Collections

Przemysław Maćkowiak

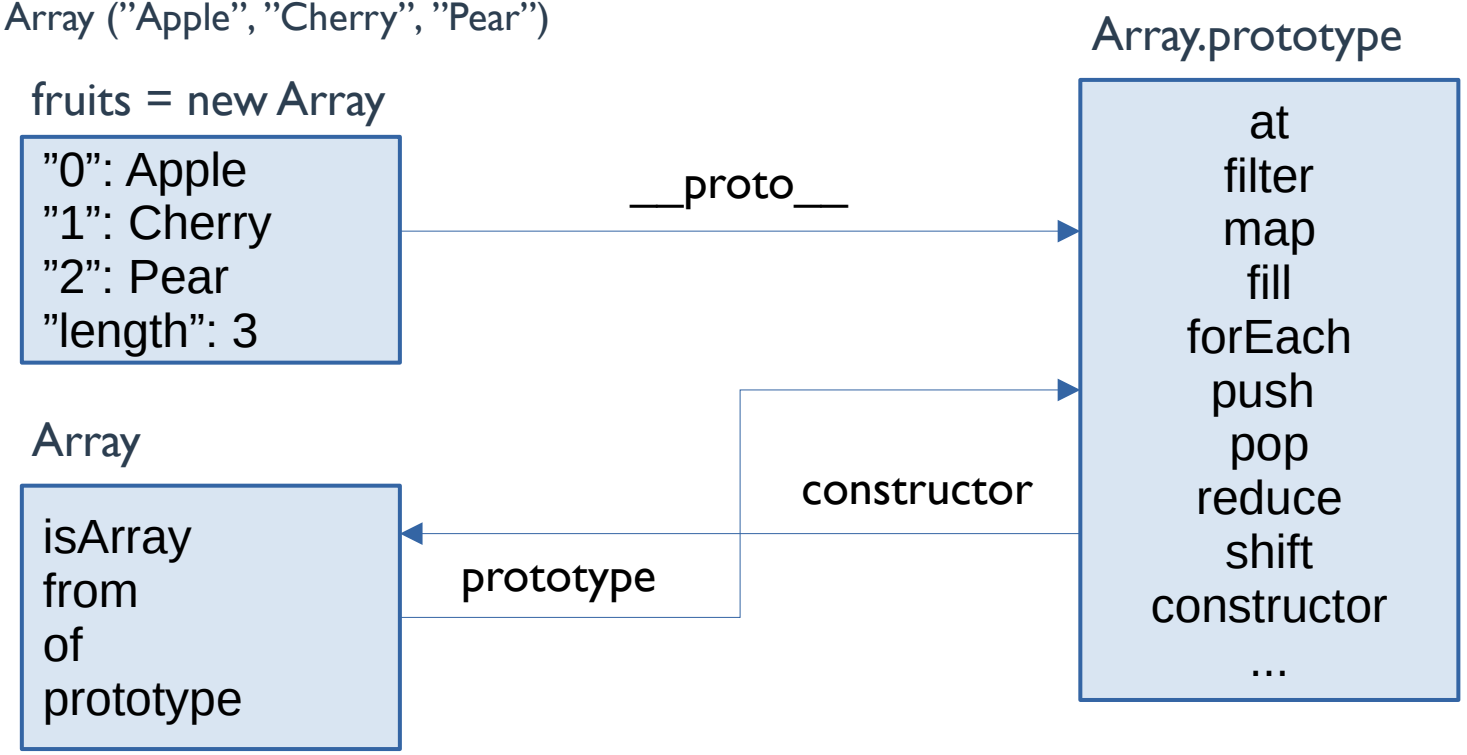
Agenda

- Array
- Map
- Set
- WeakMap
- WeakSet

Array

Features of Array

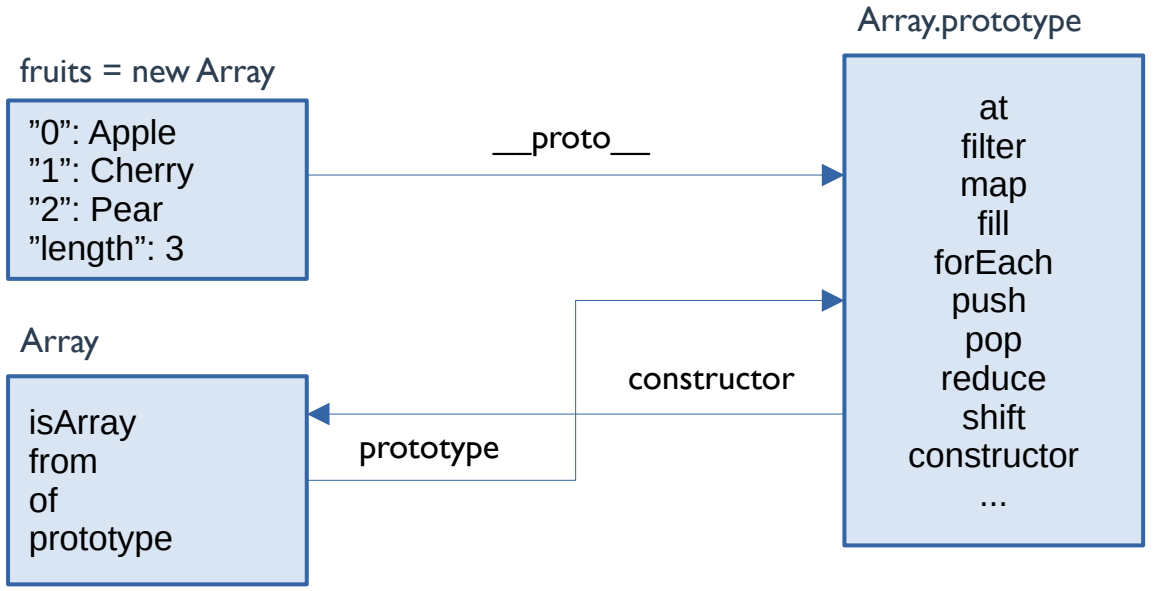
const fruits = [”Apple”, ”Cherry”, ”Pear”];
const fruits = new Array (”Apple”, ”Cherry”, ”Pear”)



