

The background of the slide features a photograph of a modern building with a facade of large, light-colored concrete panels. Some panels are recessed, creating a textured, three-dimensional effect. A person with long brown hair, wearing a green jacket, blue jeans, and a red backpack, is walking from left to right in the lower half of the image. There are some green plants in the bottom left corner. Two white rectangular boxes are placed over the image: one in the top left and one in the middle left.

Web Development II

Hoofdstuk 05 – Functioneel Programmeren
& Collections

**HO
GENT**

Functioneel Programmeren en Collections

Inleiding

Inhoud

- Functioneel programmeren
- Arrays
 - herhaling
 - map - filter - reduce
 - andere functies
- Maps
- Sets
- Rest en spread syntax

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>

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.

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

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.

- een shared state is elke variabele of object die bestaat in een gedeelde scope; of die bestaat als een property van een object die doorgegeven wordt tussen verschillende scopes

```
// using shared state
let mijnGetal = 2;

function verdubbel(){
  mijnGetal *= 2;
}

verdubbel();
console.log(mijnGetal);
```



```
// not using shared state
let mijnGetal = 2;

function verdubbel(getal){
  return getal * 2;
}

mijnGetal = verdubbel(mijnGetal);
console.log(mijnGetal);
```



Mutable vs Immutable data

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.

- een immutable (onveranderlijk) object is een object dat, na creatie, niet meer kan gewijzigd worden.
 - een mutable (veranderlijk) object kan wel gewijzigd worden
 - om wijzigingen te registreren op een immutable object wordt een gewijzigde copy van het object gecreëerd, het oorspronkelijke object blijft ongewijzigd

Mutable vs Immutable objecten

```
// Arrays zijn van nature uit mutable
const getallen = [1, 2, 3];
const getallen2 = getallen;
console.log(getallen); // [1, 2, 3]
console.log(getallen2); // [1, 2, 3]

// er bestaan verschillende Arrays
// methodes die een Array muteren
getallen.push(100);
console.log(getallen); // [1, 2, 3, 100]
console.log(getallen2); // [1, 2, 3, 100]
```



```
// Strings zijn van nature uit immutable
const zin = 'Ik ben onveranderlijk!';
let zin2 = zin;
console.log(zin); // Ik ben onveranderlijk!
console.log(zin2); // Ik ben onveranderlijk!

// er bestaan geen String methodes die een
// String muteren
zin.replace('I', 'Z');
console.log(zin); // Ik ben onveranderlijk!

zin2 = zin.replace('I', 'Z');
console.log(zin); // Ik ben onveranderlijk!
console.log(zin2); // Zk ben onveranderlijk!
```



Side-effects

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.

- 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

Imperatief vs Declaratief

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.

- imperatief – hoe?
 - bestaat typisch uit een beschrijving van de verschillende uit te voeren stappen om een resultaat te bereiken; focus op **control flow**
 - programma bevat veel details
- declaratief – wat?
 - bestaat typisch uit een beschrijving van wat het gewenste resultaat is; focus op **data-flow**
 - maakt gebruik van higher order functions
 - functies doorgeven aan functies
 - programma bevat minder details

Imperatief vs Declaratief

```
const hobbies = ['tv', 'gamen', 'koken'];

function geefIndexVan(arr, waarde) {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === waarde) return i;
  }
  return -1;
}

console.log(geefIndexVan(hobbies, 'koken'));
//2
console.log(geefIndexVan(hobbies, 'fietsen'));
//-1
```



```
const hobbies = ['tv', 'gamen', 'koken'];

console.log(hobbies.indexOf('koken')); //2
console.log(hobbies.indexOf('fietsen')); //-1
```



Samenvatting

- functioneel programmeren staat voor
 - pure functies zonder shared state en side-effects
 - onveranderlijke data in plaats van veranderlijke data
 - declaratieve stijl in plaats van imperatieve stijl
- in dit hoofdstuk maak je kennis met enkele krachtige JS array functies die dit principe omhelzen
 - map
 - filter
 - reduce
 - ...

Functioneel Programmeren en Collections

Callback functions

Callback functions

- ES6 voorziet een aantal geavanceerde methodes voor arrays die werken met het concept van een callback functie
- een callback functie, is een **functie** die je **als argument** doorgeeft aan een andere functie, deze andere functie maakt gebruik van die functie parameter
- dit zagen we reeds kort bij het afhandelen van events
`button.addEventListener('click', callbackFunction);`

Callback functions

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

een functie die een functie als tweede parameter verwacht.

aanroep naar de functie-parameter callback

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

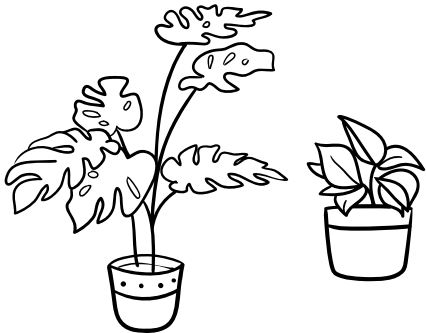
```
doHomework('math', alertFinished);
```

aanroep van de functie doHomework, de functie alertFinished wordt als tweede argument doorgegeven

Arrays - herhaling

- in volgend deeltje gaan we intensief gebruik maken van arrays
- indien je nog even je kennis over arrays wil opfrissen doorloop je best eens de voorbeelden in arraysHerhaling.js
- zorg dat in index.html van 05thFP_en_Collections het script arraysHerhaling.js wordt geladen

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



**HO
GENT**

Functioneel Programmeren en Collections

filter – map - reduce

filter – map - reduce

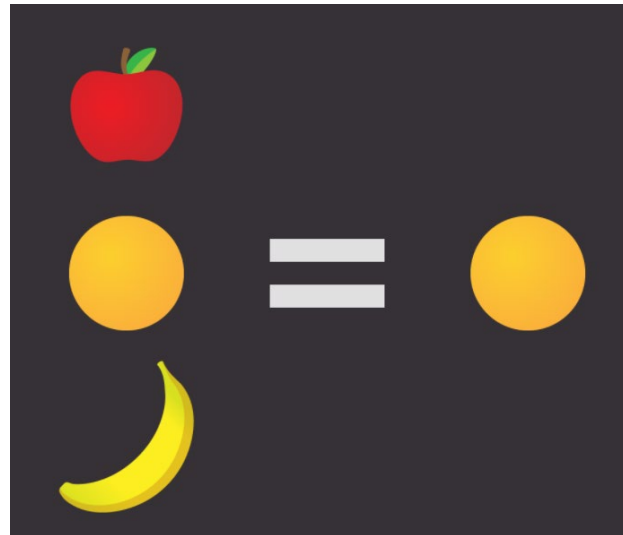
- 3 krachtige en veelgebruikte functies voor arrays die een **pure functionele programmeerstijl** onderschrijven
- hun werking vereist het **itereren** over de elementen van een array maar deze gebeurt **impliciet**
 - we gaan ons niet langer bezighouden met hoe er over de array moet geïtereerd worden, deze functies doen dit voor ons
 - wij gaan focussen op wat we willen bereiken door een geschikte callback functie aan te leveren...

