



Webapplicaties II

Hoofdstuk 04 – Functional Programming
met Arrays

A large, stylized orange cross graphic that serves as a background element on the left side of the slide.

04 Functional Programming met Arrays

Inhoud

- Functioneel programmeren
- Arrays
 - Herhaling
 - map / filter / reduce
 - Geavanceerde methodes
- Maps
- Sets
- Rest en spread operator

Functioneel programmeren

Functional programming is the process of building software by composing **pure functions**, avoiding **shared state, mutable data, and side-effects**. Functional programming is **declarative** rather than **imperative**, and application state flows through pure functions.

<https://medium.com/javascript-scene/master-the-javascript-interview-what-is-functional-programming-7f218c68b3a0>

Pure functions

- Een pure functie is een voorspelbare functie
 - Als je de functie aanroept krijg je met dezelfde input, steeds dezelfde output
 - Geen side-effects (DOM manipulatie, externe variabelen wijzigen,...)
- In een programma kan je de aanroep naar een pure functie vervangen door het resultaat van de functie aanroep zonder de werking van het programma te veranderen
- Een pure functie heeft **altijd** een return statement.

Shared state

- Een shared state is elke variabele of object die bestaat in een gedeelde scope of die wordt doorgegeven naar een andere scope

```
// Een gedeelde variabele creëren
```

```
let gedeeldeVariabele = 0;
```

```
function verhogen(){  
    gedeeldeVariabele += 1;  
}
```

```
function verdubbelen(){  
    gedeeldeVariabele *= 2;  
}
```

```
verhogen();  
console.log(gedeeldeVariabele);  
verdubbelen();  
console.log(gedeeldeVariabele);
```

Mutable vs Immutable objecten

- Een immutable (onveranderlijk) object is een object dat, na creatie, niet meer kan gewijzigd worden.
 - Een mutable (veranderlijk) object kan wel gewijzigd worden
- Als we een shared state object muteren kan dit een onvoorspelbaar/ongewenst effect hebben op ons programma
- We willen dus zoveel mogelijk onveranderlijke data.
 - Dit kunnen we door wijzigingen steeds door te voeren op kopies zodat de originele waarde behouden blijft

Mutable vs Immutable objecten

// Een array maken (muteerbaar)


```
const hobbies = [  
  'programmeren',  
  'gamen',  
  'voetbal'  
];
```

```
const omgekeerdeHobbies =  
hobbies.reverse();
```

```
console.log(omgekeerdeHobbies);  
//[ 'voetbal', 'gamen', 'programmeren' ]
```

```
console.log(hobbies);  
//[ 'voetbal', 'gamen', 'programmeren' ]
```

Muteert de
originele data



// Een string maken (niet-muteerbaar)


```
const origineel = "Ik ben niet muteerbaar";
```

```
const gewijzigd = origineel.replace("Ik ben  
niet muteerbaar", "Ik ben gewijzigd");
```

```
console.log(gewijzigd);  
// "Ik ben gewijzigd"
```

```
console.log(origineel);  
// "Ik ben niet muteerbaar"
```

Muteert de originele
data niet,
maakt een kopie en
past aan



Side-effects

- Een side effect is iedere verandering aan de toestand van een applicatie die zichtbaar is buiten de opgeroepen functie (behalve de return waarde).
 - Aanpassen van een externe variable/object
 - Loggen naar de console
 - Schrijven naar het scherm/een bestand/het netwerk
 - Oproepen van een extern process
- Side-effects dienen vermeden te worden in functional programming. Dit zorgt voor verstaanbare code die makkelijker te testen valt.

Declaratief vs Imperatief

- Imperatief
 - Focust op '**hoe**' een programma functioneert. Bestaat uit een beschrijving van de verschillende uit te voeren stappen om een resultaat te bereiken: **flow control**.
 - Programma bevat veel details
- Declaratief
 - Focust op '**wat**' een programma moet bekomen, zonder te specificeren hoe dit moet bekomen worden (meer black box). Beschrijving van de **data-flow**
 - Maakt gebruik van bestaande functies om de complexiteit te verminderen

Declaratief vs Imperatief

- Imperatief

```
const arr = ["een", "twee", "drie"];

function zoek(waarde) {
    for (let i = 0; i < arr.length; i++) {
        if(arr[i] === waarde)
            return i;
    }

    return -1;
}
```

```
zoek("twee"); //1
zoek("zes"); //-1
```

- Declaratief

```
const arr = ["een", "twee", "drie"];

/* We maken gebruik van de indexOf
methode van een array.
Hoe deze te werk gaat maakt ons niet
uit. */
```

```
arr.indexOf("twee"); //1
arr.indexOf("zes"); //-1
```

Samenvatting

- Functioneel programmeren staat voor
 - Pure functies zonder shared state en side-effects
 - Onveranderlijke data tegen over veranderlijke data
 - Declaratieve stijl boven imperatieve stijl
- Dit komt vooral naar voor bij de ES6 array functies
 - Map
 - Filter
 - Reduce

04 Functional Programming met Arrays

Arrays

Arrays - Herhaling

- Pas in index.html van 04thCollectionsStarter de link aan naar herhaling.js

```
<script src="js/herhaling.js"></script>
```

Arrays - Herhaling

```
// Een lege array creëren
let leeg1 = new Array();
let leeg2 = [];

// Initiële elementen opgeven
let fruit = ['apple', 'pear', 'lemon'];

// Individuele elementen gebruiken
console.log(fruit[1]); // pear

// Een element vervangen
fruit[2] = 'kiwi';

// Een nieuw element toevoegen
fruit[3] = 'grape';

// Het aantal elementen weergeven
console.log(fruit.length); // 4

// De ganse array tonen
console.log(fruit); // ["apple", "pear", "kiwi", "grape"]
```

Arrays - Herhaling

// Een array kan elementen van verschillende types bijhouden

```
let arr = [  
  'apple',  
  { firstname: 'Jan', lastname: 'Janssens' },  
  true,  
  function() {  
    console.log(`Hello!`);  
  }  
];
```

// de firstname laten zien van het element op positie 1

```
console.log(arr[1].firstname); // Jan
```

// de functie gebruiken op positie 3

```
arr[3](); // Hello!
```


Arrays - Herhaling

```
// pop verwijdt het laatste element en retourneert het  
console.log(fruit.pop()); // grape
```

```
// push voegt een nieuw element achteraan toe  
fruit.push('melon');  
console.log(fruit); // ["apple", "pear", "kiwi", "melon"]
```

```
// shift verwijdt het eerste element en retourneert het  
console.log(fruit.shift()); // apple
```

```
// met unshift kan je een element vooraan de array toevoegen  
fruit.unshift('orange');  
console.log(fruit); // ["orange", "pear", "kiwi", "melon"]
```

Arrays – Herhaling – Lussen

```
// De klassieke manier  
for (let i = 0; i < fruit.length; i++) {  
    console.log(fruit[i]);  
}
```

```
// Nog een manier met behulp van for-of  
for(let element of fruit){  
    console.log(element);  
}
```

```
// orange  
// pear  
// kiwi  
// melon
```