Features of Array

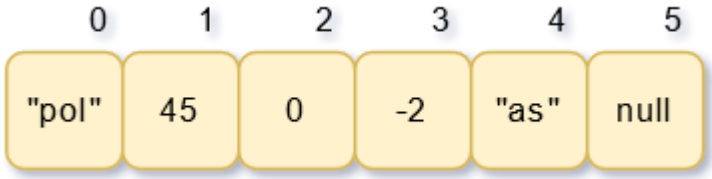
fruits = [”Apple”, ”Cherry”, ”Pear”];
fruits = new Array (”Apple”, ”Cherry”, ”Pear”)

```
Object.keys(fruits); // [”0”, ”1”, ”2”]

Object.getOwnPropertyNames(fruits);
// [ "0", "1", "2", "length" ]

Object.values(fruits);
// [”Apple”, ”Cherry”, ”Pear”]

Object.entries(fruits)
// [[”0”, ”Apple”], [”1”, ”Cherry”], [2, ”Pear”]]
```



Features of Array

- zero-indexed structure (the first element is at index 0)
- resizable
- mixed data types allowed
- some cells may be empty (sparse matrix)
- storage capacity: 2^32-1




```
> arr = new Array(2**32)
▶Uncaught RangeError: Invalid array length
    at <anonymous>:1:7
> arr = new Array(2**32-1)
◂ ▶(4294967295) [empty × 4294967295]
>
```

Initialization

```
// Initialization
emptyArr = new Array();// [], length = 0
emptyArr = Array(); // [], length = 0
tenElements = new Array(10); //[<10 empty slots>], length = 10
twoElements = new Array(10, 4); // [10, 4], length = 2
oneElement = new Array("cat"); //["cat"], length = 1
singleElementArr = Array.of(3); // [3]
singleElementArr = Array.of("dog"); // ["dog"]
twoElementArr = Array.of(3,5); // [3, 5]

fruits = new Array("apple", "banana", "cherry", "grapes"); // length = 4
console.log(Object.keys(fruits)); // [0, 1, 2, 3]
console.log(Object.values(values)); // ["apple", ... , "grapes"]
```

Operations

Accessing

[ ] bracket notation

Array.protoype.at (ES13)

Searching

Array.prototype.indexOf (ES5)

Array.protoype.lastIndexOf (ES5)

Array.protoype.findIndex (ES6)

Array.protoype.find (ES6)

Array.prototype.includes (ES7)

Higher-order functions

Array.prototype.forEach

Array.prototype.map

Array.prototype.reduce

Array.prototype.filter

Array.prototype.sort

Array.prototype.some

Array.prototype.every

Mainpulation

Array.prototype.push

Array.prototype.pop

Array.prototype.shift

Array.prototype.unshift

Array.prototype.slice

Array.prototype.splice

Accessing

- [ ] bracket notation
- Array.prototype.at

fruits[-1] => -1 is transformed into a string and treated as a key, there is no ”-1” key, so undefined is returned

```
fruits = ["pearl", "cherry", "apple", "plum", "kiwi",
          "grapefruit", "blackberry", "pearl"];
fruits[2]; // apple
fruits.at(2); // apple
fruits[-1]; // undefined
fruits.at(-1); // pearl
fruits[fruits.length - 1]; //pearl

function getLast(arr) {
    return arr.at(-1);
}
function getTheLastButOne(arr) {
    return arr.at(-2);
}
```

Searching

- Array.prototype.indexOf
- Array.protoype.lastIndexOf

```
//Find the first/last index at which an element can be found
fruits = ["pearl", "cherry", "apple", "plum", "kiwi",
          "grapefruit", "blackberry", "pearl"];
fruits.indexOf("pearl"); // 0
fruits.indexOf("pearl", 1); // 7
fruits.lastIndexOf("pearl"); // 7
fruits.indexOf("strawberry"); // -1
```

Searching

- Array.prototype.indexOf
- Array.protoype.lastIndexOf

```
fruitDescr = [
    {name: "apple", price: "0.5$", amount: "single"},
    {name: "banana", price: "0.25$", amount: "single"},
    {name: "grapes", price: "5$", amount: "1kg"},
    {name: "plum", price: "0.25$", amount: "single"},
    {name: "peach", price: "3$", amount: "1kg"},
    {name: "apricot", price: "0.75$", amount: "single"},
    {name: "cherry", price: "6$", amount: "1kg"}
];
fruitDescr.indexOf({name: "plum", price: "0.25$", amount: "single"});
// -1
```


