MAP/SET

MAP

- Mapy służą do tworzenia zbiorów z parami [klucz - wartość]. Przypominają one klasyczne obiekty, natomiast główną różnicą odróżniającą je od klasycznych obiektów, jest to, że kluczami może być tutaj dowolny typ danych.
- Tworzenie Map:

```
1  const map = new Map();
2  map.set("kolor1", "red");
3  map.set("kolor2", "blue");
4
5  //lub
6
7  const map = new Map([
8      ["kolor1", "red"],
9      ["kolor2", "blue"],
10 ]);
```

MAP

Dla każdej mapy mamy dostęp do kilku metod:

- **set(key, value) -** Ustawia nowy klucz z daną wartością,
- **get(key) -** Zwraca wartość danego klucza,
- **has(key)** - Sprawdza czy mapa ma dany klucz,
- **delete(key)** - Usuwa dany klucz i zwraca true/false jeżeli operacja się udała,
- **clear()** - Usuwa wszystkie elementy z mapy,
- **entries()** - Zwraca iterator zawierający tablicę par [klucz-wartość],
- **keys() -** Zwraca iterator zawierający listę kluczy z danej mapy,
- **values() -** Zwraca iterator zawierający listę wartości z danej mapy,
- **ForEach** - robi pętlę po elementach mapy,
- **prototype[@@iterator]() -** Zwraca iterator zawierający tablicę par [klucz-wartość]

KLUCZE W MAPIE

- Mapy w przeciwieństwie do obiektów mogą mieć klucze dowolnego typu, gdzie w przypadku obiektów (w tym tablic) są one konwertowane na tekst:

```
const map = new Map();

map.set("1", "Kot");
map.set(1, "Pies");

console.log(map); //{"1" => "Kot", 1 => "Pies"}
```

```
const ob = {}

ob["1"] = "Kot";
ob[1] = "Pies";

console.log(ob); //{"1" : "Pies"}
```

```
const map = new Map();

const ob1 = { name : "test1" }
const ob2 = { name : "test2" }

map.set(ob1, "koty");
map.set(ob2, "psy");
map.set("[object Object]", "świnki");

console.log(map); //{{…} => "koty", {…} => "psy", "[object Object]" => "świnki"}
```

KLUCZE W MAPIE

- W przypadku klasycznych obiektów, klucze zawsze są konwertowane na tekst (obiekty na zapis `[object Object]`:

```
1  const map = {}
2
3  const ob1 = { name : "test1" }
4  const ob2 = { name : "test2" }
5
6  map[ob1] = "koty";
7  map[ob2] = "psy"; //ob2 skonwertowany na "[object Object]"
8  map["[object Object]"] = "świnki";
9
10 console.log(map); //{"[object Object]": "świnki"}
```

PĘTLA PO MAPIE

Jeżeli będziemy chcieli iterować po mapie, możemy wykorzystać pętlę *for of* i poniższe funkcje zwracające iteratory:

- **entries()** - Zwraca tablicę par klucz-wartość,
- **keys()** - Zwraca tablicę kluczy,
- **values()** - Zwraca tablicę wartości

```
const map = new Map([
    ["kolor1", "red"],
    ["kolor2", "blue"],
    ["kolor3", "yellow"]
]);

for (const key of map.keys()) {
    //kolor1, kolor2, kolor3
}

for (const key of map.values()) {
    //red, blue, yellow
}

for (const entry of map.entries()) {
    //["kolo1", "red"]...
}

for (const [key, value] of map.entries()) {
    //key : "kolo1", value : "red"...
}

for (const entry of map) {
    //["kolor1", "red"]...
}
```

PĘTLA PO MAPIE

Do iterowania możemy też wykorzystać wbudowaną w mapy funkcję *forEach*:

```
const map = new Map([
    ["kolor1", "red"],
    ["kolor2", "blue"],
    ["kolor3", "yellow"]
]);

map.forEach((value, key, map) => {
    console.log(`
        Wartość: ${value}
        Klucz: ${key}
    `);
});
```

SET

- Obiekt Set jest kolekcją składającą się z unikalnych wartości, gdzie każda wartość może być zarówno typu prostego jak i złożonego. W przeciwieństwie do mapy jest to zbiór pojedynczych wartości.
- Tworzenie Set:

```
1   const set = new Set();
2   set.add(1);
3   set.add("text");
4   set.add({name: "kot"});
5   console.log(set); //{1, "text", {name : "kot"}}
6
7   //lub
8   //const set = new Set(elementIterowalny);
9   const set = new Set([1, 1, 2, 2, 3, 4]); //{1, 2, 3, 4}
10  const set = new Set("kajak"); //{"k", "a", "j"}
```

SET

Obiekty Set mają podobne właściwości i metody co obiekty typu Map, z małymi różnicami:

- **add(value)** - Dodaje nową unikatową wartość. Zwraca Set,
- **clear()** - Czyści całą mapę,
- **delete(key)** - Usuwa dany klucz i zwraca true/false jeżeli operacja się udała,
- **entries()** - Zwraca iterator zawierający tablicę par [klucz-wartość],
- **has(key)** - Sprawdza czy mapa ma dany klucz,
- **keys()** - Zwraca iterator zawierający listę kluczy z danej mapy,
- **values()** - Zwraca iterator zawierający listę wartości z danej mapy,
- **forEach** - robi pętlę po elementach mapy,
- **prototype[@@iterator]()** - Zwraca iterator zawierający tablicę par [klucz-wartość]

SET

- W przypadku Set() klucze i wartości są takie same, dlatego robiąc pętle nie ważne czy użyjemy values(), keys(), entries() czy po prostu zrobimy pętlę for of:

```
const set = new Set([1, "kot", "pies", "świnka"]);

//wszystkie pętle zadziałają podobnie
for (const val of set.values()) {
    console.log(val);
}

for (const key of set.keys()) {
    console.log(key);
}

for (const [key, val] of set.entries()) {
    console.log(key, val); //key === val
}

for (const el of set) {
    console.log(el);
}
```

SET I TABLICE

- Dzięki temu, że Set zawiera niepowtarzające się wartości, możemy to wykorzystać do odsiewania duplikatów w praktycznie dowolnym elemencie iteracyjnym - np. w tablicy:

```
const tab = [1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 5, 5];

const set = new Set(tab);
console.log(set); //{1, 2, 3, 4, 5}

const uniqueTab = [...set];
console.log(uniqueTab); //[1, 2, 3, 4, 5]
```

```
const tab = [
    "ala",
    "bala",
    "cala",
    "ala",
    "ala"
]

const tabUnique = [... new Set(...tab)];
console.log(tabUnique); //["ala", "bala", "cala"]
```

SET I TABLICE

- To samo tyczy się oczywiście dynamicznie tworzonych setów:

```
1  const set = new Set("kot");
2  console.log(set) //Set {"k", "o", "t"}
3  set.add("k");
4  set.add("k");
5  set.add("t");
6  set.add("y");
7  console.log(set); //Set {"k", "o", "t", "y"}
```

WEAKMAP()

WeakMap to odmiana Mapy, którą od Map rozróżniają trzy rzeczy:

- Nie można po niej iterować,
- Kluczami mogą być tylko obiekty,
- Jej elementy są automatycznie usuwane gdy do danego obiektu (klucza) nie będzie referencji,

Aby stworzyć nową WeakMap, skorzystamy z instrukcji:

```
1   const ob1 = {};
2   const ob2 = {};
3   const ob3 = {};
4
5   const weak = new WeakMap();
6   weak.set(ob1, "lorem");
7   wm.set(ob2, {name : "Karol"});
8
9   weak.get(ob1); //"lorem"
10  weak.has(ob1); //true
11  weak.has(ob3); //false
```

WEAKMAP()

Każda mapa daje nam kilka metod:

- **set(key, value)** - Ustawia wartość dla klucza,
- **get(key)** - Pobiera wartość klucza,
- **has(key)** - Zwraca true/false w zależności czy dana WeakMap posiada klucz o danej nazwie,
- **delete(key) -** Usuwa wartość przypisaną do klucza,

WEAKMAP()

- W odróżnieniu do Map elementy WeakMap są automatycznie usuwane jeżeli do danego obiektu/klucza nie będzie żadnych referencji.

```
1  let ob = { name : "Karol" }
2
3  const weak = new WeakMap();
4  weak.set(ob, "...");
5
6  ob = null;
7  console.log(weak);
```

WEAKSET()

- Są to kolekcje składające się z unikalnych obiektów. Podobnie do WeakMap obiekty takie będą automatycznie usuwane z WeakSet, jeżeli do danego obiektu zostaną usunięte wszystkie referencje.

```
1  const set = new WeakSet();
2  const a = {};
3  const b = {};
4
5  set.add(a);
6  set.add(b);
7  set.add(b);
8  console.log(set); //{a, b}
```

WEAKSET()

Każdy WeakSet udostępnia nam metody:

- **add(ob)** - Dodaje dany obiekt do kolekcji,
- **delete(ob) -** Usuwa dany obiekt z kolekcji,
- **has(ob)** - Zwraca true/false w zależności, czy dana kolekcja zawiera dany obiekt,

WEAKSET()

- WeakSet idealnie nadaje się do zbierania w jeden zbiór obiektów, które potencjalnie w dalszej części skryptu mogą zostać usunięte, a więc nie powinny być trzymane w naszej "liście":

```
let user1 = {}
let user2 = {}
let user3 = {}

const userList = new WeakSet();
userList.add(user1);
userList.add(user2);
userList.add(user3);

user1 = null;
userList.has(user1); //false
```