Arrays – Herhaling

```
// Elementen verwijderen
// Verwijder het element op positie 1
delete fruit[1];
console.log(fruit); // ["orange", empty, "kiwi", "melon"]

// De functie splice
// Verwijder 2 elementen vertrekkend van positie 1 en voeg "pineapple",
// "strawberry", "blueberry" in
// De verwijderde elementen worden geretourneerd
console.log(fruit.splice(1, 2, 'pineapple', 'strawberry', 'blueberry'));
// [empty, "kiwi"]
console.log(fruit);
// ["orange", "pineapple", "strawberry", "blueberry", "melon"]

// De functie slice retourneert een nieuwe array waarbij alle items
// gekopieerd worden
// vanaf de startindex tot (niet tot en met) de eindindex
console.log(fruit.slice(2, 5)); // ["strawberry", "blueberry", "melon"]
```

Arrays – Herhaling

```
// Zoeken in een array
// De functie indexOf(item, from) zoekt naar item startend van positie
// from (default waarde 0)
// en retourneert de index waar het gezochte item gevonden werd. Anders
// wordt er -1 geretourneerd
console.log(fruit.indexOf('blueberry')); // 3
console.log(fruit.indexOf('orange')); // -1

// De functie lastIndexOf(item, from) doet hetzelfde maar zoekt van
// rechts naar links
console.log(fruit.lastIndexOf('blueberry')); // 3
console.log(fruit.lastIndexOf('orange')); // -1

// De functie includes(item, from) zoekt naar item startend van positie
// from en retourneert true wanneer het gezochte item werd gevonden
console.log(fruit.includes('blueberry')); // true
console.log(fruit.includes('blueberry', 4)); // false
console.log(fruit.includes('orange')); // false
```

Arrays – Herhaling

```
// De functie reverse keert de volgorde van de elementen in de array om
fruit.reverse();
console.log(fruit);
// ["strawberry", "pineapple", "blueberry", "orange", "melon"]
```

```
// De functie split splitst de meegegeven string op in stukken
// op basis van het opgegeven scheidingsteken
let namen = 'Bilbo, Gandalf, Nazgul';
let arrNamen1 = namen.split(',');
console.log(arrNamen1); // ["Bilbo", " Gandalf", " Nazgul"]
```

```
// De split methode heeft een optioneel tweede argument,
// namelijk de maximumlengte van de array
// Als dit tweede argument opgegeven wordt,
// worden alle extra elementen genegeerd;
let arrNamen2 = namen.split(',', 2);
console.log(arrNamen2); // ["Bilbo", " Gandalf"]
```

```
let str = 'test';
console.log(str.split('')); // ["t", "e", "s", "t"]
```

Arrays – Herhaling

```
// De functie join is de omgekeerde bewerking.  
// De functie join creëert een join waarbij  
// de items gescheiden worden door het opgegeven scheidingsteken  
let arrNamen3 = ['Bilbo', 'Gandalf', 'Nazgul'];  
let strNamen3 = arrNamen3.join(';');  
console.log(strNamen3); // Bilbo;Gandalf;Nazgul
```

Callback functions

- ES6 voorziet een aantal geavanceerde methodes voor arrays.
- Deze werken met het concept van een callback functie
- Een callback functie, is een functie die wordt uitgevoerd **NADAT** een andere functie klaar is.
- Dit zagen we reeds kort bij het afhandelen van event

```
button.addEventListener('click', callbackFunction);
```

`callbackFunction` wordt uitgevoerd NADAT de `click` functie klaar is

**HO
GENT**

Callback functions

```
function doHomework(subject, callback) {  
  console.log(`Starting my ${subject} homework.`);  
  callback();  
}
```

Een functie wordt aangemaakt die een callback functie als parameter verwacht.

```
function alertFinished(){  
  console.log('Finished my homework');  
}
```

De functie eindigt met het oproepen van de callback functie

Definitie van de callback functie

```
doHomework('math', alertFinished);
```

Oproepen van de originele functie, die een 2^{de} (callback)functie meegeeft als argument – deze wordt niet direct opgeroepen

Map – Filter – Reduce

- Map, Filter en Reduce zijn geavanceerde methodes van de Array die de functionele programmeerstijl onderschrijven aan de hand van een callback functie
- Map
 - Als je een bewerking wil toepassen op ieder element van een array en een bewerkte kopie van de originele array terug wil krijgen.
- Filter
 - Als je al een array hebt en je wil de elementen uit de array die aan bepaalde criteria voldoen
- Reduce
 - Als je al een array hebt en je wil de elementen uit de array gebruiken om iets nieuws te berekenen

Map – Filter - Reduce

- Deze geavanceerde methodes verwachten dat we een callback functie meegeven
- De callback functie wordt opgeroepen voor ieder item in de array
- Bij iedere iteratie krijgt de functie automatisch een aantal argumenten mee
 - Value – de huidige waarde tijdens de iteratie
 - Index – de huidige index (teller) van de iteratie
 - Array – een kopie van de hele array

```
arr.map(callbackFunctie);  
  
function callbackFunctie(value, index, array) {  
}
```

```
//de inline versie met een anonieme functie  
arr.map(function(value, index, array) {  
  
});
```

```
//de arrow notatie  
arr.map((value, index, array) => {});
```

Map – Filter - Reduce

- Pas in index.html van 04thCollectionsStarter de link aan naar mapFilterReduce.js

```
<script src="js/mapFilterReduce.js"></script>
```

Map – Filter - Reduce

- Alle voorbeelden zijn gebaseerd op de volgende data:

(index)	name	size	weight
0	"cat"	"small"	5
1	"dog"	"small"	10
2	"lion"	"medium"	150
3	"elephant"	"big"	5000

```
const animals = [  
  {  
    name: 'cat',  
    size: 'small',  
    weight: 5  
  },  
  {  
    name: 'dog',  
    size: 'small',  
    weight: 10  
  },  
  {  
    name: 'lion',  
    size: 'medium',  
    weight: 150  
  },  
  {  
    name: 'elephant',  
    size: 'big',  
    weight: 5000  
  }  
];
```

Map

```
// Voorbeeld 1: We willen een array met de namen van de dieren
// for - lus
let animal_names_1 = [];
for (let i = 0; i < animals.length; i++) {
    animal_names_1.push(animals[i].name);
}
console.log(animal_names_1); // ["cat", "dog", "lion", "elephant"]

// map
let animal_names_2 = animals.map(callbackFunction);

function callbackfunction(value, index, array) {
    return value.name;
}
console.log(animal_names_2); // ["cat", "dog", "lion", "elephant"]
```

Map

```
// map - arrow callback functie
let animal_names_2 = animals.map((value, index, array) => {
  return value.name;
});
console.log(animal_names_2); // ["cat", "dog", "lion", "elephant"]

// omdat er geen gebruik gemaakt wordt van index en array
// had je dit ook als volgt kunnen schrijven
let animal_names_3 = animals.map(value => {
  return value.name;
});
console.log(animal_names_3); // ["cat", "dog", "lion", "elephant"]

// of simpelweg
let animal_names_4 = animals.map(value => value.name);
console.log(animal_names_4); // ["cat", "dog", "lion", "elephant"]
```