Internally the functions use === operator while making comparison

Searching

- Array.protoype.findIndex

```
//Find the first element index that satisfies a test
fruitDescr = [
    {name: "apple", price: "0.5$", amount: "single"},
    {name: "banana", price: "0.25$", amount: "single"},
    {name: "grapes", price: "5$", amount: "1kg"},
    {name: "plum", price: "0.25$", amount: "single"},
    {name: "peach", price: "3$", amount: "1kg"},
    {name: "apricot", price: "0.75$", amount: "single"},
    {name: "cherry", price: "6$", amount: "1kg"}
];
fruitDescr.indexOf({name: "plum", price: "0.25$", amount: "single"});
plumIndex = fruitDescr.findIndex(fruit => fruit.name === "plum" && fruit.price === "0.25$"
    && fruit.amount === "single"); // 3
plumStringRepr = JSON.stringify({name: "plum", price: "0.25$", amount: "single"});
plumIndex = fruitDescr.findIndex(fruit => JSON.stringify(fruit) === plumStringRepr);
// 3
```

Searching

- Array.protoype.find

```
// find the first element that passes a testing function
// Array.prototype.find
const fruitDescr = [
    {name: "apple", price: "0.5$", amount: "single"},
    {name: "banana", price: "0.25$", amount: "single"},
    {name: "grapes", price: "5$", amount: "1kg"},
    {name: "plum", price: "0.25$", amount: "single"},
    {name: "peach", price: "3$", amount: "1kg"},
    {name: "apricot", price: "0.75$", amount: "single"},
    {name: "cherry", price: "6$", amount: "1kg"}
];
const foundFruit = fruitDescr.find(fruit => fruit.amount === "1kg");
```

Searching

- Array.protoype.includes

```
// Array.prototype.includes
// Checks whether an elements is in an array

fruits = ["pearl", "cherry", "apple", "plum", "kiwi",
          "grapefruit", "blackberry", "pearl"];
apple = {name: "apple", price: "0.5$", amount: "single"};
banana = {name: "banana", price: "0.25$", amount: "single"};
cherry = {name: "cherry", price: "6$", amount: "1kg"};
fruitDescr = [apple, banana]

fruits.includes("grapefruit"); // true
fruitDescr.includes(banana); // true
fruitDescr.includes(cherry); // false
```

Modification

myArr = new Array(„test”, 45, -90);

- Array.prototype.push
  myArr.push(1.45, 0.12)

- Array.prototype.pop
  removedEl = myArr.pop()

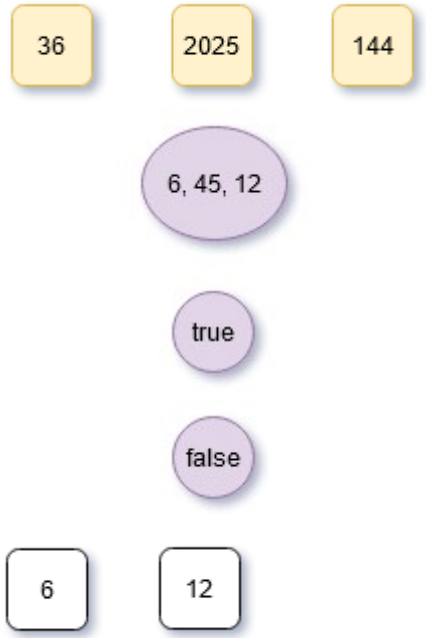
- Array.prototype.shift
  removedEl = myArr.shift()

- Array.prototype.unshift
  myArr.unshift(100, 1000)

Higher-order functions

numbers = new Array(6, 45, 12);

- Array.prototype.**map**
  myArr.map(el => el * el);
- Array.prototype.**forEach**
  myArr.map(el => console.log(el));
- Array.prototype.**some**
  isSomeBigger= myArr.some(el => el > 10);
- Array.prototype.**every**
  areAllBigger = myArr.every(el => el > 10);
- Array.prototype.**filter**
  areAllBigger = myArr.filter(el => el % 2);



Higher-order functions

numbers = new Array(6, 45, 12);

- Array.prototype.**sort**()

Array.prototype.**sort**((a,b) => a – b)

Array.prototype.**sort**((a,b) => b – a)

sort **mutates** the original array

Higher-order functions

- Sorting objects

```
people = [{
    name: "Sandra",
    age: 24,
}, {
    name: "John",
    age: 30,
}, {
    name: "Steph",
    age: 70,
}, {
    name: "Stacy",
    age: 45,
}];

people.sort((a, b) => a.age - b.age);
```

```
▼ (4) [{…}, {…}, {…}, {…}]
  ▶ 0: {name: 'Sandra', age: 24}
  ▶ 1: {name: 'John', age: 30}
  ▶ 2: {name: 'Stacy', age: 45}
  ▶ 3: {name: 'Steph', age: 70}
    length: 4
  ▶ [[Prototype]]: Array(0)
```

Higher-order functions

- Filter

```
sides = [4, 3, 9, 10, 15, 14, 21, 100, 1000];
filtered = [];
for(let i = 0; i < sides.length; ++i) {
    if (sides[i] % 2) {
        filtered.push(sides[i]);
    }
}
```

```
filtered = filter.sides(side => side % 2);
```

```
filtered
▼(4) [3, 9, 15, 21] i
    0: 3
    1: 9
    2: 15
    3: 21
    length: 4
  ▶[[Prototype]]: Array(0)
```

