



# Webapplicaties II

Hoofdstuk 04 – Functional Programming  
met Arrays



# **04 Functional Programming met Arrays**

# Inhoud

- Functioneel programmeren
- Arrays
  - Herhaling
  - map / filter / reduce
- 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', 'voetbal' ]
```

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

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 complexheid 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

# 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

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

Definitie van de callback functie

Oproepen van de originele functie, die een 2<sup>de</sup> (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
```

# Arrays – geavanceerde methodes

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

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

# 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
```

# Arrays – geavanceerde methodes

```
// Stel dat we een array van objecten hebben.  
// Hoe kunnen we een object terugvinden  
// dat aan een specifieke voorwaarde voldoet  
const users = [  
  { id: 1, firstname: 'Jan', lastname: 'Janssens' },  
  { id: 2, firstname: 'Eva', lastname: 'De Smet' },  
  { id: 3, firstname: 'Pieter', lastname: 'Martens' }  
];  
  
const user = users.find(item => item.id === 1);  
console.log(user); // {id: 1, firstname: "Jan", lastname: "Janssens"}  
  
// De functie findIndex werkt analoog maar retourneert de index  
const indexuser = users.findIndex(item => item.id === 1);  
console.log(indexuser); // 0
```

# Arrays – geavanceerde methodes

```
// De functie sort sorteert de items van de array als strings by default
console.log(fruit.sort());
// ["blueberry", "melon", "orange", "pineapple", "strawberry"]

// Stel dat je je eigen sorteermethode wil meegeven, dan kan dat als volgt
// Je moet zelf een functie compare voorzien
// Algemene syntax van de functie compare
// function compare(a, b) {
//   if (a > b) return 1; --> een positief getal betekent dat a groter is dan b
//   if (a == b) return 0;
//   if (a < b) return -1; --> een negatief getal betekent dat a kleiner is dan b
// }

// Stel dat we de strings willen sorteren op aantal letters
function sorterenOpAantalLetters(a, b) {
    if (a.length > b.length) return 1;
    if (a.length === b.length) return 0;
    if (a.length < b.length) return -1;
}
console.log(fruit.sort(sorterenOpAantalLetters));
// ["melon", "orange", "blueberry", "pineapple", "strawberry"]
```

# Arrays – geavanceerde methodes

```
// Je kan het voorgaande ook verkort schrijven als
fruit.sort(function(a, b) {
    if (a.length > b.length) return 1;
    if (a.length === b.length) return 0;
    if (a.length < b.length) return -1;
});

console.log(fruit);
// ["melon", "orange", "blueberry", "pineapple", "strawberry"]

// Het voorgaande kan je nog korter schrijven als volgt
// a.length > b.length => a.length - b.length > 0
// --> een positief getal betekent dat a groter is dan b
// a.length === b.length => a.length - b.length is 0
// a.length < b.length => a.length - b.length < 0
// --> een negatief getal betekent dat a kleiner is dan b
fruit.sort((a, b) => a.length - b.length);
console.log(fruit);
// ["melon", "orange", "blueberry", "pineapple", "strawberry"]
```

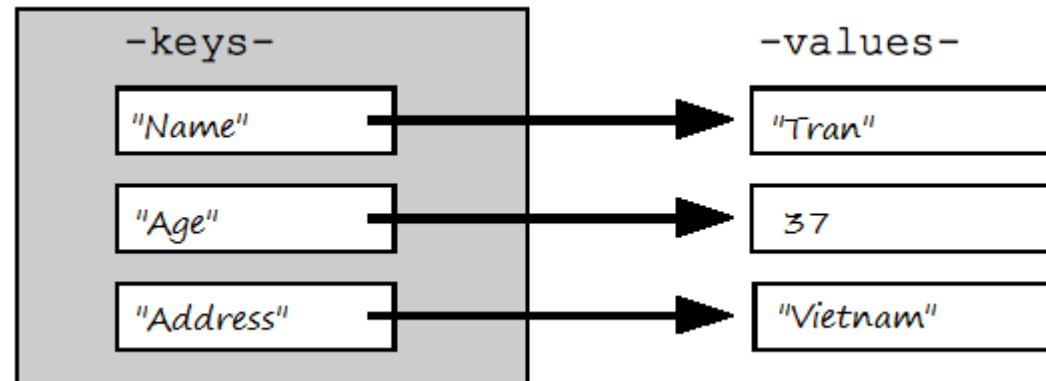
# **04 Functional Programming met Arrays**

Maps



# Maps

- Een map is vergelijkbaar met een object literal. Het belangrijke verschil is dat de keys van om het even welk type kunnen zijn



# Maps

- De belangrijkste methodes zijn
  - `new Map()` – creëert de map
  - `map.set(key, value)` – slaat de tuple (key, value) op in de map
  - `map.get(key)` – retourneert de value die hoort bij de key. Als de key niet bestaat wordt `undefined` geretourneerd.
  - `map.has(key)` – retourneert `true` als de key bestaat, anders `false`
  - `map.delete(key)` – verwijdert de value met de overeenkomstige key uit de map
  - `map.clear()` – maakt de map leeg
  - `map.size` – retourneert het aantal elementen in de map

# Maps

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

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

# Maps