filter – map - reduce

- elk van deze functies verwacht een **callback** functie
 - tijdens het impliciet itereren wordt de callback functie **aangeropen**
 - hierbij worden automatisch **argumenten doorgegeven** aan de callback functie
 - typisch argumenten die kunnen worden doorgegeven, en waar wij dus als **parameters** van onze **callback** functie mee kunnen werken:
 - een **element** uit de array
 - de **index** van een element uit de array
 - de **array** zelf

filter

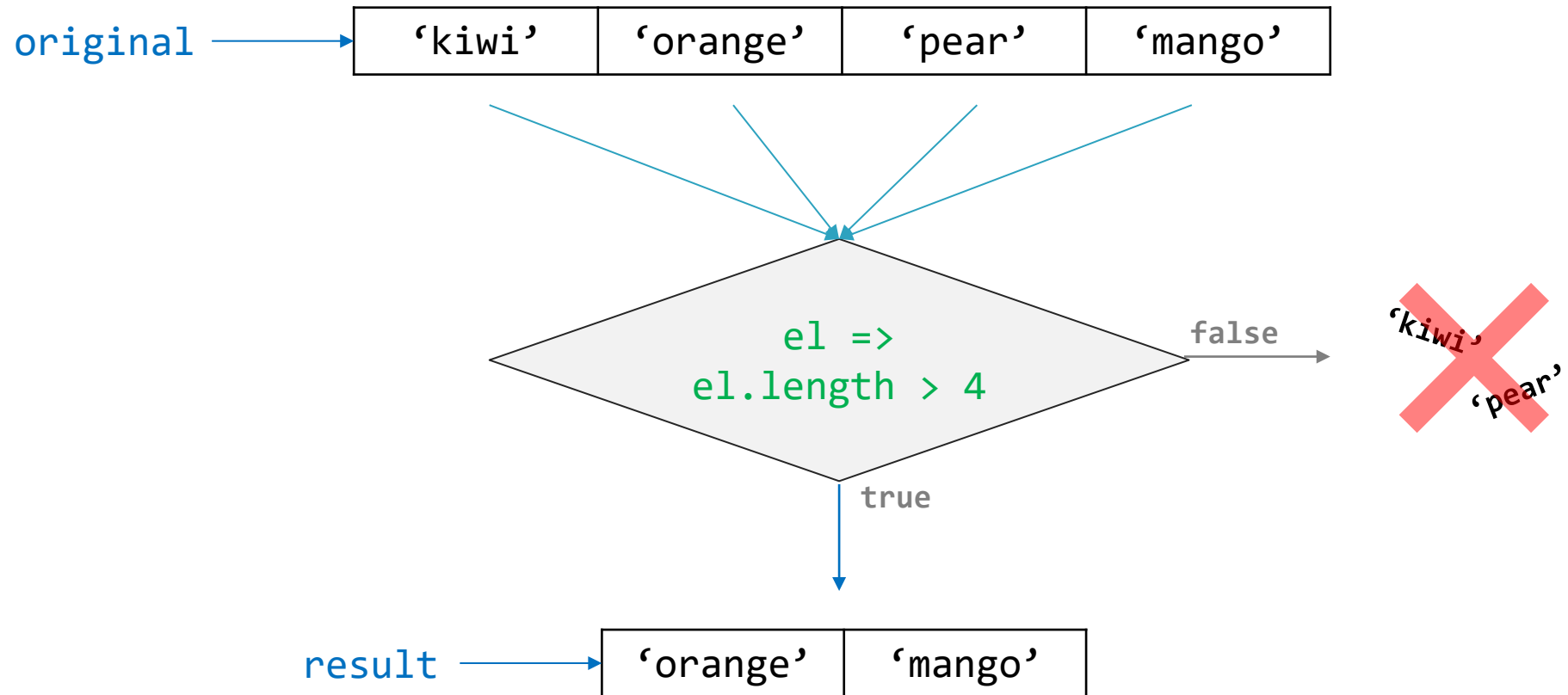
- stelt je in staat om enkel elementen die aan een bepaalde **voorwaarde** voldoen te weerhouden.
- wanneer je een array filtert krijg je als resultaat een **nieuwe array** met enkel de **elementen die aan de filter-voorwaarde voldoen**.



filter

- de **voorwaarde** wordt als een **boolse functie** doorgegeven aan de **filter functie**
 - dit is de **callback** functie
- het itereren over de array gebeurt impliciet
- voor elk element uit de array wordt de callback functie aangeroepen
 - elementen waarvoor de boolse callback functie **true** oplevert komen in het **resultaat** terecht, andere elementen niet

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



filter

- filter roept voor elk element uit de array de callback functie aan met hoogstens **3 argumenten**
 - het **element**
 - de **index** van het element
 - de **array** waarop de filter werd aangeroepen

```
// Arrow function
filter((element) => { /* ... */ })
filter((element, index) => { /* ... */ })
filter((element, index, array) => { /* ... */ })

// Callback function
filter(callbackFn)
filter(callbackFn, thisArg)

// Inline callback function
filter(function(element) { /* ... */ })
filter(function(element, index) { /* ... */ })
filter(function(element, index, array){ /* ... */ })
filter(function(element, index, array) { /* ... */ }, thisArg)
```

Parameters

callbackFn

Function is a predicate, to test each element of the array. Return a value that coerces to `true` to keep the element, or to `false` otherwise.

It accepts three arguments:

element

The current element being processed in the array.

index Optional

The index of the current element being processed in the array.

array Optional

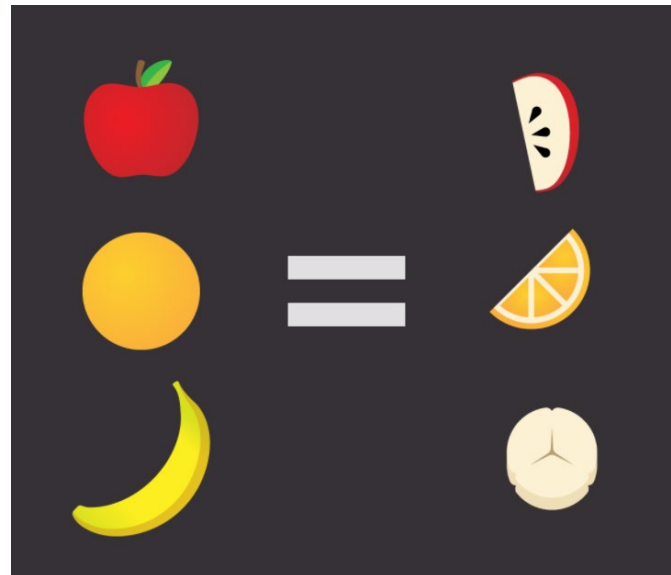
The array on which `filter()` was called.