Filter

```
// Voorbeeld 2: We willen een array met de kleine dieren
// for - lus
let small_animals_1 = [];
for (let i = 0; i < animals.length; i++) {
    if (animals[i].size === 'small') {
        small_animals_1.push(animals[i]);
    }
}
console.log(small_animals_1); // [{name: "cat", size: "small", weight: 5}, {name:
"dog", size: "small", weight: 10}]

// filter
let small_animals_2 = animals.filter((value, index, array) => {
    return value.size === 'small';
});
console.log(small_animals_2); // [{name: "cat", size: "small", weight: 5}, {name:
"dog", size: "small", weight: 10}]

// omdat er geen gebruik gemaakt wordt van index en array
// had je dit ook als volgt kunnen schrijven
let small_animals_3 = animals.filter(value => {
    return value.size === 'small';
});
console.log(small_animals_3); // [{name: "cat", size: "small", weight: 5}, {name:
"dog", size: "small", weight: 10}]
```

Reduce

```
// Voorbeeld 3: We willen de totale som van de gewichten
// van de dieren kennen
// for - lus
let total_weight_1 = 0;
for (let i = 0; i < animals.length; i++) {
    total_weight_1 += animals[i].weight;
}
console.log(total_weight_1); // 5165

// reduce
let total_weight_2 = animals.reduce((result, value, index, array) => {
    return (result += value.weight);}, 0);
console.log(total_weight_2); // 5165

// omdat er geen gebruik gemaakt wordt van index en array
// had je dit ook als volgt kunnen schrijven
let total_weight_3 = animals.reduce((result, value) => {
    return (result += value.weight);}, 0);
console.log(total_weight_3); // 5165
```


Reduce – Hoe werkt dit?

Iteratie	Huidig result	Value (huidige waarde)	Nieuw Result
1	0	5	5
2	5	10	15
3	15	150	165
4	165	5000	5165

- Hoe kan je de code omvormen zodat het gewicht van de olifant niet meegerekend wordt?

```
let total_weight_4 = animals.reduce((result, value) => {  
  return (value.name === 'elephant' ? result : result + value.weight);  
}, 0);  
console.log(total_weight_4); // 165
```

04 Functional Programming met Arrays

Geavanceerde methodes

forEach

generische functie die over een array itereert
en een **callback** functie aanroept tijdens elke iteratie

callback

Function to execute on each element. It accepts between one and three arguments:

currentValue


The current element being processed in the array.

index | Optional

The index **currentValue** in the array.

array | Optional

The array **forEach()** was called upon.



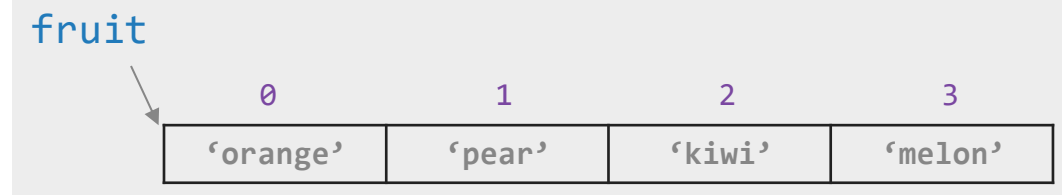
Argumenten worden
positioneel doorgegeven
aan de parameters van de
callback functie

```
(el) => ...  
(el, i) => ...  
(el, i, arr) => ...
```

de **forEach** retourneert geen resultaat!

Arrays – geavanceerde methodes

```
// De klassieke manier van itereren
for (let i = 0; i < fruit.length; i++) {
    console.log(fruit[i]);
}
```

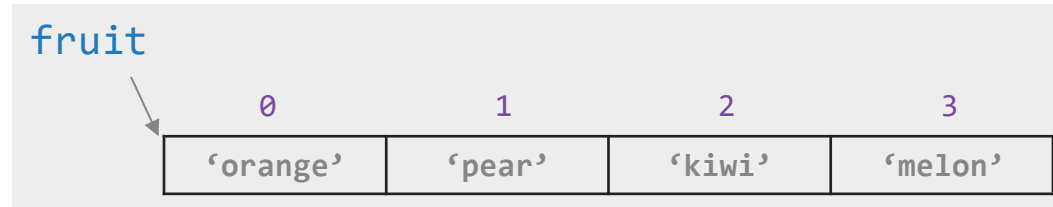


```
// Nog een manier met behulp van de geavanceerde methode: forEach
fruit.forEach(function(element) {
    console.log(element);
});
// orange
// pear
// kiwi
// melon
```

```
// Hetzelfde maar korter met behulp van arrow functies
fruit.forEach((element) => console.log(element));
```

```
// Idem. Als er maar één parameter is moeten er geen ronde haakjes staan
// rond de parameter
fruit.forEach(element => console.log(element));
```

Arrays – geavanceerde methodes



```
// De meest algemene vorm van forEach
fruit.forEach((item, index, array) => {
  console.log(`${item} is at index ${index} in ${array}`);
});
```

```
// orange is at index 0 in orange,pear,kiwi,melon
// pear is at index 1 in orange,pear,kiwi,melon
// kiwi is at index 2 in orange,pear,kiwi,melon
// melon is at index 3 in orange,pear,kiwi,melon
```