Higher-order functions

- Filter

```
cars = [{
    type: "Audi",motor: "3.0",
}, {
    type: "Fiat", motor: "1.1",
}, {
    type: "Citroen", motor: "2.0",
}];

filtered = [];
for(let i = 0; i < cars.length; ++i) {
    if (parseFloat(cars[i].motor) > 2.5) {
        filtered.push(cars[i]);
    }
}
```

```
> filtered
<- ▼[{…}] i
    ▶0: {type: 'Audi', motor: '3.0'}
     length: 1
    ▶[[Prototype]]: Array(0)
```

```
filtered = cars.filter(car => parseFloat(car.motor) > 2.5);
```

Higher-order functions

- **Task**

  Given a table below, filter out the array and return only string based entries
  (consider only the below set of values, do not take into account *NaN*, *Inifnity* and *Number* instances)

```
items = ["pineapple", 56, "apple", 12.9, "water", 1, 0];
```

Higher-order functions

- **Solution**

```
items.filter(item => typeof item === "number");
```

Higher order functions

- **Task**

Having the table on the right, perform the following actions:

- Filter out entries for which an user is > 40 and < 20 years old
- Leave only those elements whose user's name starts with „A", „B" or „C"
- Create an array that contains only „female" entries

```
arr = [{
    name: "Pol",age: 41,
}, {
    name: "Anny", age: 35,
}, {
    name: "Chris", age: 32,
}, {
    name: "Anthony", age: 70,
}, {
    name: "Julia", age: 25,
}, {
    name: "Brian", age: 18,
}, {
    name: "Sindy", age: 29,
}];
```

Solution

```
// filter out entries for which an user is > 40 and < 20 years old
filtered = arr.filter(entry => entry.age <= 40 && entry.age >= 20);
console.log(filtered)

//Leave only those elements whose user's name starts with „A", „B" or „C"
filtered = arr.filter(entry => entry.name.startsWith("A") ||
                    entry.name.startsWith("B") || entry.name.startsWith("C"));
console.log(filtered)

// Create an array that contains only „female" entries
filtered = arr.filter(entry => ["Sindy", "Anny", "Julia"].includes(entry.name));
console.log(filtered)
```

Higher-order functions

- Array.prototype.map

```
fruits = ["apple", "pineapple", "plum", "cherry"];
names = ["Paul", "Adam", "Joseph", "Anny", "Frank"];
names.map(name => {
    const randomIndex = Math.floor(Math.random() * fruits.length);
    return `${name} likes ${fruits[randomIndex]}`;
});
```

```
(5) ['Paul likes plum', 'Adam likes plum', 'Joseph likes plum', 'An
ny likes pineapple', 'Frank likes apple']
  0: "Paul likes plum"
  1: "Adam likes plum"
  2: "Joseph likes plum"
  3: "Anny likes pineapple"
  4: "Frank likes apple"
  length: 5
▶ [[Prototype]]: Array(0)
```


Does **not mutate** the original array

Higher-order functions

- **Task**

  Treat the elements of the below array as a side of a square and calculate its area and perimeter

```
sides = [12.5, 67, 124, 0, 90, 100, 1000];
```

- **Task**

  Create an array (100 elements) of randomly generated numbers
  (max value = 1000, min value = 0)

Higher-order functions

- **Solution**

```
sides.map(side => ({
    area: side * side,
    perimeter: 4 * side,
}));
```

```
(7) [{…}, {…}, {…}, {…}, {…}, {…}, {…}]
  0: {area: 156.25, perimeter: 50}
  1: {area: 4489, perimeter: 268}
  2: {area: 15376, perimeter: 496}
  3: {area: 0, perimeter: 0}
  4: {area: 8100, perimeter: 360}
  5: {area: 10000, perimeter: 400}
  6: {area: 1000000, perimeter: 4000}
  length: 7
  [[Prototype]]: Array(0)
```

```
N = 100, MAX_VALUE = 1000;
numbers = new Array(N).fill().map(() => Math.random() * MAX_VALUE);
```

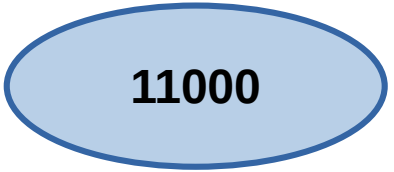
```
> N = 100, MAX_VALUE = 1000;
  numbers = new Array(N).fill().map(() => Math.random()
  * MAX_VALUE);
<- (100) [476.35915291268185, 736.8301329486234, 446.8
  729157022988, 654.8486246664036, 799.5352863442324,
  400.85077918124966, 698.7093803158692, 795.32338672
  02421, 418.7378969338249, 929.2701842729143, 920.49
  73492073462, 471.91661529101924, 928.1199569581804,
  885.8249744296562, 31.47852761837222, 740.301888115
  1351, 300.20582364617866, 822.5158619720477, 198.52
  8012538844, 533.5973989485572, 470.0056026221733, 2
  77.83438187134556, 187.0286541878654, 93.8420820876
  3134, 834.6086602248799, 811.0471973455855, 715.803
  2270214084, 323.67744029501046, 270.55279421903447,
  959.2897609920142, 372.5500030155233, 621.273900267
```

Higher-order functions

- **Array.protoype.reduce**

  Compute the total value of the below devices

```
items = [{
    name: "PSX", price: 300, items: 4,
}, {
    name: "Nintendo", price: 400, items: 2,
}, {
    name: "Xbox", price: 500, items: 10,
}, {
    name: "Wii", price: 40, items: 100,
},];

totalPrice = items.reduce((acc, device) => acc + device.items * device.price , 0);
```