thisArg Optional

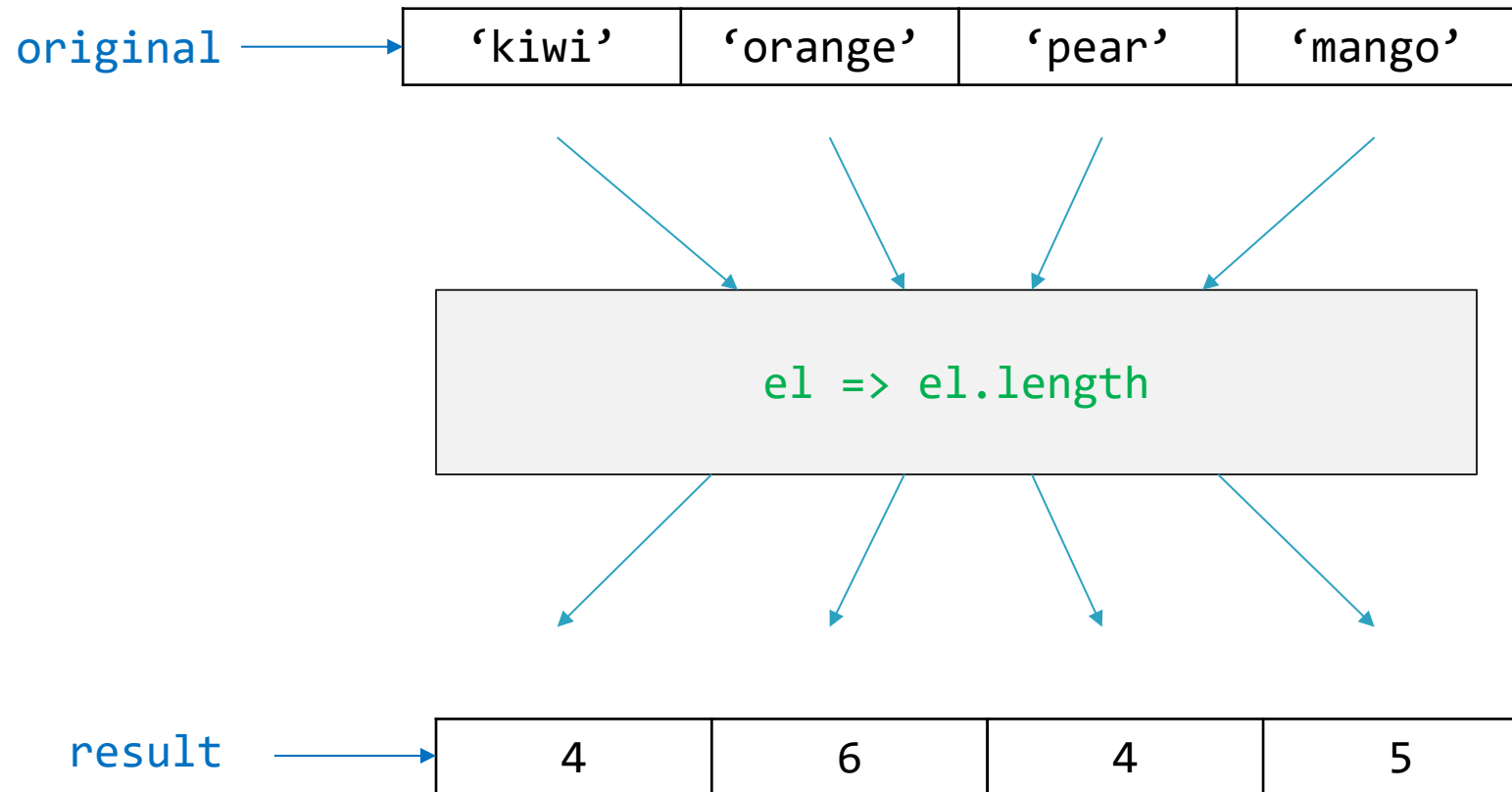
Value to use as `this` when executing `callbackFn`.

map

- stelt je in staat om **elk element** uit een array te **transformeren**
- wanneer je een array mapt krijg je als resultaat een **nieuwe array** die evenveel elementen bevat als de originele array
 - voor elk element uit de originele array wordt een **callback functie** aangeroepen
 - de **return waarden** komen in de resulterende array te zitten




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



map

- analoog aan filter roept map voor elk element uit de array de callback functie aan met hoogstens **3 argumenten**
 - het **element**
 - de **index** van het element
 - de **array** waarop de filter werd aangeroepen

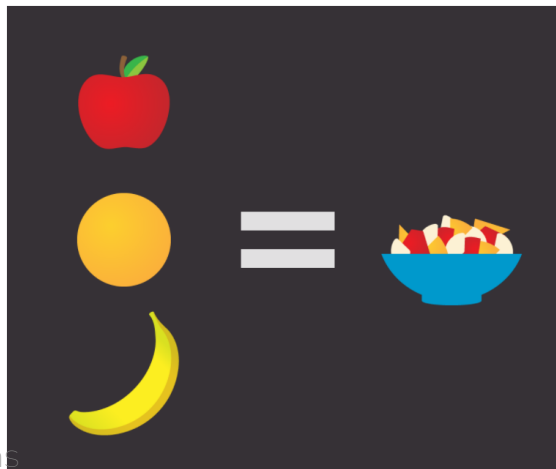
```
// Arrow function
map((element) => { /* ... */ })
map((element, index) => { /* ... */ })
map((element, index, array) => { /* ... */ })

// Callback function
map(callbackFn)
map(callbackFn, thisArg)

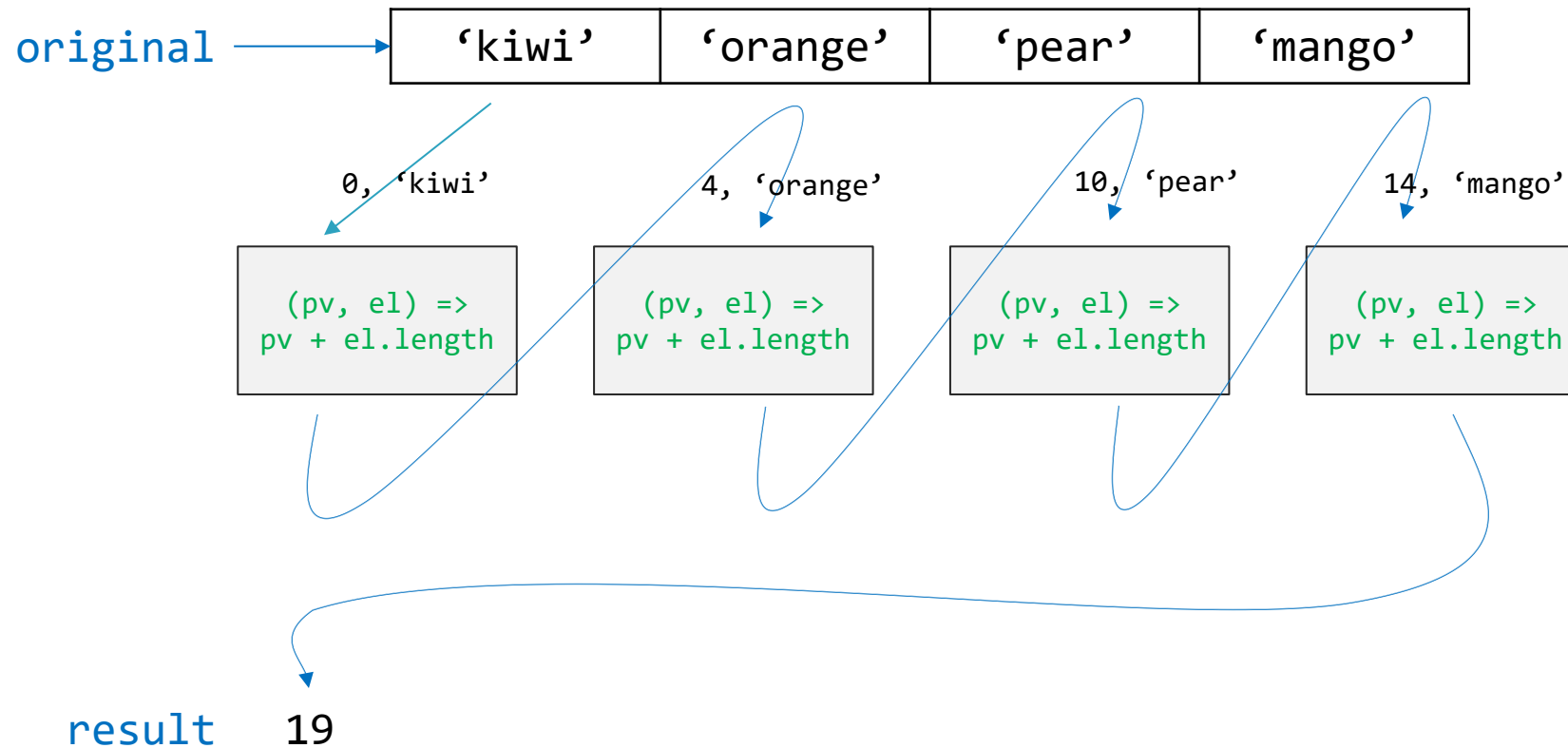
// Inline callback function
map(function(element) { /* ... */ })
map(function(element, index) { /* ... */ })
map(function(element, index, array){ /* ... */ })
map(function(element, index, array) { /* ... */ }, thisArg)
```