original →

'kiwi'	'orange'	'pear'	'mango'
--------	----------	--------	---------

result →

```
result = original.filter(el => el.length > 4)
```

original →

'kiwi'	'orange'	'pear'	'mango'
--------	----------	--------	---------

result →

```
result = original.map(el => el.length)
```

original →

'kiwi'	'orange'	'pear'	'mango'
--------	----------	--------	---------

(acc, el) => {acc: el.length, el: length+ el.length
0 4 10 14

19

result

```
result = original.reduce((acc, el) => acc + el.length, 0)
```

original →

'kiwi'	'orange'	'pear'	'mango'
--------	----------	--------	---------

kiwi
orange
pear
mango

```
original.forEach(el => console.log(el))
```

HO GENT

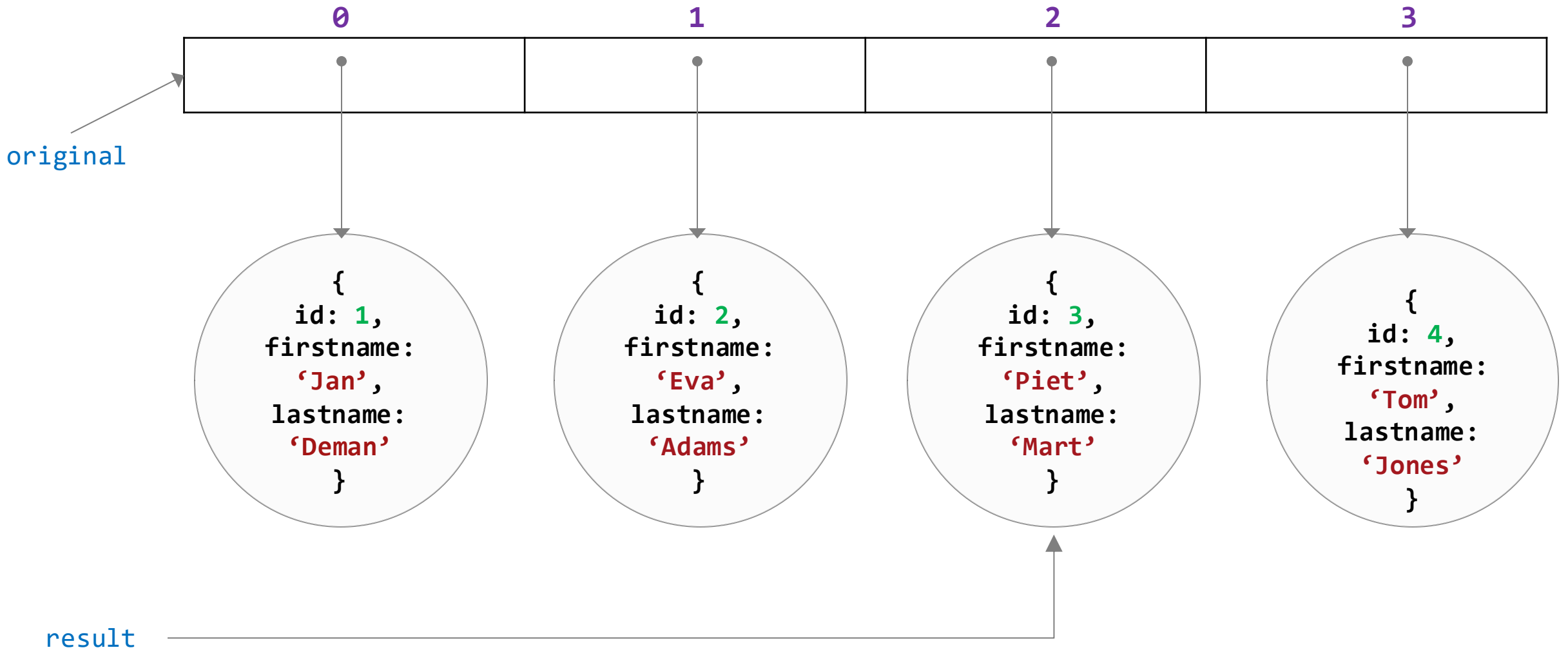
find & findIndex

retourneert (de index van) het eerste element
waarvoor de callback (~voorwaarde) true retourneert

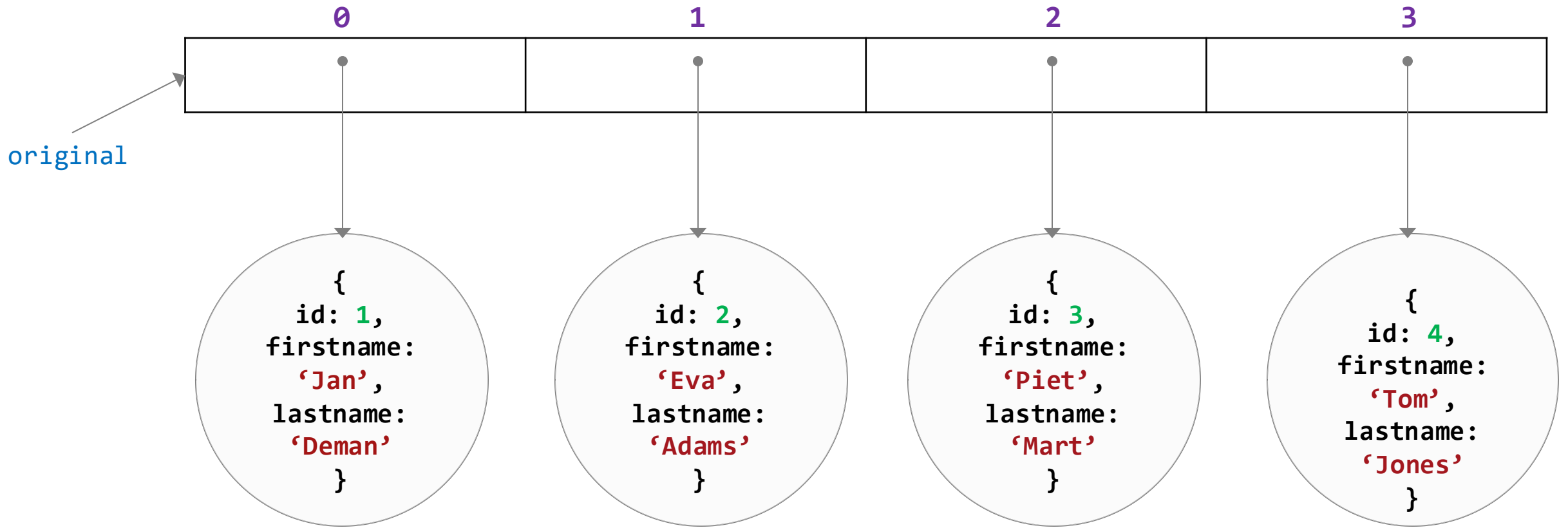
*indien geen enkel element aan de
voorwaarde voldoet dan retourneert
find: undefined & findIndex: -1*



Herinner je nog de **indexOf** methode?



```
result = original.find(p => p.firstname === 'Piet')  
result = original.find(({firstname}) => firstname === 'Piet')
```

result 2

```
result = original.findIndex(p => p.firstname === 'Piet')
```

SORT

callback is een **comparator** die bepaald hoe elementen zullen gesorteerd worden



Indien je geen **comparator** voorziet worden de elementen van de array omgezet naar strings en wordt er 'alfabetisch' gesorteerd

```
function compare(a, b) {  
  if (a is less than b by some ordering criterion) {  
    return -1;  
  }  
  if (a is greater than b by the ordering criterion) {  
    return 1;  
  }  
  // a must be equal to b  
  return 0;  
}
```

de array wordt **in-place** gesorteerd,
sort retourneert de gesorteerde array, dit is dus geen copy van de
originele array



```
result = original.sort()
```



```
result = original.sort((el1, el2) => el1.length - el2.length)
```



```
result = original.sort()
```

Arrays – geavanceerde methodes

- Pas in index.html van 04thCollectionsStarter de link aan naar advanced.js

```
<script src="js/advanced.js"></script>
```

er bestaan nog veel meer handige
array methodes



zie MDN website
some, every, reverse, ...

04 Functional Programming met Arrays

Maps

Maps

- Pas in index.html van 04thCollectionsStarter de link aan naar maps.js

```
<script src="js/maps.js"></script>
```

The **Map** object holds
key-value pairs.

It remembers the original insertion order of the keys.