Higher-order functions

- **Task**
  - Compute the sum of all elements

```
N = 100, MAX_VALUE = 1000;
numbers = new Array(N).fill().map(() => Math.random() * MAX_VALUE);
```

Higher-order functions

- **Solution**

```
totalSum = numbers.reduce((sum, el) => sum + el, 0);
```

Playing with *length* property

The total number of elements is defined by **length** property. This is also *writable* property. Just look below how *length* modification changes an array content

```
arr = ["Cherry", "Grapes", "Apple", "Plum", "Pear"];
arr.length // 5
arr[10] // undefined

console.log(arr.join(" ")); // "Cherry Grapes Apple Plum Pear"
arr.length = 10;
arr //  Array(10) [ "Cherry", "Grapes", "Apple", "Plum", "Pear", <5 empty slots> ]

arr.length = 2; // deletes elements
arr // Array [ "Cherry", "Grapes" ]
console.log(arr.join(" ")); // "Cherry Grapes"
```

array-like objects

Array methods are generic what means they access only array elements using *indexes* and ***length*** property. That is why they can be invoked for **array-like** objects

```
obj = {
    "1": "Pol",
    "2": "Martin",
    "0": "Fredy",
    "3": "James",
    "4": "John",
    length: 5,
}

filtered = Array.prototype.filter.call(obj, name => name.startsWith("J"));
console.log(filtered); // ['James', 'John']
```

array-like objects

Array methods are generic what means they access only array elements using *indexes* and ***length*** property. That is why they can be invoked for **array-like** objects

```
obj = {
    "1": "Pol",
    "2": "Martin",
    "0": "Fredy",
    "3": "James",
    "4": "John",
    length: 5,
}

joined = Array.prototype.join.call(obj, " -> ");
console.log(joined); // Fredy -> Pol -> Martin -> James -> John
```

array-like objects

Array methods are generic what means they access only array elements using *indexes* and ***length*** property. That is why they can be invoked for **array-like** objects

```
obj = {
    "1": "Pol",
    "2": "Martin",
    "0": "Fredy",
    "3": "James",
    "4": "John",
    length: 5,
}

isJamesPresent = Array.prototype.includes.call(obj, "James");
console.log(isJamesPresent); // true
```

DOM collections

NodeList and HTMLCollection can be considered as array-like structures

```
divs = document.querySelectorAll("div"); // NodeList structure
Array.prototype.forEach.call(divs, el => el.id !== "" && console.log(el.id));
```

```
> Array.prototype.forEach.call(divs, el => el.id !== "" && console.log(el.id))
  spacing_hidden_wrapper
  spacing_hidden
  header_wrapper
  header
  search_elements_hidden
  duckbar
  zero_click_wrapper
  vertical wrapper
```

Map

Map

- key-value collection
  - Both key and value might be **any** data type
  - Remembers the original insertion order,
  - Any value can be used as a key and as a value
  - A key may apperar only once

Maps must be implemented using either hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of elements in the collection. The data

https://tc39.es/ecma262/multipage/keyed-collections.html#sec-map-objects

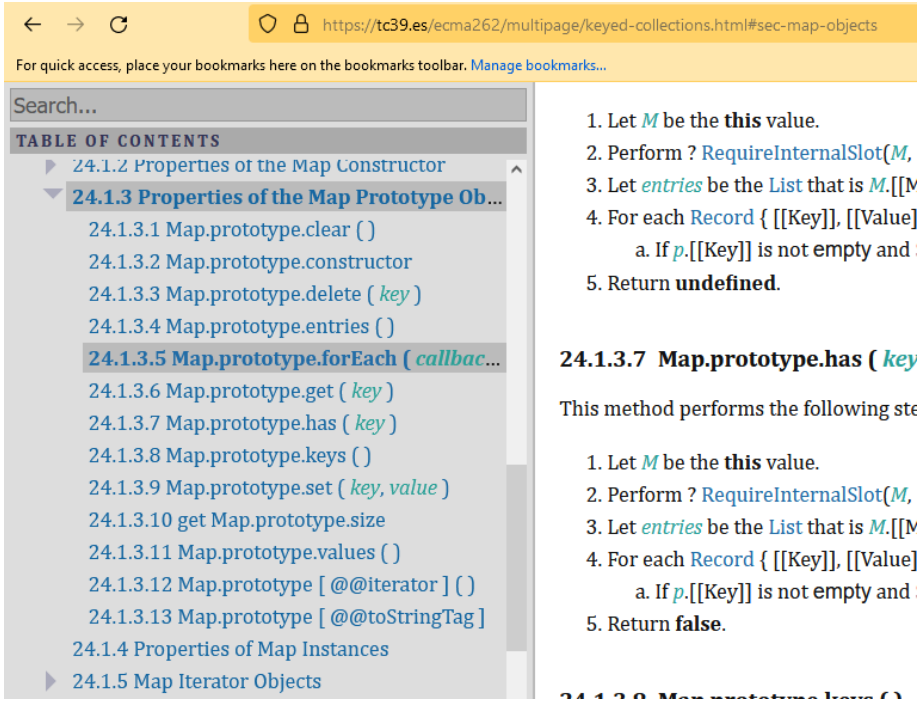
Map

- Map is a global object,
- Cannot be called as a regular function,
- Invoked using ***new*** operator

```
> Map
ƒ Map() { [native code] }
> Map()
▶Uncaught TypeError: Constructor Map requires 'new'
    at Map (<anonymous>)
    at <anonymous>:1:1
> new Map
▶Map(0) {size: 0}
>
```