```
let results1 = new Map();
results1.set('Club Brugge', 64);
results1.set('Anderlecht', 52);
results1.set('Charleroi', 51);

console.log(results1.get('Club Brugge')); // 64
console.log(results1.size); // 3
console.log(results1.has('Charleroi')); // true
results1.clear();
console.log(results1); // Map(0) {}

// Er kan ook gebruik gemaakt worden van chaining
let results2 = new Map();
results2
.set('Club Brugge', 64)
.set('Anderlecht', 52)
.set('Charleroi', 51);

console.log(results2.get('Club Brugge')); // 64
console.log(results2.size); // 3
```

# Maps

- We kunnen ook een Map creëren op basis van een array die wordt doorgegeven. Dit is volledig analoog met het voorgaande

```
let results3 = new Map([  
  ['Club Brugge', 64],  
  ['Anderlecht', 52],  
  ['Charleroi', 51]  
]);
```

```
console.log(results3.get('Anderlecht')); // 52  
console.log(results3.size); // 3
```

# Maps

- Het belangrijke verschil tussen een Map en een object literal is dat maps ook objecten kunnen hebben als keys

```
let results4 = new Map();
let p1 = { team: 'Club Brugge', trainer: 'Leko' };
let p2 = { team: 'Anderlecht', trainer: 'Vanhaezebroeck' };
let p3 = { team: 'Charleroi', trainer: 'Mazzu' };

results4.set(p1, 64);
results4.set(p2, 52);
results4.set(p3, 51);

console.log(results4.get(p1)); // 64
console.log(results4); // 3
let p4 = { team: 'Club Brugge', trainer: 'Leko' };
console.log(results4.get(p4)); // undefined
```

# Maps

- Om de keys van een map te vergelijken wordt er gebruik gemaakt van het algoritme SameValueZero
- Het werkt op ongeveer dezelfde manier als ===
  - Het verschil is dat NaN wordt beschouwd als gelijk aan NaN
  - We kunnen dus NaN ook gebruiken als key

# Maps

- Nogmaals met de array manier

```
let results5 = new Map([[p1, 64], [p2, 52], [p3, 51]]);
```

```
console.log(results5.get(p2)); // 52
```

```
console.log(results5.size); // 3
```



# Maps

- Om te itereren over een map kunnen we itereren over de keys

```
for (let player of results5.keys()) {  
    console.log(player);  
}  
// {team: "Club Brugge", trainer: "Leko"}  
// {team: "Anderlecht", trainer: "Vanhaezebroeck"}  
// {team: "Charleroi", trainer: "Mazzu"}
```

- Of over de values

```
for (let v of results5.values()) {  
    console.log(v);  
}  
// 64  
// 52  
// 51
```

# Maps

- We kunnen ook itereren over de [key, value] paren

```
// Itereren over [key, value] entries
for (let [k, v] of results5) {
  console.log(`club = ${k.team}, trainer = ${k.trainer}, punten = ${v}`);
}
// club = Club Brugge, trainer = Leko, punten = 64
// club = Anderlecht, trainer = Vanhaezebroeck, punten = 52
// club = Charleroi, trainer = Mazzu, punten = 51
```

- Voorgaande is een verkorte schrijfwijze van het volgende

```
for (let [k, v] of results5.entries()) {
  console.log(`club = ${k.team}, trainer = ${k.trainer}, punten = ${v}`);
}
// club = Club Brugge, trainer = Leko, punten = 64
// club = Anderlecht, trainer = Vanhaezebroeck, punten = 52
// club = Charleroi, trainer = Mazzu, punten = 51
```

# Maps

- We kunnen ook gebruik maken van de forEach

