMap( fx→x )`

7. Array methods: Iterators - Examples

`forEach(fx→x)` works on Arrays, Sets, & Maps, *Note: method has no return type*

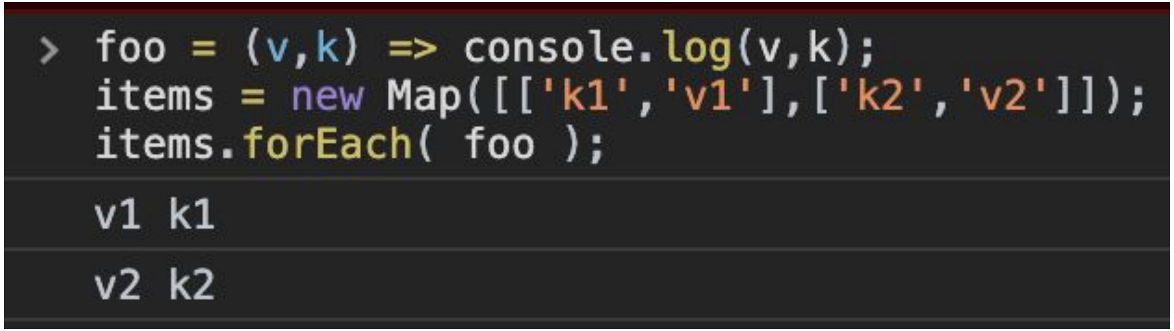


```
> foo = (v,k) => console.log(v,k);
  arr = ["a", "b", "c"];
  arr.forEach( foo );
  a 0
  b 1
  c 2
```



```
> foo = (v,k) => console.log(v,k);
  items = new Set(["a", "b", "c"]);
  items.forEach( foo );
  a a
  b b
  c c
```

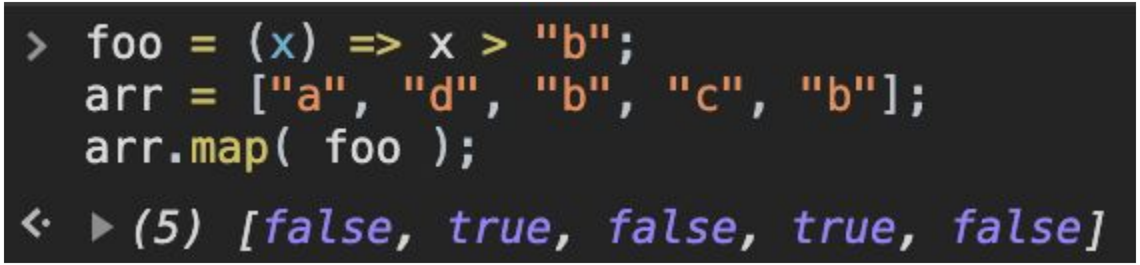
`forEach( fx→x )`
executes the given function on each item in the array, set, map



```
> foo = (v,k) => console.log(v,k);
  items = new Map([['k1','v1'],['k2','v2']]);
  items.forEach( foo );
  v1 k1
  v2 k2
```

7. Array methods: Iterators - Examples

`map( fx→x )` returns a new array with the returned results from the given function

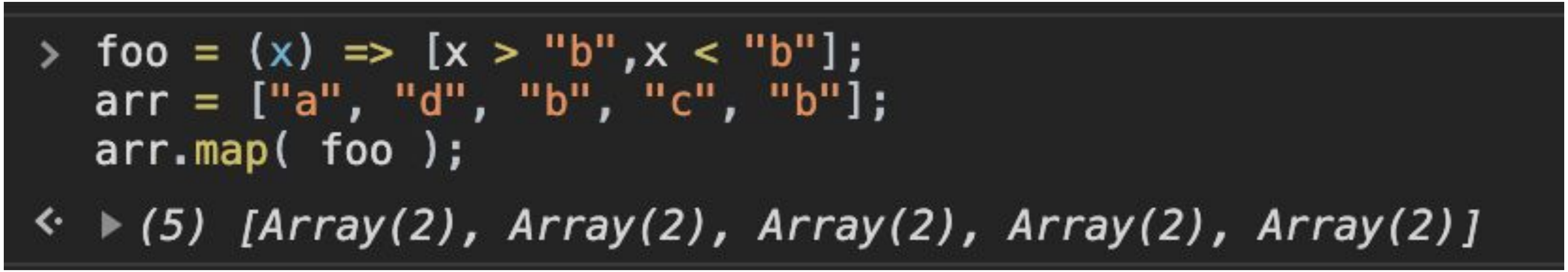


```
> foo = (x) => x > "b";
  arr = ["a", "d", "b", "c", "b"];
  arr.map( foo );
<· ▶ (5) [false, true, false, true, false]
```