Map

```
> Map.prototype
Map {constructor: ƒ, get: ƒ, set: ƒ, has: ƒ, delete: ƒ, …}
  clear: ƒ clear()
  constructor: ƒ Map()
  delete: ƒ delete()
  entries: ƒ entries()
  forEach: ƒ forEach()
  get: ƒ ()
  has: ƒ has()
  keys: ƒ keys()
  set: ƒ ()
  size: (...)
  values: ƒ values()
  Symbol(Symbol.iterator): ƒ entries()
  Symbol(Symbol.toStringTag): "Map"
  get size: ƒ size()
  [[Prototype]]: Object
```



Map

Purpose: familiarize yourself with initialization, set, get and iteration regarding Map object

Task: write a map that keeps information about a price of fruits. Using keys like "apple", "grapes" you will get corresponsding data (price and info about whether that is a price for 1kg or for a single item). Show how you can iterate over the structure

Examples:

fruits("apple"); // {price: "0.5$", amount: "single"}

```
const fruitDescr = [
    ["apple", {price: "0.5$", amount: "single"}],
    ["banana", {price: "0.25$", amount: "single"}],
    ["grapes", {price: "5$", amount: "1kg"}],
    ["plum", {price: "0.25$", amount: "single"}],
    ["peach", {price: "3$", amount: "1kg"}],
    ["apricot", {price: "0.75$", amount: "single"}],
    ["cherry", {price: "6$", amount: "1kg"}],
];
```

Map – initialization, *set*, *get*, *has*

```
// initialization and setter
const fruitDescr = [
    ["apple", {price: "0.5$", amount: "single"}],
    ["banana", {price: "0.25$", amount: "single"}],
    ["grapes", {price: "5$", amount: "1kg"}],
    ["plum", {price: "0.25$", amount: "single"}],
    ["peach", {price: "3$", amount: "1kg"}],
    ["apricot", {price: "0.75$", amount: "single"}],
    ["cherry", {price: "6$", amount: "1kg"}],
];
const fruits1 = new Map(fruitDescr);
const fruits2 = new Map();
fruitDescr.forEach(fruit => {
    const [fruitName, fruitDetails] = fruit;
    fruits2.set(fruitName, fruitDetails);
});

//only one key exists
fruits1.set("plum", null);
fruits2.set("plum", null);
```

```
> // Get the description of "cherry"
  fruits1.get("cherry");
< ▶ {price: '6$', amount: '1kg'}
> fruits2.get("cherry");
< ▶ {price: '6$', amount: '1kg'}
> // Get the description of "plum"
  fruits1.get("plum");
< undefined
> fruits2.get("plum");
< undefined

> // find out whether a given key exists
  fruits1.get("pear");
< undefined
> fruits1.has("pear");
< false
>
```

Map - iterations

```
const fruitDescr = [
    ["apple", {price: "0.5$", amount: "single"}],
    ["banana", {price: "0.25$", amount: "single"}],
    ["grapes", {price: "5$", amount: "1kg"}],
    ["plum", {price: "0.25$", amount: "single"}],
    ["peach", {price: "3$", amount: "1kg"}],
    ["apricot", {price: "0.75$", amount: "single"}],
    ["cherry", {price: "6$", amount: "1kg"}],
];
```

```
// Iterations
// displaying only fruit names:
for (let fruitName of fruits1.keys()) {
    console.log(`${fruitName}`);
}
// show only fruit description
for (let fruitDescription of fruits1.values()) {
    console.log(`${fruitDescription.price} ${fruitDescription.amount}`);
}
// displaying the full content of a map
for (let [fruiteName, fruitDescription] of fruits1) {
    console.log(`${fruiteName}: ${fruitDescription.price} ${fruitDescription.amount}`);
}
fruits1.forEach((fruitDescription, fruitName) => {
    console.log(`${fruitName}: ${fruitDescription.price} ${fruitDescription.amount}`);
});
```

Map – other usefull scenarios

```
// other usefull use cases

// cloning
const fruitsCloned = new Map(fruits1); // shallow copy !
fruitsCloned.get("apple").price = "20$"; // {price: "0.5$", amount: "single"}

console.log(fruits1.get("apple").price); // 20$

// get the total numbers of fruits
console.log(`Map contains ${fruits2.size} fruit(s)`);

// clear the whole structure
fruits1.clear();
```