reduce

- stelt je in staat om op basis van de elementen in een array **een waarde te 'berekenen'**
 - een impliciete iteratie laat een **callback** functie los op elk element van de originele array
 - de **return waarde** van deze callback wordt van de ene iteratie doorgegeven naar de volgende iteratie
- de resulterende waarde kan van **eender welk type** zijn

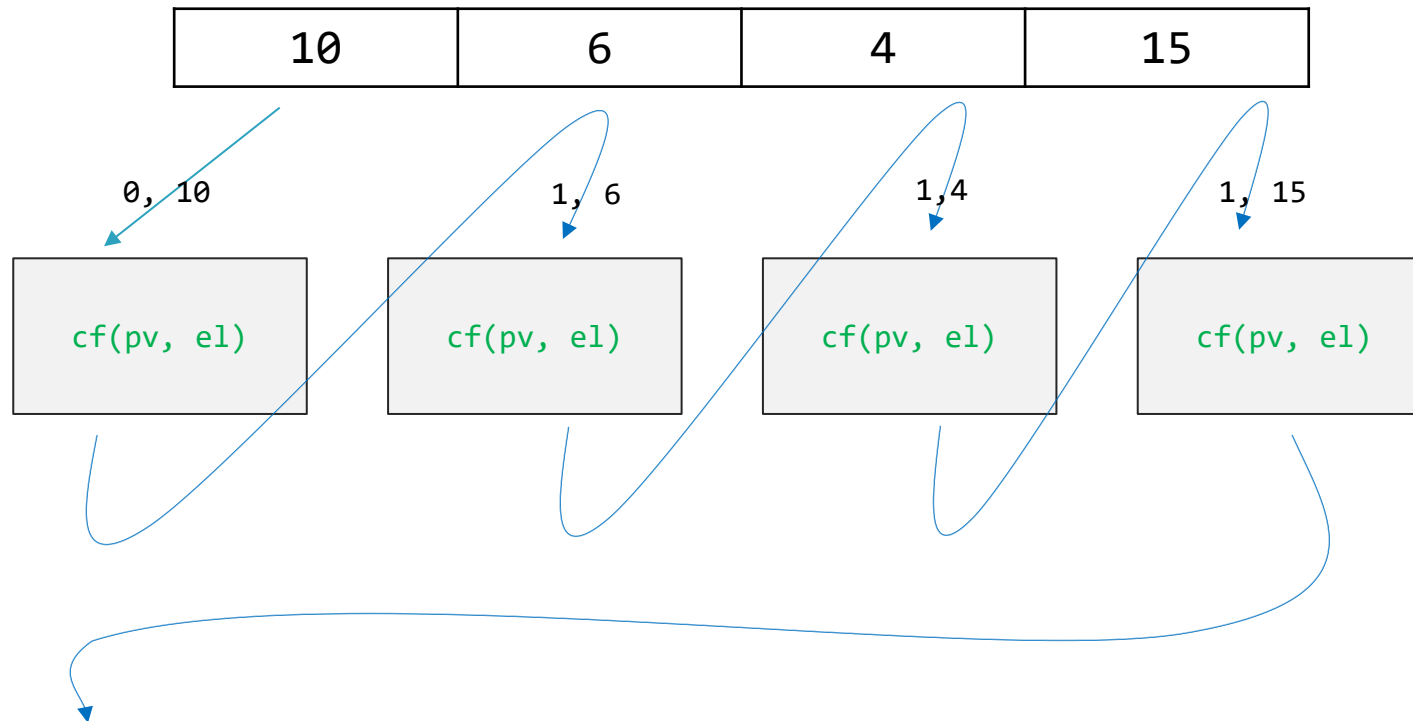


```
result = original.reduce((pv, el) => pv + el.length, 0)
```



```
aantalGeslaagden = [10, 6, 4, 15].reduce(cf, 0)
```

```
const cf = (pv, mark) => pv + (mark >= 10);
```



`aantalGeslaagden` 2

reduce

- merk op dat de functie reduce wordt aangeroepen met 2 argumenten:
 - de **callback** functie
 - de initiele waarde voor de **previousValue** (1^{ste} iteratie)
 - je mag deze weglaten, in dat geval wordt
 - aan de previousValue voor de eerste iteratie het eerste element uit de array doorgegeven
 - begint de iteratie vanaf het tweede element in de array



```
som = [10, 20, 30, 40].reduce((pv, i) => pv + i)
```

```
som = [10, 20, 30, 40].reduce((pv, i) => pv + i, 0)
```

in beide gevallen is som 100

reduce

- merk op dat de callback functie nu als eerste parameter 'previousValue' heeft

```
// Arrow function
reduce((previousValue, currentValue) => { /* ... */ })
reduce((previousValue, currentValue, currentIndex) => { /* ... */ })
reduce((previousValue, currentValue, currentIndex, array) => { /* ... */ })
reduce((previousValue, currentValue, currentIndex, array) => { /* ... */ }, initialValue)

// Callback function
reduce(callbackFn)
reduce(callbackFn, initialValue)

// Inline callback function
reduce(function(previousValue, currentValue) { /* ... */ })
reduce(function(previousValue, currentValue, currentIndex) { /* ... */ })
reduce(function(previousValue, currentValue, currentIndex, array) { /* ... */ })
reduce(function(previousValue, currentValue, currentIndex, array) { /* ... */ },
initialValue)
```

Parameters

callbackFn

A "reducer" function that takes four arguments:

- `previousValue`: the value resulting from the previous call to `callbackFn`. On first call, `initialValue` if specified, otherwise the value of `array[0]`.
- `currentValue`: the value of the current element. On first call, the value of `array[0]` if an `initialValue` was specified, otherwise the value of `array[1]`.
- `currentIndex`: the index position of `currentValue` in the array. On first call, `0` if `initialValue` was specified, otherwise `1`.
- `array`: the array to traverse.