7. Array methods: Iterators - Examples

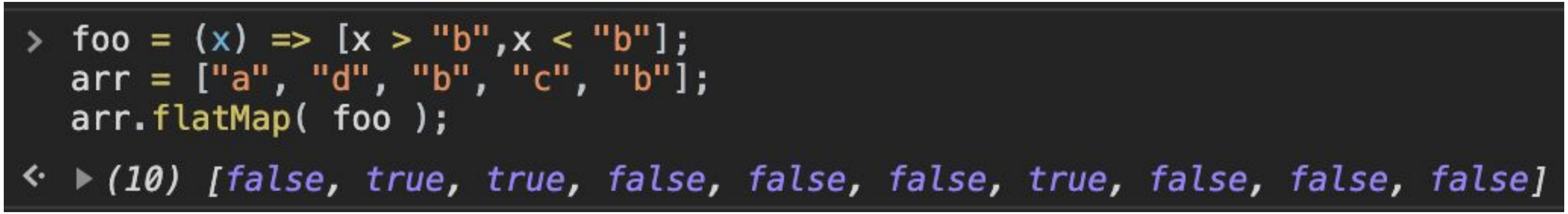
`flatMap( fx→x )` similar to map, but also flattens arrays returned from the given function

map



```
> foo = (x) => [x > "b",x < "b"];
  arr = ["a", "d", "b", "c", "b"];
  arr.map( foo );
<· ▶ (5) [Array(2), Array(2), Array(2), Array(2), Array(2)]
```

flatMap



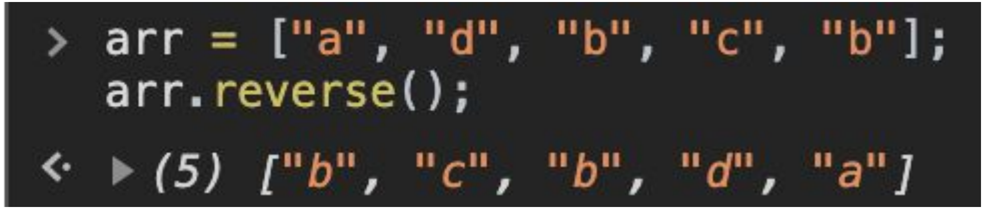
```
> foo = (x) => [x > "b",x < "b"];
  arr = ["a", "d", "b", "c", "b"];
  arr.flatMap( foo );
<· ▶ (10) [false, true, true, false, false, false, true, false, false, false]
```

7. Array methods: Sorters - Overall

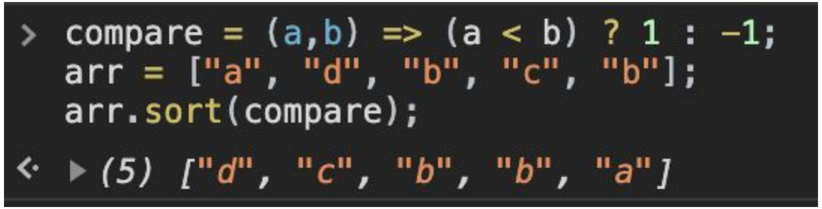
First Order Functions *(values as parameters)*	High Order Functions *(functions as parameters)*
`reverse()`	`sort( fx→x )`