*Any value (both objects and primitive values) may be used
as either a key or a value.*

Maps

properties

size

methods

set

get

has

delete

clear

keys

values

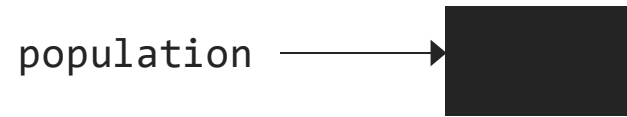
entries

forEach

Maps

- gebruik de constructor `Map()` om een nieuwe Map te creëren

```
const population = new Map();
```



Maps

- `set(key, value)` voegt key-value pair toe of past bestaand key-value pair aan

```
population.set('Belgium', 11589623);  
population.set('Burkina Faso', 3273);  
population.set('Iceland', 341243);  
population.set('Burkina Faso', 20903273);
```

population →

```
▼ 0: {"Belgium" => 11589623}  
    key: "Belgium"  
    value: 11589623  
▼ 1: {"Burkina Faso" => 3273}  
    key: "Burkina Faso"  
    value: 3273  
▼ 2: {"Iceland" => 341243}  
    key: "Iceland"  
    value: 341243
```

Maps

- `set(key, value)` retourneert de aangepaste map, dit maakt **method chaining** mogelijk

population

```
.set('Belgium', 11589623)  
.set('Burkina Faso', 20903273);  
.set('Iceland', 341243);
```

population →

```
▼ 0: {"Belgium" => 11589623}  
   key: "Belgium"  
   value: 11589623  
▼ 1: {"Burkina Faso" => 20903273}  
   key: "Burkina Faso"  
   value: 20903273  
▼ 2: {"Iceland" => 341243}  
   key: "Iceland"  
   value: 341243
```

Maps

- `get(key)` retourneert de value van de entry met de opgegeven een key (undefined voor onbestaande key)

```
let country = prompt('Enter name of country.');
```

```
while (country) {  
    alert(`${population.get(country)} people live in ${country}`);  
    country = prompt('Enter name of country.');
```

```
}
```

127.0.0.1:5500 meldt het volgende

341243 people live in Iceland

127.0.0.1:5500 meldt het volgende

undefined people live in France

OK

population



```
▼ 0: {"Belgium" => 11589623}  
    key: "Belgium"  
    value: 11589623  
▼ 1: {"Burkina Faso" => 20903273}  
    key: "Burkina Faso"  
    value: 20903273  
▼ 2: {"Iceland" => 341243}  
    key: "Iceland"  
    value: 341243
```

Maps

- **key equality** wordt bepaald adhv **===**

```
const primitive1 = 'abcdef';  
const primitive2 = 'abcdef';  
console.log(primitive1 === primitive2);
```

← true

```
const object1 = [1, 2, 3];  
const object2 = [1, 2, 3];  
console.log(object1 === object2);
```

← false

Maps

- via de **size property** kan je weten hoeveel key-value pairs een map bevat

```
alert(`We have data for ${population.size} countries.`);
```

127.0.0.1:5500 meldt het volgende

We have data for 3 countries.

OK

population



```
▼ 0: {"Belgium" => 11589623}  
  key: "Belgium"  
  value: 11589623  
▼ 1: {"Burkina Faso" => 20903273}  
  key: "Burkina Faso"  
  value: 20903273  
▼ 2: {"Iceland" => 341243}  
  key: "Iceland"  
  value: 341243
```

Maps

- **has(key)** retourneert een boolean die aangeeft of een entry met de opgegeven key aanwezig is in de map

```
if (population.has(country))  
    alert(`${population.get(country)} people live in ${country}`);  
else  
    alert(`We have no data for ${country}`);
```

127.0.0.1:5500 meldt het volgende

341243 people live in Iceland

127.0.0.1:5500 meldt het volgende

We have no data for France

OK

population

```
▼ 0: {"Belgium" => 11589623}  
    key: "Belgium"  
    value: 11589623  
▼ 1: {"Burkina Faso" => 20903273}  
    key: "Burkina Faso"  
    value: 20903273  
▼ 2: {"Iceland" => 341243}  
    key: "Iceland"  
    value: 341243
```


Maps

- via de boolse methode **delete(key)** kan je een entry met opgegeven key verwijderen uit een Map

```
if (population.delete(country))  
    alert(`The entry for ${country} was deleted`);  
else  
    alert(` Couldn't not delete ${country}...`);
```

127.0.0.1:5500 meldt het volgende

The entry for Burkina Faso was deleted

127.0.0.1:5500 meldt het volgende

Couldn't not delete France...

OK

population



```
▼ 0: {"Belgium" => 11589623}  
  key: "Belgium"  
  value: 11589623  
▼ 1: {"Iceland" => 341243}  
  key: "Iceland"  
  value: 341243  
▼ 2: {"Iceland" => 341243}  
  key: "Iceland"  
  value: 341243
```

Maps

- gebruik `clear()` om een map in 1 stap te ledigen

```
population.clear();  
alert(`The map has been cleared. It contains ${population.size} entries...`);
```

127.0.0.1:5500 meldt het volgende

The map has been cleared. It contains 0 entries...

OK

population



```
▼ 0: {"Belgium" => 589623}  
  key: "Belgium"  
  value: 589623  
▼ 1: {"Iceland" => 341243}  
  key: "Iceland"  
  value: 341243
```

Maps

- `keys()` retourneert een itereerbaar object
 - dit object bevat alle keys in **insertion order**
 - je kan over dit object itereren met **for .. of** loop

```
message = 'Keys in our map:\n';  
for (const key of population.keys()) {  
    message += `${key}\n`;  
}  
alert(message);
```

127.0.0.1:5500 meldt het volgende

Keys in our map:
Belgium
Burkina Faso
Iceland

OK

population

▼ 0: {"Belgium" => 11589623}
key: "Belgium"
value: 11589623
▼ 1: {"Burkina Faso" => 20903273}
key: "Burkina Faso"
value: 20903273
▼ 2: {"Iceland" => 341243}
key: "Iceland"
value: 341243

Maps

- `values()` retourneert een itereerbaar object met alle values

```
message = 'Values in our map:\n';  
for (const value of population.values()) {  
    message += `${value}\n`;  
}  
alert(message);
```

127.0.0.1:5500 meldt het volgende

Values in our map:

11589623
20903275
341243

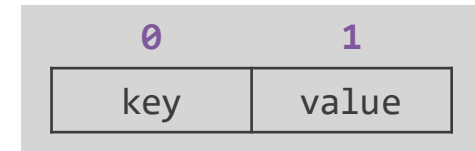
OK

population

▼ 0: {"Belgium" => 11589623}
key: "Belgium"
value: 11589623
▼ 1: {"Burkina Faso" => 20903273}
key: "Burkina Faso"
value: 20903273
▼ 2: {"Iceland" => 341243}
key: "Iceland"
value: 341243

Maps

- **entries()** retourneert een itereerbaar object met alle entries
 - elke entry zit in een array van lengte 2



```
message = 'Entries in our map:\n';  
for (const entry of population.entries()) {  
    message += `${entry[0]} has ${entry[1]} people.\n`;  
}  
alert(message);
```

127.0.0.1:5500 meldt het volgende

Entries in our map:
Belgium has 11589623 people.
Burkina Faso has 20903275 people.
Iceland has 341243 people.