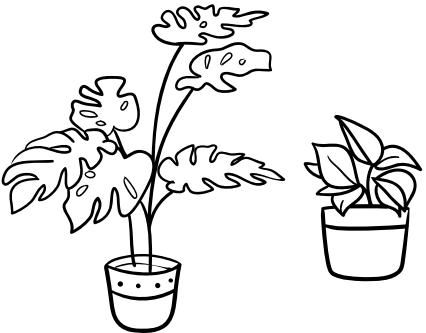
initialValue Optional

A value to which `previousValue` is initialized the first time the callback is called. If `initialValue` is specified, that also causes `currentValue` to be initialized to the first value in the array. If `initialValue` is *not* specified, `previousValue` is initialized to the first value in the array, and `currentValue` is initialized to the second value in the array.

filter – map - reduce

- pas in index.html van 05tFP_en_Collections de link aan naar mapFilterReduce.js

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



**HO
GENT**

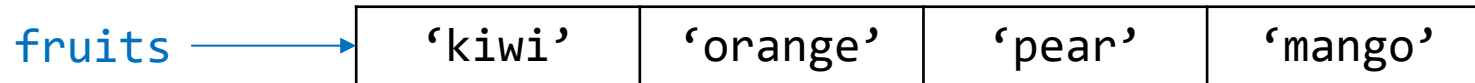
Functioneel Programmeren en Collections

nog enkele array methodes

forEach

- dit is een **algemene methode** die impliciet over een array **itereert** en voor elk element een **callback functie** aanroept
- de methode retourneert geen resultaat
 - undefined
- je kan niet uit de impliciete iteratie springen met break of continue
 - maak hiervoor gebruik van klassieke lus

```
fruits.forEach((el, i, arr) =>  
  console.log(`${el} sits in slot ${i} in an array of ${arr.length} elements`));
```

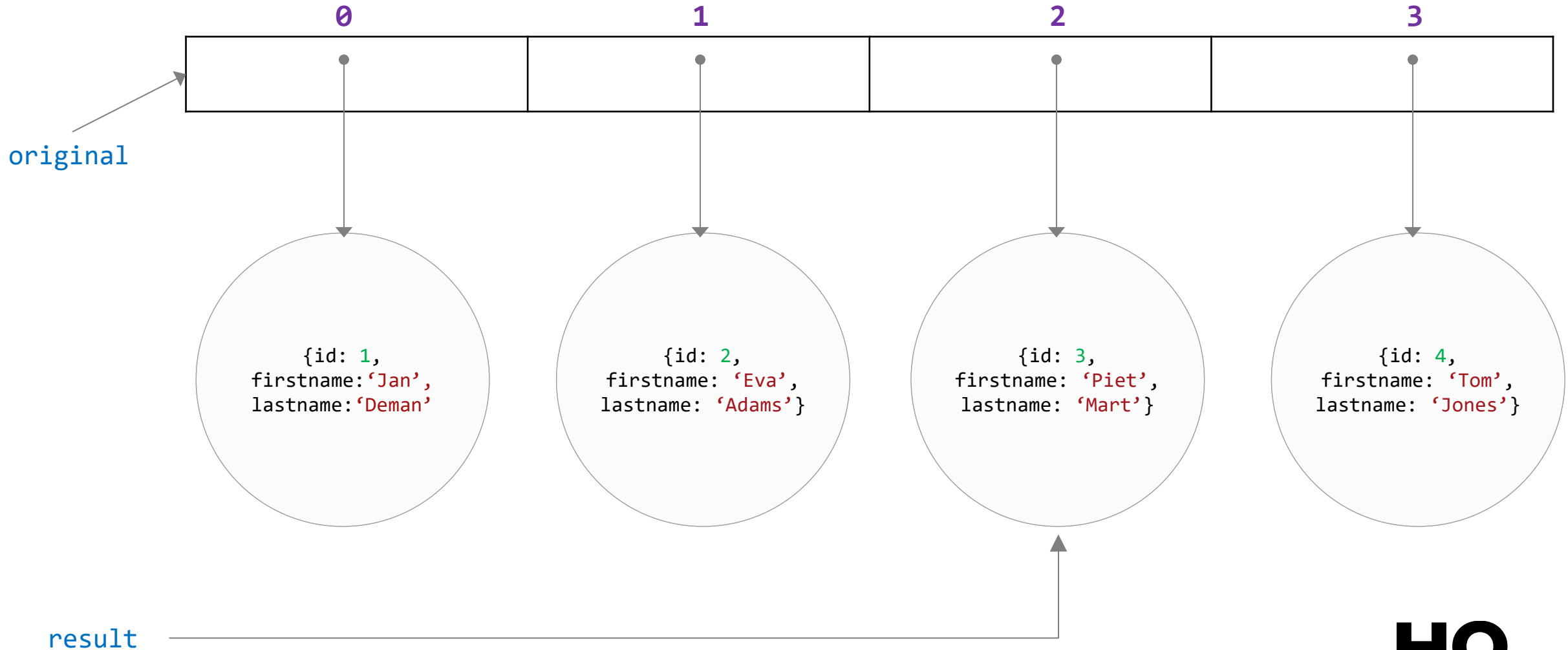


kiwi sits in slot 0 in an array of 4 elements
orange sits in slot 1 in an array of 4 elements
pear sits in slot 2 in an array of 4 elements
mango sits in slot 3 in an array of 4 elements

find

- deze methode retourneert het **eerste element** in een array dat aan een **bepaalde voorwaarde** voldoet
- de voorwaarde wordt aangereikt via een **boolse callback** functie
- indien geen enkel element aan de voorwaarde voldoet dan retourneert de methode **undefined**