Map – conversion to Array and to Object

```
// conversion to Array
// fruitArr = Array.from(fruits1.entries());
fruitArr = Array.from(fruits1);
fruitArrSpread = [...fruits1];
fruitNamesArr = Array.from(fruits1.keys());
fruitDescriptArr = Array.from(fruits1.values());
```

```
> fruitArr = Array.from(fruits1);
(7) [Array(2), Array(2), Array(2), Array(2), Array(2), Array(2), Array(2)]
  0: (2) ['apple', {…}]
  1: Array(2)
    0: "banana"
    1: {price: '0.25$', amount: 'single'}
    length: 2
    [[Prototype]]: Array(0)
  2: (2) ['grapes', {…}]
  3: (2) ['plum', {…}]
  4: (2) ['peach', {…}]
  5: (2) ['apricot', {…}]
  6: (2) ['cherry', {…}]
  length: 7
  [[Prototype]]: Array(0)
>
```

```
// conversion to Object
fruitsObj = Object.fromEntries(fruits1);
```

```
> Object.fromEntries(fruits1)
{apple: {…}, banana: {…}, grapes: {…}, plum: {…}, peach: {…}, …}
  apple:
    amount: "single"
    price: "0.5$"
    [[Prototype]]: Object
  apricot: {price: '0.75$', amount: 'single'}
  banana: {price: '0.25$', amount: 'single'}
  cherry: {price: '6$', amount: '1kg'}
  grapes: {price: '5$', amount: '1kg'}
  peach: {price: '3$', amount: '1kg'}
  plum: {price: '0.25$', amount: 'single'}
  [[Prototype]]: Object
>
```

Why not to use Object as a map ?

```
// alternative
let map = {};

map.name = "Pol";
map.age = 41;
map.fruits = [
    ["apple", {price: "0.5$", amount: "single"}],
    ["banana", {price: "0.25$", amount: "single"}],
];

console.log(map.name, map.age); // Pol 41
map.fruits.forEach(fruit => {
    console.log(fruit[0], fruit[1].price);// apple 0.5$ \n banana 0.25$
});
```

Why not to use Object as a map ?

```
// collisions with build in functions
map.toString = true;

// cannot get the size easily
Object.keys(map).length; // all enumerable properites without inherited ones
Object.getOwnPropertyNames(map).length; // all enumerable and non-enumerbale own properties

// only strings and symbols as a key
map[45] = "Poland";
map[null] = "0;"
Object.keys(map) // ['45', 'name', 'age', 'fruits', 'toString', 'null']

// iteration does not keep insertion order
for(let el in map) {
    if (Object.prototype.hasOwnProperty.call(map, el)) {
        console.log(el);
    }
} // 45, name, age, fruits, toString, null
```


automatic conversion of keys to string

Map vs Object

topic	Map	Object
keys	any data type	only string and symobls
size	***size*** property	no
iteration	itterable by default (see *for of*)	need to use *Object.keys* or *Object.values*
performance	access time is sublinear	not optimized for frequent addition/removal
serialization	more complicated *JSON.stringify(Array.from(map.entries()));*	*JSON.stringify*
insertion order	kept	*not kept*
default keys	no	yes (possible collisions)

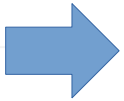
WeakMap

WeakMap

Same like map but keys must be **objects**

However the reference to a key is not created (see an example) ! This implies that the presence of a key in a map does not prevent it from being removed by the garbage collector

```
> map = new Map();
< ▶ Map(0) {size: 0}
> obj = {name: "Pol", age: 41};
< ▶ {name: 'Pol', age: 41}
> map.set(obj, "someValue");
< ▶ Map(1) {{…} => 'someValue'}
> obj = null;
< null
> map
< ▼ Map(1) {{…} => 'someValue'}
    ▼ [[Entries]]
      ▶ 0: {Object => "someValue"}
      size: 1
    ▶ [[Prototype]]: Map
>
```



```
> weak = new WeakMap
< ▶ WeakMap {}
> obj = {name: "Pol", age: 41}
< ▶ {name: 'Pol', age: 41}
> weak.set(obj, "someValue");
< ▶ WeakMap {{…} => 'someValue'}
> obj = null
< null
> gc()
< undefined
> weak
< ▶ WeakMap {}
```


setting null does not cause for gc to be run

In chrome, you can call gc explicitly but even then the cleaning may not happen

```
chrome --js-flags="--expose-gc"
```

Use case - memoization

- The code on the left side performs well. However if we do not need longer ***refObj*** (which can be heavy), then we need to clear the map !!! [cache.clear()]. Otherwise it will not be collected. On the other hand the same code on the right side has no problem with it

```
cache = new Map();
function heavyCalculation(refObj) {
    if (cache.has(refObj)) {
        return cache.get(refObj);
    }
    let result;
    // perform looong computation
    ///...
    map.set(refObj, result);

    return result;
}
```

```
cache = new WeakMap();
function heavyCalculation(refObj) {
    if (cache.has(refObj)) {
        return cache.get(refObj);
    }
    let result;
    // perform looong computation
    ///...
    map.set(refObj, result);

    return result;
}
```

Use case – attaching the additional data

- Modify the code below to track how many times the given objects was processed by a function

```
function process(refObj) {
    // ... some processing
}
```

Solution

- Notice, that if the object is no longer needed (there is no reference to it), then the corresponfing value is automatically collected !

```
const processCounterMap = new WeakMap;
function process(refObj) {
    const counter = processCounterMap.has(refObj) ? ++processCounterMap.get(refObj) : 1;
    // ... some processing

    processCounterMap.set(refObj, {
        counter,
    });
}
```