OK

population

```
▼ 0: {"Belgium" => 11589623}  
    key: "Belgium"  
    value: 11589623  
▼ 1: {"Burkina Faso" => 20903273}  
    key: "Burkina Faso"  
    value: 20903273  
▼ 2: {"Iceland" => 341243}  
    key: "Iceland"  
    value: 341243
```

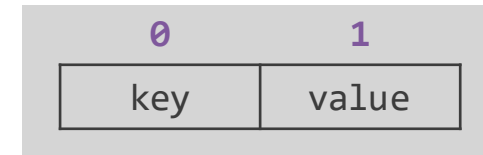
Maps

- tip: maak gebruik van **array destructuring**

```
for (const entry of population.entries()) {  
  message += `${entry[0]} has ${entry[1]} people.\n`;  
}
```

↓

```
for (const [key, value] of population.entries()) {  
  message += `${key} has ${value} people.\n`;  
}
```



Maps

- op map is ook de `forEach(callback)` gedefinieerd

`callback` is invoked with **three arguments**:

- the element's `value`
- the element `key`
- the `Map` object being traversed

```
message = 'Countries with less than 5000000 people:\n'  
population.forEach((value, key) => {  
    if (value < 5000000)  
        message += `${key} with ${value} people\n`;  
});  
alert(message);
```

127.0.0.1:5500 meldt het volgende

Countries with less than 5000000 people:
Iceland with 341243 people

OK

population



```
▼ 0: {"Belgium" => 11589623}  
    key: "Belgium"  
    value: 11589623  
▼ 1: {"Burkina Faso" => 20903273}  
    key: "Burkina Faso"  
    value: 20903273  
▼ 2: {"Iceland" => 341243}  
    key: "Iceland"  
    value: 341243
```

Maps

- constructor `Map()` revisited
 - je kan een **array** argument gebruiken om een map te creëren met een aantal entries
 - in deze array stop je key-value pairs in de vorm `[key, value]`

```
population = new Map([  
    ['Belgium', 11589623],  
    ['Burkina Faso', 20903275],  
    ['Iceland', 341243],  
]);
```


Maps

- voorbeeld: een map met objecten als keys

```
const bel = {  
  name: 'Belgium',  
  region: 'Europe'  
};
```

```
const zim = {  
  name: 'Zimbabwe',  
  region: 'Africa'  
};
```

```
const col = {  
  name: 'Colombia',  
  region: 'Latin America'  
};
```

bel → {name: "Belgium", region: "Europe"}

zim → {name: "Zimbabwe", region: "Africa"}

col → {name: "Colombia", region: "Latin America"}

Maps

- voorbeeld: een map met objecten als keys

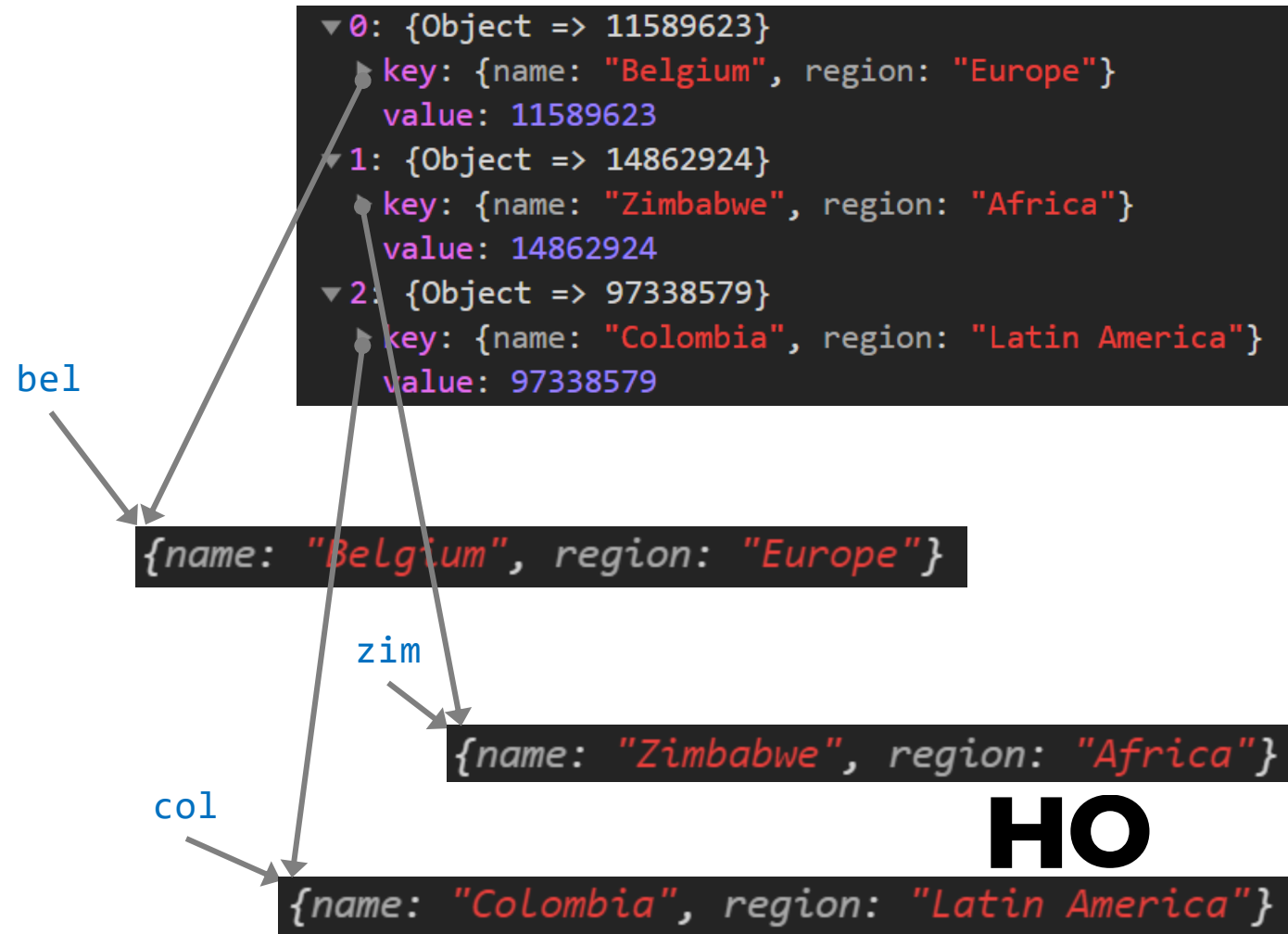
```
population.set(bel, 11589623);  
population.set(zim, 14862924);  
population.set(col, 97338579);
```

```
alert(` ${population.get(col)} live in  
    ${col.name} ` );
```

127.0.0.1:5500 meldt het volgende

97338579 live in Colombia

OK



HO

Maps

- let op voor key equality

```
const col2 = {  
  name: 'Colombia',  
  region: 'Latin America'  
};
```

col2

```
{name: "Colombia", region: "Latin America"}
```

```
alert(`${population.get(col2)} live in ${col2.name}`);
```

127.0.0.1:5500 meldt het volgende
undefined live in Colombia

OK

bel

```
{name: "Belgium", region: "Europe"}
```

zim

```
{name: "Zimbabwe", region: "Africa"}
```

col

```
{name: "Colombia", region: "Latin America"}
```

```
▼ 0: {Object => 11589623}  
  key: {name: "Belgium", region: "Europe"}  
  value: 11589623  
▼ 1: {Object => 14862924}  
  key: {name: "Zimbabwe", region: "Africa"}  
  value: 14862924  
▼ 2: {Object => 97338579}  
  key: {name: "Colombia", region: "Latin America"}  
  value: 97338579
```

HO

Maps

object literal

keys zijn strings (of symbols)

map

keys kunnen eender wat zijn

size property

makkelijk itereerbaar met
insertion order garantie (bv.
forEach)

performant toevoegen en
verwijderen

**HO
GENT**

04 Functional Programming met Arrays

Sets

Sets

- Pas in index.html van 04thCollectionsStarter de link aan naar sets.js

```
<script src="js/sets.js"></script>
```

The **Set** object holds
unique values.