7. Array methods: Sorters - Examples

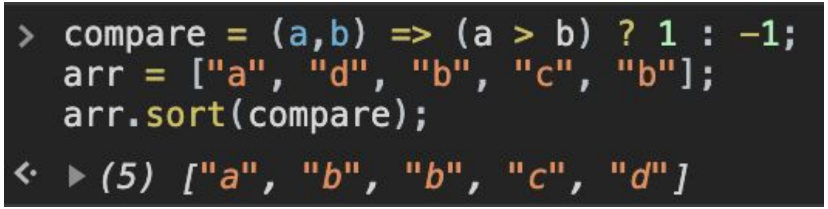
`reverse()`
reverses the order of the items in the array



```
> arr = ["a", "d", "b", "c", "b"];
  arr.reverse();
<· ▶ (5) ["b", "c", "b", "d", "a"]
```



```
> compare = (a,b) => (a < b) ? 1 : -1;
  arr = ["a", "d", "b", "c", "b"];
  arr.sort(compare);
<· ▶ (5) ["d", "c", "b", "b", "a"]
```



```
> compare = (a,b) => (a > b) ? 1 : -1;
  arr = ["a", "d", "b", "c", "b"];
  arr.sort(compare);
<· ▶ (5) ["a", "b", "b", "c", "d"]
```

`sort(fx→x)`
reorders items in array based on a given compare function. Compare function returns -1, 0, 1

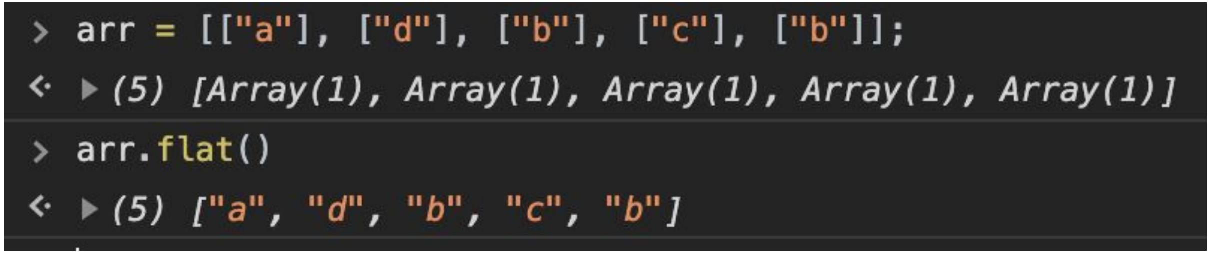
7. Array methods: Simplifiers - Overall

First Order Functions *(values as parameters)*	High Order Functions *(functions as parameters)*

`flat()`	`filter( fx→x )`
	`reduce( fx→x )`
	`reduceRight( fx→x )`

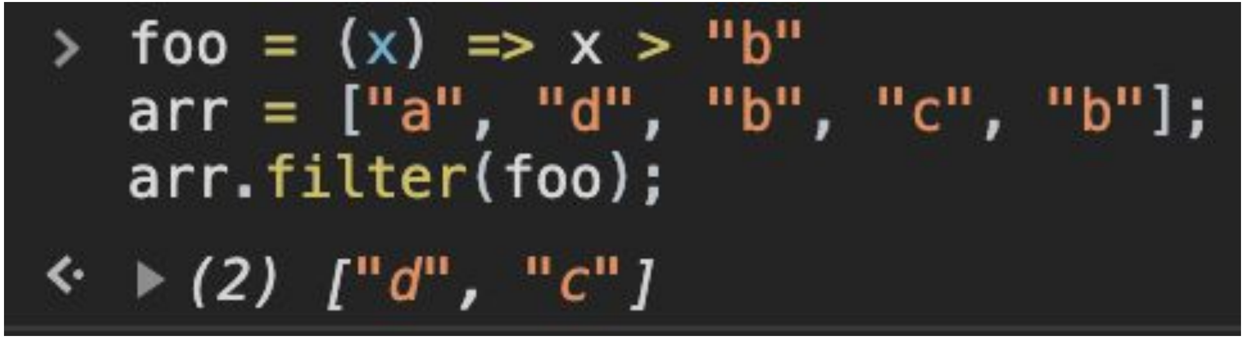
7. Array methods: Simplifiers - Examples

`flat()`:
returns new array that flattens all inner array items



```
> arr = [["a"], ["d"], ["b"], ["c"], ["b"]];
<· ▶ (5) [Array(1), Array(1), Array(1), Array(1), Array(1)]
> arr.flat()
<· ▶ (5) ["a", "d", "b", "c", "b"]
```