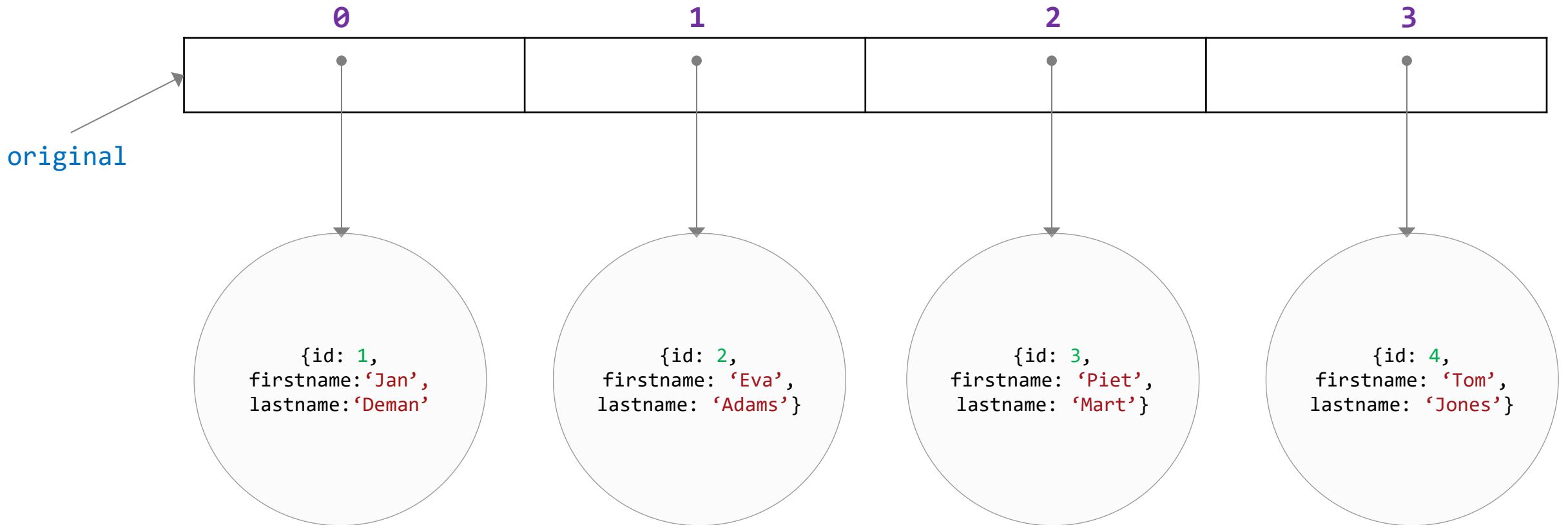
```
result = original.find(p => p.firstname === 'Piet');  
result = original.find(({firstname}) => firstname === 'Piet'); // analoog met object destructuring
```



findIndex

- analoog aan find maar retourneert de index van het eerste **element** in een array die aan een **bepaalde voorwaarde** voldoet
- de voorwaarde wordt aangereikt via een **boolse callback** functie
- indien geen enkel element aan de voorwaarde voldoet dan retourneert de methode **-1**

```
original.findIndex(p => p.firstname === 'Piet');  
original.findIndex(({firstname}) => firstname === 'Piet'); // analoog met object destructuring
```

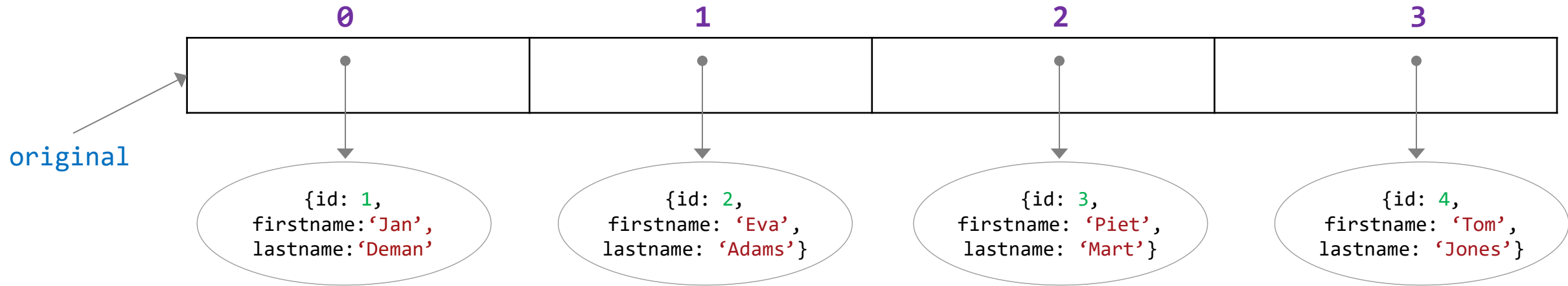


result 2

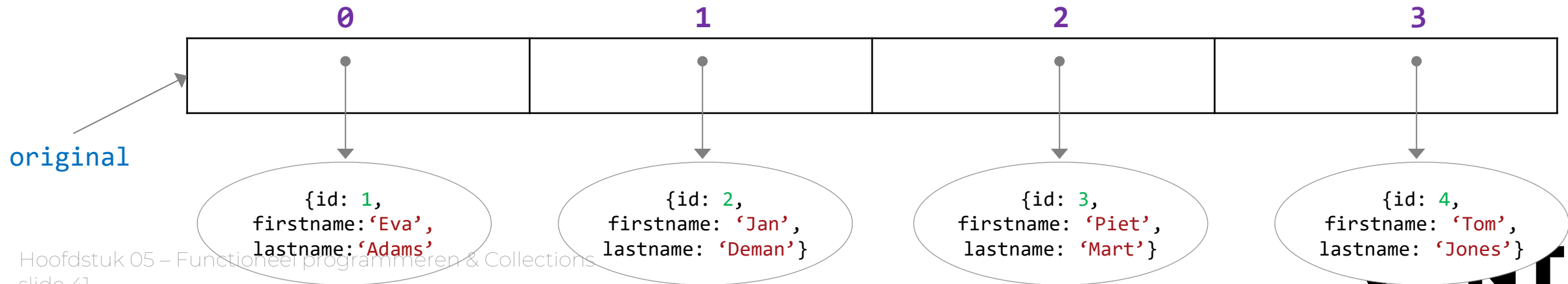
sort

- deze methode **sorteert** een array **in-place** gebruikmakend van een **compare callback functie**
 - hoewel er **geen copy** van de array wordt gemaakt wordt de gesorteerde array ook als resultaat geretourneerd
 - je kan sort aanroepen zonder een compare functie
 - de naar string omgezette versie van elk element van de array wordt gebruikt om de array alfabetisch te sorteren
- **compare functie**

<code>compareFunction(a, b)</code> return value	sort order
<code>> 0</code>	sort <code>b</code> before <code>a</code>
<code>< 0</code>	sort <code>a</code> before <code>b</code>
<code>=== 0</code>	keep original order of <code>a</code> and <code>b</code>