*It remembers the **original insertion order** of the values.*

*Any value (**both objects and primitive values**) may be used.*

Sets

properties

size

methods

add

has

delete

clear

values

entries

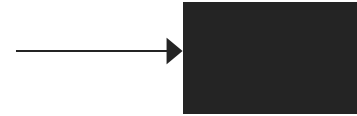
forEach

Sets

- gebruik de constructor `Set()` om een nieuwe Set te creëren

```
let viewers = new Set();
```

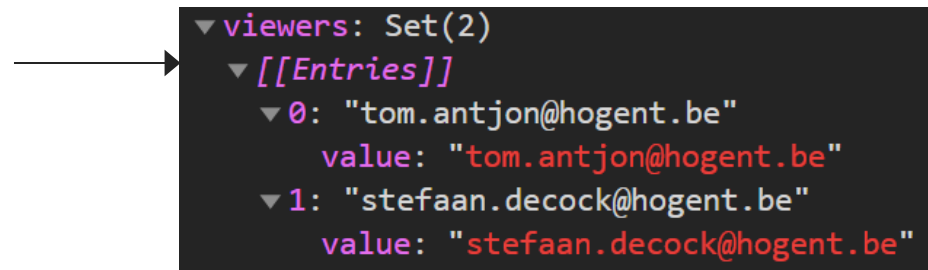
viewers



- of, geef eventueel via een array de initiële waarden op

```
let viewers = new Set(["tom.antjon@hogent.be", "stefaan.decock@hogent.be"]);
```

viewers



**HO
GENT**


Sets

- `add(value)` indien de value nog niet aanwezig is in de set dan wordt value toegevoegd aan de set; de methode retourneert de al dan niet gewijzigde set

viewers

```
.add('patrick.lauwaerts@hogent.be')  
.add('stefaan.samyn@hogent.be')  
.add('pieter.vanderhelst@hogent.be')  
.add('benjamin.vertonghen@hogent.be')  
.add('patrick.lauwaerts@hogent.be');
```

viewers



```
▼ viewers: Set(6)  
  ▼ [[Entries]]  
    ► 0: "tom.antjon@hogent.be"  
    ► 1: "stefaan.decock@hogent.be"  
    ► 2: "patrick.lauwaerts@hogent.be"  
    ► 3: "stefaan.samyn@hogent.be"  
    ► 4: "pieter.vanderhelst@hogent.be"  
    ► 5: "benjamin.vertonghen@hogent.be"
```

Sets

- **size-property** bevat het aantal values in de set


```
alert(`We have now ${viewers.size} unique viewers...`);
```

127.0.0.1:5500 meldt het volgende

We have now 6 unique viewers...

OK

viewers



```
▼ viewers: Set(6)  
  ▼ [[Entries]]  
    ► 0: "tom.antjon@hogent.be"  
    ► 1: "stefaan.decock@hogent.be"  
    ► 2: "patrick.lauwaerts@hogent.be"  
    ► 3: "stefaan.samyn@hogent.be"  
    ► 4: "pieter.vanderhelst@hogent.be"  
    ► 5: "benjamin.vertonghen@hogent.be"
```

Sets

- **has(value)** retourneert een boolean die aangeeft of de opgegeven value aanwezig is in de set
 - er op dezelfde manier als bij Maps gebruik gemaakt van **===**

```
viewer = prompt('Who do you want to follow up?', 'email');  
while (viewer) {  
    alert(`${viewer} has${viewers.has(viewer) ? '' : ' not'} watched this video.`);  
    viewer = prompt('Who else do you want to follow up?', 'email');  
}
```

127.0.0.1:5500 meldt het volgende
benjamin.vertonghen@hogent.be has watched this video.

127.0.0.1:5500 meldt het volgende
Tom.Antjon@hogent.be has not watched this video.

OK

viewers →

```
▼ viewers: Set(6)  
  ▼ [[Entries]]  
    ► 0: "tom.antjon@hogent.be"  
    ► 1: "stefaan.decock@hogent.be"  
    ► 2: "patrick.lauwaerts@hogent.be"  
    ► 3: "stefaan.samyn@hogent.be"  
    ► 4: "pieter.vanderhelst@hogent.be"  
    ► 5: "benjamin.vertonghen@hogent.be"
```

Sets

- via de boolse methode `delete(value)` kan je een value verwijderen uit een Set

```
viewer = prompt('Who do you want to remove from the set of viewers?', 'email');
while (viewer) {
    alert(`${viewer} was ${viewers.delete(viewer) ? '' : 'not'} removed.`);
    viewer = prompt('Who else do you want to remove?', 'email');
}
```

127.0.0.1:5500 meldt het volgende
pieter.vanderhelst@hogent.be was removed.

127.0.0.1:5500 meldt het volgende
clever.forever@student.hogent.be was not removed.

OK

gebruik `clear()` om de Set in 1 keer te ledigen

viewers →

```
Set(5)
  [5]
  0: "tom.antjon@hogent.be"
  1: "stefaan.decock@hogent.be"
  2: "patrick.lauwaerts@hogent.be"
  3: "stefaan.samyn@hogent.be"
  4: "benjamin.vertonghen@hogent.be"
  5: "benjamin.vertonghen@hogent.be"
```

Sets

- `values()` retourneert een itereerbaar object met alle values
- `entries()` retourneert een itereerbaar object met alle [value, value] pairs
 - enkel voor compatibiliteit met de Map

```
message = 'All collections can hold a mix of values from different types...\n';
viewers.clear();
viewers.add(5);
viewers.add(['a', 'b', 'c']);
viewers.add(true);
viewers.add('aString');
viewers.add(x => x * 2);
viewers.add(new Map());

for (const viewer of viewers.values()) {
  message += `${typeof viewer}\n`;
}
alert(message);
```

127.0.0.1:5500 meldt het volgende

All collections can hold a mix of values from different types...

number

object

boolean

string

function

object

Sets

- **for .. of** lus kan je ook rechtstreeks op de set gebruiken values

```
message = 'All collections can hold a mix of values from different types...\n';
viewers.clear();
viewers.add(5);
viewers.add(['a', 'b', 'c']);
viewers.add(true);
viewers.add('aString');
viewers.add(x => x * 2);
viewers.add(new Map());

for (const viewer of viewers) {
  message += `${(typeof viewer)}\n`;
}
alert(message);
```

127.0.0.1:5500 meldt het volgende

All collections can hold a mix of values from different types...

number

object

boolean

string

function

object

Sets

- op Set is ook de `forEach(callback)` gedefinieerd

`callback`

Function to execute for each element, taking three arguments:

`currentValue, currentKey`

The current element being processed in the `Set`. As there are no keys in `Set`, the value is passed for both arguments.

`set`

The `Set` object which `forEach()` was called upon.

```
message = 'All strings in the set:\n';  
viewers.forEach((value) => message += typeof value === 'string' ? `${value}\n` : '');  
alert(message);
```

127.0.0.1:5500 meldt het volgende

All strings in the set:
aString

OK

04 Functional Programming met Arrays

Rest & spread syntax

...