Set

API

- **Set keeps the unique data**. The passed variable may be a primitive data or an object
- The interface is very similar to Map API

```
> Set.prototype
< Set {constructor: ƒ, has: ƒ, add: ƒ, de
    add: ƒ add()
    clear: ƒ clear()
    constructor: ƒ Set()
    delete: ƒ delete()
    entries: ƒ entries()
    forEach: ƒ forEach()
    has: ƒ has()
    keys: ƒ values()
    size: (...)
    values: ƒ values()
    Symbol(Symbol.iterator): ƒ values()
    Symbol(Symbol.toStringTag): "Set"
    get size: ƒ size()
    [[Prototype]]: Object
>
```

```
> Map.prototype
< Map {constructor: ƒ, get: ƒ, set: ƒ, has:
    clear: ƒ clear()
    constructor: ƒ Map()
    delete: ƒ delete()
    entries: ƒ entries()
    forEach: ƒ forEach()
    get: ƒ ()
    has: ƒ has()
    keys: ƒ keys()
    set: ƒ ()
    size: (...)
    values: ƒ values()
    Symbol(Symbol.iterator): ƒ entries()
    Symbol(Symbol.toStringTag): "Map"
    get size: ƒ size()
    [[Prototype]]: Object
>
```

Task

- **Task:**
  - Data contains the login and logout time and an associated user. Notice, that some entries may be bound with the same user
  - **Goal**:
    - create a set
    - populate it using data on the right side
    - display the set content

```
const data = [{
        user: "Pol",
        logging: "05:10:14",
        logout: "06:10:15",
    },{
        user: "Anny",
        logging: "12:08:14",
        logout: "16:04:45",
    },{
        user: "Chris",
        logging: "12:20:14",
        logout: "14:10:15",
    },{
        user: "Brian",
        logging: "17:20:34",
        logout: "19:30:45",
    },{
        user: "Cindy",
        logging: "08:00:00",
        logout: "17:00:00",
    },{
```

Solution

- Populating a set

```
// how to create a set
users = new Set();
data.forEach(entry => {
    users.add(entry.user);
});

userNames = data.map(entry => entry.user);
users = new Set(userNames);
```

- Displaying data

```
// displaying data
for(let user of users) {
    console.log(user);
}
users.forEach(user => console.log(user));
console.log([...users.values()]); // displays as an array
console.log([...users.keys()]); // keys is an alias for values
```

WeakSet

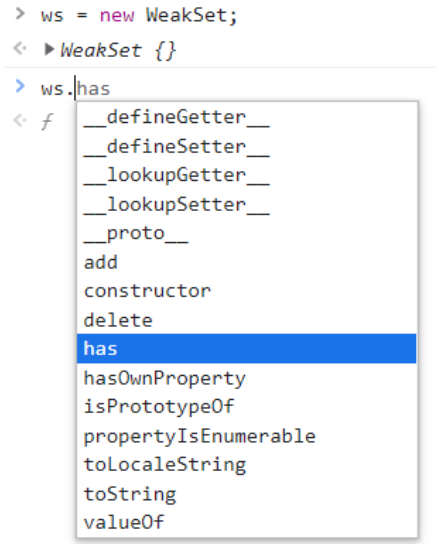
WeakSet

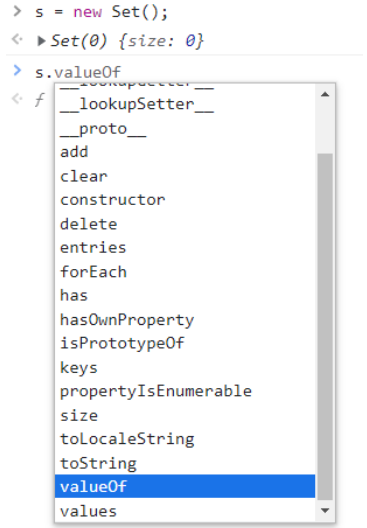
- Similar like a regular *Set*, however it keeps weak references (and only objects). If there is no reference to the kept value, then it will be automatically removed by the garbage collector from the weak set
- Helps reduces the memory consupmtion
- You may need *WeakSet* if you want to tag an object (set some flag) without the object mutation
- *WeakSet* may be considered as a special cas of WeakMap where all values are boolean
- Not-iterable, no *size* property. There are some rationals which claim that *WeakSet* cannot be iterable becuase it would be GC dependent (would return non-deterministic results)

WeakSet

```
ws = new WeakSet
{
    let obj = {
        counter: 2,
    };

    ws.add(obj);
    ws.has(obj); // true
}

ws.has(obj); // false
```

WeakSet - API





Notice, there is no functions like keys, values or forEach

WeakSet

- **Task**
  - Modify the below code to assure for a process function to be invoked only for *Fruit* based objects

```
class Fruit {
    process() {
        //...
    }
}
```

Solution

```
const fruitInstSet = new WeakSet;
class Fruit {
    constructor() {
        fruitInstSet.add(this);
    }
    process() {
        if (!fruitInstSet.has(this)) {
            throw new Error("cannot call process function for a non-fruit object");
        }
        // ...
    }
}
```

WeakSet

- **Task**
  - Modify the below code to prevent the same object from being added again

```
function addToDomContainer(el) {
    $container.appendChild(el);
    console.log(`Element added to the container ${el.id}`);
}
```

In a nutshell, the task is to add information that if **el** has been already seen, then it should not be handled again

Solution

```
const addedElements = new WeakSet();
function addToDomContainer(el) {
    if (!addedElements.has(el)) {
        throw new Error("Cannot add the same element again", el);
    }
    $container.appendChild(el);
    addedElements.add(el);
    console.log(`Element added to the container ${el.id}`);
}
```

References

- https://www.zhenghao.io/posts/object-vs-map
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map
- https://javascript.info/weakmap-weakset

Thank You !