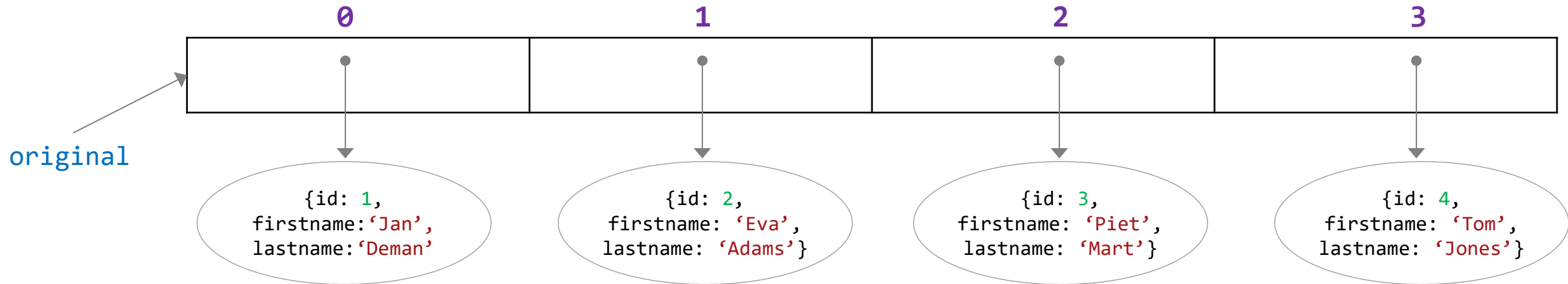


```
original.sort(( { firstname: f1 }, { firstname: f2 } ) => {  
  const p1Name = f1.toUpperCase();  
  const p2Name = f2.toUpperCase();  
  return p1Name < p2Name ? -1 : p1Name > p2Name ? 1 : 0;  
});
```

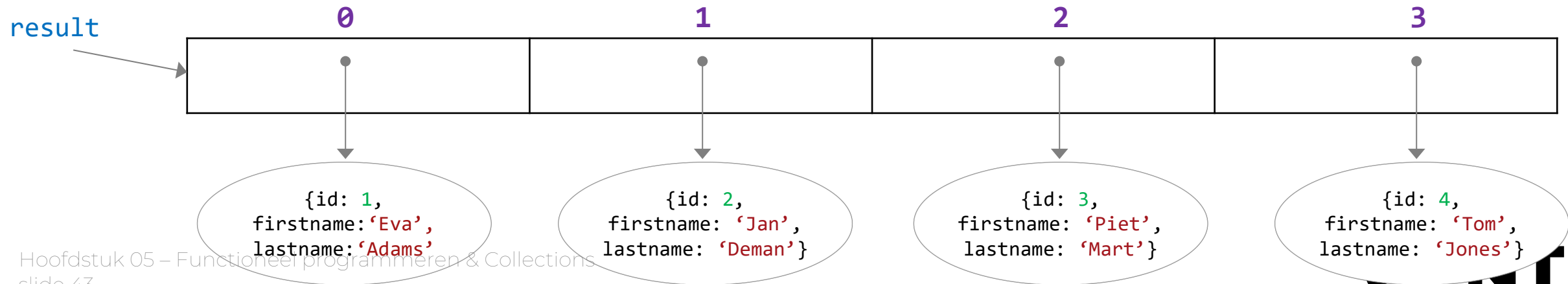


toSorted

- deze methode laat de originele array ongewijzigd en retourneert een gesorteerde **copy** van de array
 - dit leunt meer aan bij het functioneel paradigma daar deze methode geen side effect heeft op de originele array
 - gebruik compare functie identiek als bij sort()



```
const result = original.toSorted(({ firstname: f1 }, { firstname: f2 }) => {  
  const p1Name = f1.toUpperCase();  
  const p2Name = f2.toUpperCase();  
  return p1Name < p2Name ? -1 : p1Name > p2Name ? 1 : 0;  
});
```



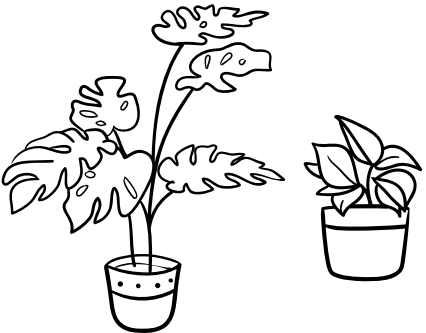
En zo veel meer...

- maak gebruik van MDN om nog veel meer array methodes te ontdekken
- je gaat zelf geen code schrijven om ...
 - de inhoud van een array om te keren!
 - `reverse` (in place) & `toReversed` (copying)
 - te testen of er minstens één element in een array zit die aan een voorwaarde voldoet! `some`
 - te testen of elk element in een array voldoet aan een voorwaarde! `every`
 - een meerdimensionale array om te zetten naar een één dimensionale array! `flat`
 - ...