Rest en spread syntax

- Pas in index.html van 04thCollectionsStarter de link aan naar restAndSpread.js

```
<script src="js/restAndSpread.js"></script>
```

Spread syntax

- via de **spread syntax** kunnen we een **iterable** 'uitklappen' in afzonderlijke elementen
- deze elementen kunnen dan gebruikt worden
 - als **argumenten bij functie aanroepen**
 - als **elementen van een array** bij de array literal notation
- volgende built-in iterable types kennen we *(merk op dat Object niet in de lijst staat!)*
 - **string**
 - **map**
 - **set**
 - **array**

Spread syntax

- voorbeeld:

```
const numbers = [20, 30, 40, 50];
```

op plaatsen in je script waar je 20, 30, 40, 50
wil gebruiken kan je `...numbers` zetten

- bv. in een functie aanroep

```
Math.max(1, 20, 30, 40, 50, 8);
```


```
Math.max(10, ...numbers, 8);
```

- bv. in een array literal

```
const numbers2 = [-1, 5, 11, 20, 30, 40, 50];
```

```
const numbers2 = [-1, 5, 11, ...numbers];
```

Spread syntax



<code>string</code>	karakters
<code>map</code>	[key, value] pairs
<code>map.keys()</code>	keys
<code>map.values()</code>	values
<code>set</code>	values
<code>set.values()</code>	values
<code>array</code>	elementen

Spread syntax

- voorbeeld: spread syntax & array literals

```
const aString = 'Javascript';  
console.log([...aString]);
```

```
const anArray = ['a', 'b', 'c'];  
console.log([1, 2, ...anArray, 3, 4]);
```

```
const aMap = new Map([  
  ['Belgium', 11589623],  
  ['Burkina Faso', 20903275],  
  ['Iceland', 341243],  
]);  
console.log([1, 2, ...aMap, 3, 4]);
```

```
const aSet = new Set(["tom.antjon@hogent.be", "  
stefaan.decock@hogent.be"]);  
console.log([1, 2, ...aSet, 3, 4]);
```

```
► (10) ["J", "a", "v", "a", "s", "c", "r", "i", "p", "t"]
```

```
► (7) [1, 2, "a", "b", "c", 3, 4]
```

```
▼ (7) [1, 2, Array(2), Array(2), Array(2), 3, 4] ⓘ  
  0: 1  
  1: 2  
  2: (2) ["Belgium", 11589623]  
  3: (2) ["Burkina Faso", 20903275]  
  4: (2) ["Iceland", 341243]  
  5: 3  
  6: 4  
  length: 7
```

```
► (6) [1, 2, "tom.antjon@hogent.be", "stefaan.decock@hogent.be", 3, 4]
```

Spread syntax

- voorbeeld: spread syntax & functie aanroepen

```
const numbers = [-1, 5, 11, 3];  
  
console.log(Math.max(...numbers));  
console.log(Math.max(1, 10, ...numbers, 20, 2));  
console.log(Math.max(...aMap.values()));
```

```
const aMap = new Map([  
  ['Belgium', 11589623],  
  ['Burkina Faso', 20903275],  
  ['Iceland', 341243],  
]);
```

```
11  
20  
20903275
```

Spread syntax

- voorbeeld: arrays samenvoegen

```
const arr1 = ['Jan', 'Piet'];  
const arr2 = ['Joris', 'Korneel'];  
  
// maak een shallow copy die de inhoud van beide arrays bevat:  
const arr12 = [...arr1, ...arr2];  
console.log(arr12); // ["Jan", "Piet", "Joris", "Korneel"]  
  
// voeg aan arr1 de elementen van arr2 toe  
arr1.push(...arr2);  
console.log(arr1); // ["Jan", "Piet", "Joris", "Korneel"]
```


Spread syntax

- Je kan **iterables, zoals maps, sets, ... omvormen tot arrays** om zo gebruik te kunnen maken van de krachtige array-methodes.
 - stap 1: converteer de iterable naar een array via **[...yourMapOrSet]**
 - stap 2: maak gebruik van **array-methodes**
 - stap 3: converteer het resultaat terug naar een map/set via gebruik van de **constructor** waarbij je **de initiële waarden aanlevert via de array**

Spread syntax

- voorbeeld: de keys van een map alfabetisch sorteren

```
console.log(`Before sort:`);
console.log(population);
population = new Map([...population].sort(
  ([key1], [key2]) => {
    if (key1 < key2) return -1;
    if (key1 > key2) return 1;
    return 0;
  }));
console.log(`After sort:`);
console.log(population);
```

Before sort:

```
► Map(3) {"Iceland" => 341243, "Burkina Faso" => 20903275, "Belgium" => 11589623}
```

After sort:

```
► Map(3) {"Belgium" => 11589623, "Burkina Faso" => 20903275, "Iceland" => 341243}
```

Spread syntax

- voorbeeld2: we willen onze map population aanpassen zodat enkel landen met meer dan 5000000 inwoners overblijven

```
console.log(`Original map:`);  
console.log(population);  
  
population = new Map([...population].filter(  
    ([country, population]) => population > 5000000));  
  
console.log(`Map without big countries:`);  
console.log(population);
```

Original map:

```
► Map(3) {"Belgium" => 11589623, "Burkina Faso" => 20903275, "Iceland" => 341243}
```

Map without big countries:

```
► Map(2) {"Belgium" => 11589623, "Burkina Faso" => 20903275}
```

Rest en spread syntax

- voorbeeld3: we willen de inhoud van twee maps samenvoegen

```
console.log('Landen in eerste map:');  
console.log(population);  
console.log('Landen in tweede map:');  
console.log(population2);  
const combinedPopulation = new Map([...population, ...population2]);  
console.log('Alle landen samen in 1 map:');  
console.log(combinedPopulation);
```

Landen in eerste map:

```
► Map(2) {"Belgium" => 11589623, "Burkina Faso" => 20903275}
```

Landen in tweede map:

```
► Map(2) {"Zimbabwe" => 14862924, "Colombia" => 97338579}
```

Alle landen samen in 1 map:

```
► Map(4) {"Belgium" => 11589623, "Burkina Faso" => 20903275, "Zimbabwe" => 14862924, "Colombia" => 97338579}
```

Rest parameter syntax

- via de **rest parameter** syntax kunnen we een onbepaald aantal argumenten aanleveren aan de **parameter van een functie**
 - dit moet de **laatste parameter** van de functie zijn
 - die parameter is een **array** waarin alle 'overige' argumenten worden verzameld

```
function showName(lastname, ...firstnames) {  
  console.log(`De parameter firstnames bevat ${firstnames}`);  
  const i = firstnames.reduce((initials, current) => initials + current[0], '');  
  return `${i} ${lastname}`;  
}
```

```
console.log(showName('Rowling', 'Joanne', 'Kathleen'));  
console.log(showName('Rubens', 'Pieter', 'Paul'));
```

De parameter firstnames bevat

► (2) ["Joanne", "Kathleen"]

JK Rowling

De parameter firstnames bevat

► (2) ["Pieter", "Paul"]

PP Rubens

Rest parameter syntax

- het **rest** pattern laat ook toe het resterende deel van een array vast te nemen in een variabele tijdens **array destructuring**
 - je kan de rest operator enkel bij de **laatste in de rij van variabelen** zetten

```
const [a, ...b] = ['Jan', 'Piet', 'Korneel', 'Steven', 'Maarten'];  
console.log(a);  
console.log(b);
```

Jan

► (4) ["Piet", "Korneel", "Steven", "Maarten"]

the end.