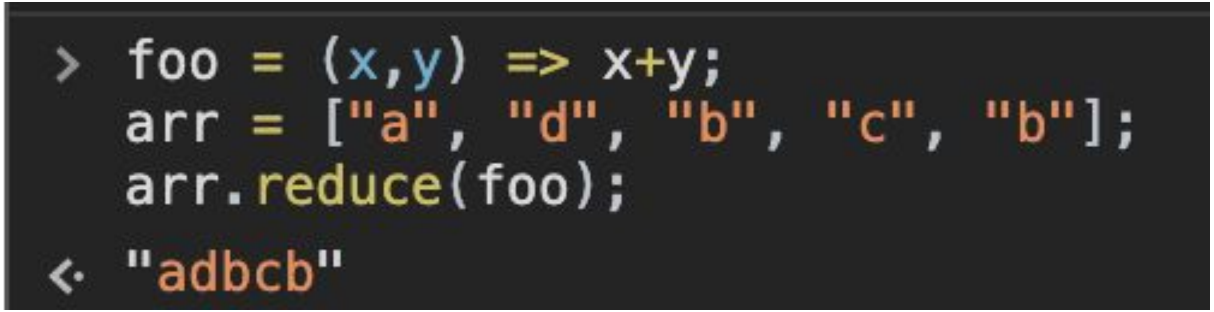
`filter(fx→x)`:
returns new array containing all items that meet condition



```
> foo = (x) => x > "b"
  arr = ["a", "d", "b", "c", "b"];
  arr.filter(foo);
<· ▶ (2) ["d", "c"]
```

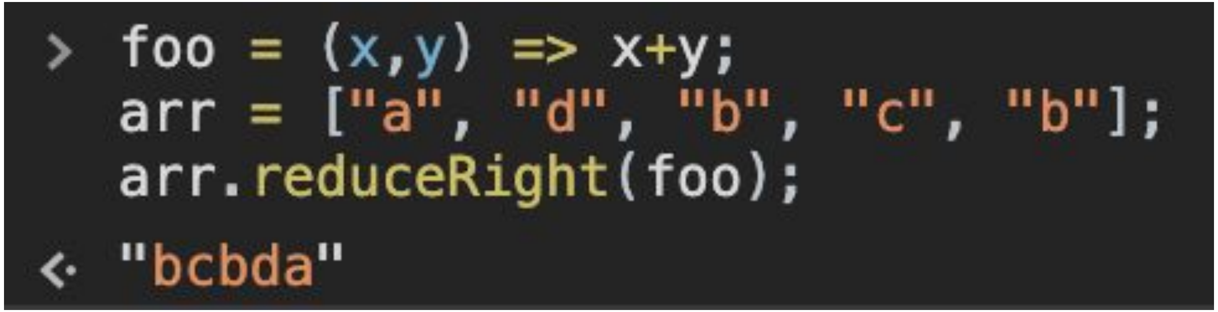
7. Array methods: Simplifiers - Examples

`reduce(fx→x):`
returns a value from computation on all items from left-to-right order



```
> foo = (x,y) => x+y;
  arr = ["a", "d", "b", "c", "b"];
  arr.reduce(foo);
<· "adbcb"
```

`reduceRight(fx→x):`
returns a value from computation on all items from right-to-left order



```
> foo = (x,y) => x+y;
  arr = ["a", "d", "b", "c", "b"];
  arr.reduceRight(foo);
<· "bcbda"
```

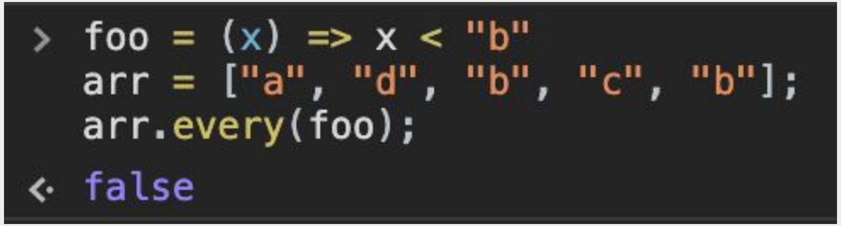
7. Array methods: Existential Quantifiers - Overall

First Order Functions *(values as parameters)*	High Order Functions *(functions as parameters)*