Arrays – geavanceerde methodes

- Pas in index.html van 05thFP_en_Collections de link aan naar advanced.js

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



**HO
GENT**

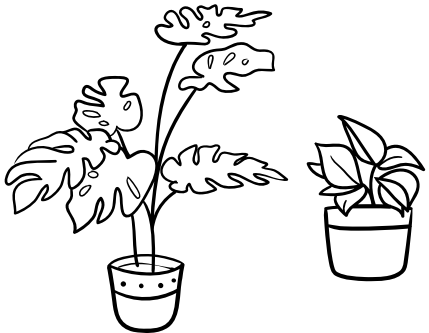
Functioneel Programmeren en Collections

Maps

Maps

- Pas in index.html van 05thFP_en_Collections de link aan naar maps.js

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



**HO
GENT**

Maps

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

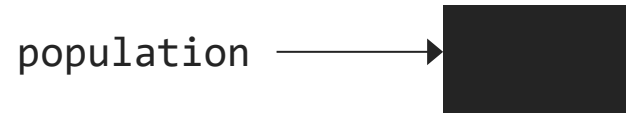
entries

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: 20903273  
▼ 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 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: {"France" => 641243}  
    key: "France"  
    value: 641243
```

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" => 89623}  
  key: "Belgium"  
  value: 89623  
▼ 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



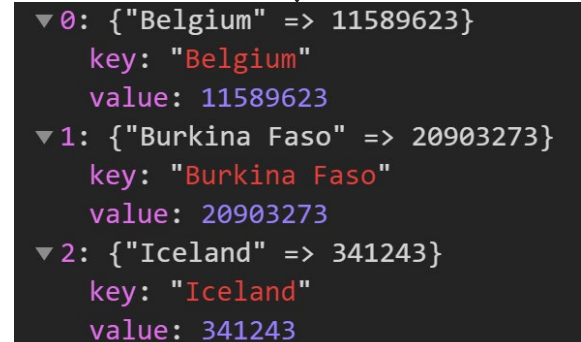
```
message = 'Entries in our map:\n';  
for (const entry of population.entries()) {  
    message += `${entry[0]} has ${entry[1]} people.\n`;  
}  
alert(message);
```

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




A diagram showing a Map structure. An arrow points from the word 'population' to a dark box containing three entries. Each entry is a JavaScript object with a 'key' and a 'value' property. The entries are: 0: {"Belgium" => 11589623}, 1: {"Burkina Faso" => 20903273}, and 2: {"Iceland" => 341243}.

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

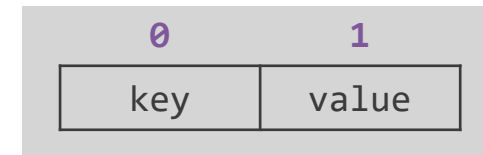
Maps

- 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

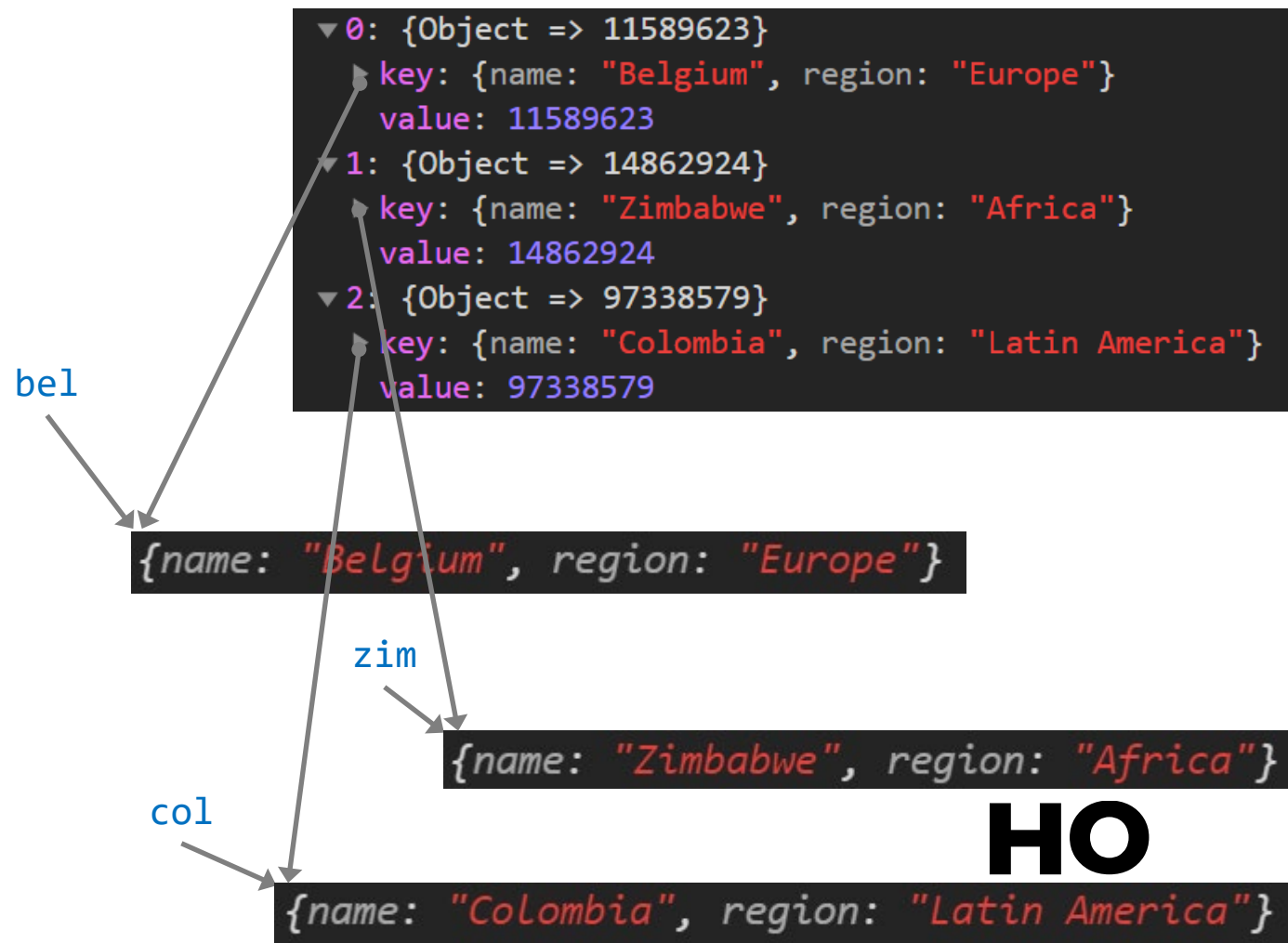
```
population.set(bel, 11589623);  
population.set(zim, 14862924);  
population.set(col, 97338579);
```

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

127.0.0.1:5500 meldt het volgende

97338579 live in Colombia

OK



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

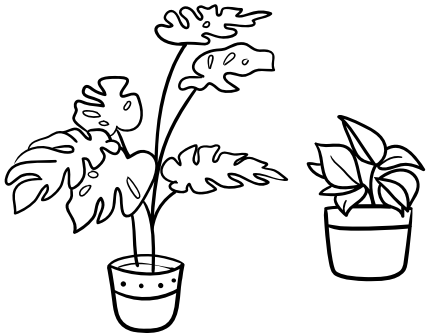
Functioneel Programmeren en Collections

Sets

Sets

- Pas in index.html van 05thFP_en_Collections de link aan naar sets.js

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



**HO
GENT**

Sets

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

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

viewers



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

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

viewers



```
▼ viewers: Set(2)  
  ▼ [[Entries]]  
    ▼ 0: "tom.antjon@hogent.be"  
      value: "tom.antjon@hogent.be"  
    ▼ 1: "stefaan.decock@hogent.be"  
      value: "stefaan.decock@hogent.be"
```

**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 now have ${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 wordt 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)
  ies]]
▶ 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

```
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

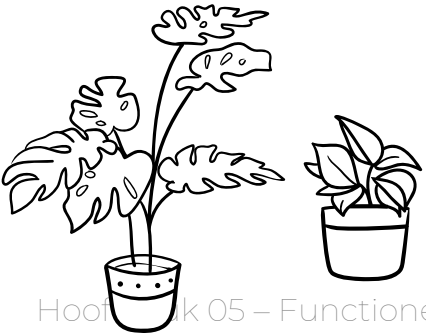
Rest & spread syntax

...

Rest en spread syntax

- Pas in index.html van 05thFP_en_Collections 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(1, ...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 with 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 syntax

- via de **rest** 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 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"]

Rest en Spread properties

- bij **object destructuring** kan je de 'overige properties' van een object verzamelen via de **rest property**
 - de niet reeds expliciet opgepikte key/value pairs komen in een nieuw object te zitten

```
const { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };  
console.log(x); // 1  
console.log(y); // 2  
console.log(z); // { a: 3, b: 4 }
```

Rest en Spread properties

- bij **object initializers** kan je de properties van een gegeven object kopiëren naar het nieuwe object via de **spread property**

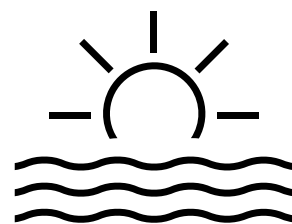
```
const x = 1;
const y = 2;
const z = { a: 3, b: 4 };
const newObj = { x, y, ...z };
console.log(newObj); // { x: 1, y: 2, a: 3, b: 4 }
```

- je kan op deze manier eenvoudig een shallow clone van een object nemen

```
const clone = { ...newObj };
console.log(clone); // { x: 1, y: 2, a: 3, b: 4 }
```

- je kan op deze manier ook objecten samenvoegen

```
const otherObj = { s: 10, t: 20 };
const mergedObj = { ...newObj, ...otherObj };
console.log(mergedObj); // {x: 1, y: 2, a: 3, b: 4, s: 10, t: 20}
```

**HO
GENT**