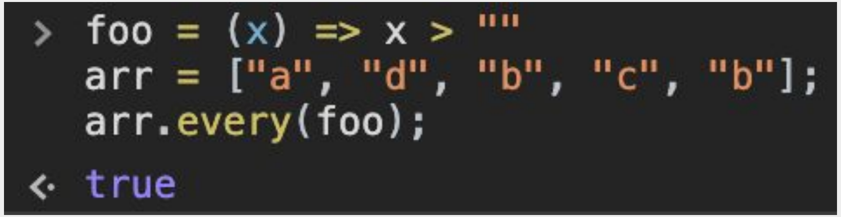
`None`	`every( fx⟶x )`
	`some( fx⟶x )`

7. Array methods: Existential Quantifiers - Examples

`every(fx→x):`
boolean result whether all items meet condition from the given function

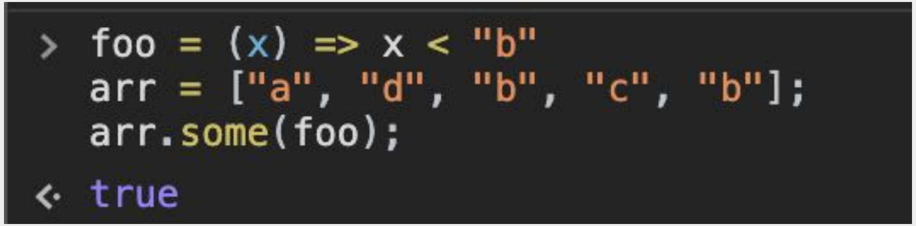


```
> foo = (x) => x < "b"
  arr = ["a", "d", "b", "c", "b"];
  arr.every(foo);
<· false
```

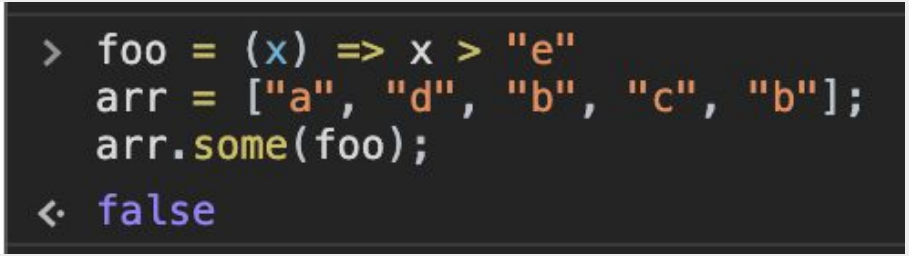


```
> foo = (x) => x > ""
  arr = ["a", "d", "b", "c", "b"];
  arr.every(foo);
<· true
```

`some(fx→x):`
boolean result whether at least one item meets condition from the given function



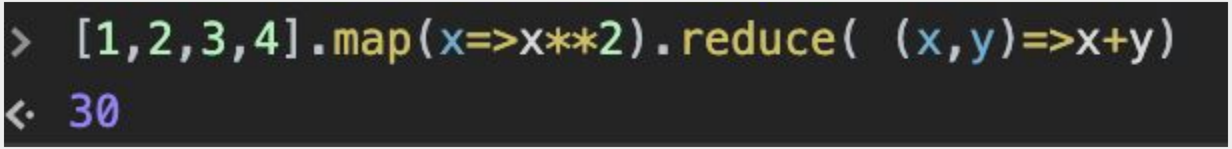
```
> foo = (x) => x < "b"
  arr = ["a", "d", "b", "c", "b"];
  arr.some(foo);
<· true
```



```
> foo = (x) => x > "e"
  arr = ["a", "d", "b", "c", "b"];
  arr.some(foo);
<· false
```

7. Array methods: Cascading Calls

A final note to these powerful built-in Array methods. They may be cascaded one into another to compute a result. This approach of chaining function calls is the basis for the **functional programming** paradigm



```
> [1,2,3,4].map(x=>x**2).reduce( (x,y)=>x+y)
<· 30
```

Explanation:

Step 1: Start with array [1,2,3,4]

Step 2: Map an anonymous function that returns a new array where each element is squared [1,4,9,16]

Step 3: Reduce the new array into a single value whereby all the elements are added together: 30

The End.

Coming Soon... to a Moodle near you!

Lab 5	'JavaScript - Data Structures'
Homework 5	Data Visualizations