```
// Je kan ook gebruik maken van forEach
results5.forEach((value, key, map) => {
  console.log(`club = ${key.team}, trainer = ${key.trainer}, punten = ${value}`);
});
// club = Club Brugge, trainer = Leko, punten = 64
// club = Anderlecht, trainer = Vanhaezebroeck, punten = 52
// club = Charleroi, trainer = Mazzu, punten = 51
```

# **04 Functional Programming met Arrays**

Sets

# Sets

- Een set is een collectie van waarden, waarbij elke waarde maar 1 keer mag voorkomen
- Belangrijkste methodes
  - `new Set(iterable)` – creëert een set, eventueel vertrekkend van een array van waarden
  - `set.add(value)` – voegt de value toe aan de set. De set wordt geretourneerd
  - `set.delete(value)` – verwijdert de value. Retourneert true als de value bestond, anders false
  - `set.has(value)` – retourneert true als de value bestaat in de set, anders false
  - `set.clear()` – maakt de set leeg
  - `set.size` – retourneert het aantal elementen in de set

# Sets

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

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

# Sets

- Als we waarden toevoegen aan een set, zullen er enkel unieke waarden worden bijgehouden.

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

```
let john = { firstname: "John", lastname: "Williams" };
```

```
let pete = { firstname: "Pete", lastname: "Johnsons" };
```

```
let mary = { firstname: "Mary", lastname: "Stevens" };
```

```
set.add(john);
```

```
set.add(pete);
```

```
set.add(mary);
```

```
set.add(john);
```

```
set.add(mary);
```

```
// De set houdt enkel de unieke waarden bij  
console.log(set.size); // 3
```

# Sets

- Itereren over een set kan met de for-of (meest gebruikt)

```
for (let user of set) {  
    console.log(`${user.firstname} ${user.lastname}`);  
}  
// John Williams  
// Pete Johnsons  
// Mary Stevens
```

- Of itereren over de keys met de for-of

```
for (let user of set.keys()) {  
    console.log(`${user.firstname} ${user.lastname}`);  
}  
// John Williams  
// Pete Johnsons  
// Mary Stevens
```



# Sets

- We kunnen ook itereren over de values. Dit is hetzelfde als over de keys, dient voor compatibiliteit met Map

```
for (let user of set.values()) {  
    console.log(`${user.firstname} ${user.lastname}`);  
}  
// John Williams  
// Pete Johnsons  
// Mary Stevens
```

- Of itereren over de [key,value] paren

```
for (let [k, v] of set.entries()) {  
    console.log(`key = ${k.firstname} ${k.lastname}, value = ${v.firstname}  
${v.lastname}`);  
}  
// key = John Williams, value = John Williams  
// key = Pete Johnsons, value = Pete Johnsons  
// key = Mary Stevens, value = Mary Stevens
```

# Rest en spread operator bij Map en Set

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

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

# Rest en spread operator

- De rest operator verzamelt de resterende elementen van een iterable in een array

// Voorbeeld

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

// Nog een voorbeeld van de rest operator

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

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

# Rest en spread operator

- De spread operator doet net het omgekeerde van de rest operator en vormt de items van een iterable om tot individuele elementen

```
console.log(Math.max(-1, 5, 11, 3)); // 11
// Het volgende werkt niet omdat de functie max niet werkt op arrays
console.log(Math.max([-1, 5, 11, 3])); // NaN
// De spread operator wordt gebruikt om dit probleem op te lossen
console.log(Math.max(...[-1, 5, 11, 3])); // 11
// In tegenstelling tot de rest operator, kan je de spread operator
// op een willekeurige plaats in de sequentie gebruiken
console.log(Math.max(-1, ...[-1, 5, 11], 3)); // 11
```

# Rest en spread operator

```
// Nog een voorbeeld
const arr1 = ['Jan', 'Piet'];
const arr2 = ['Joris', 'Korneel'];
const arr12 = [...arr1, ...arr2];
console.log(arr12); // ["Jan", "Piet", "Joris", "Korneel"]
// Een andere mogelijkheid
arr1.push(...arr2);
console.log(arr1); // ["Jan", "Piet", "Joris", "Korneel"]

// Nog een voorbeeld
const str = "Hello";
console.log([...str]); // ["H", "e", "l", "l", "o"]
```

# Rest en spread operator

- Met behulp van de spread operator kan je een iterable omvormen naar een array als volgt. Een set wordt omgevormd naar een array

```
const set = new Set([11, -1, 6]);  
const arr = [...set];  
console.log(arr); // [11, -1, 6]
```

# Rest en spread operator

- Op Arrays kan je de bewerkingen `map()` en `filter()` uitvoeren. Dit bestaat niet voor Maps. De oplossing gaat als volgt door de iterable in een tussenstap om te vormen naar een array
  - stap 1: Converteer de Map naar een Array van `[key, value]` paren
  - stap 2: Maak gebruik van `map` of `filter` op de Array
  - stap 3: Converteer het resultaat terug naar een Map

# Rest en spread operator

- Voorbeeld: iedere ploeg krijgt 3 extra punten

```
const originalMap = new Map();
originalMap.set('Club Brugge', 64);
originalMap.set('Anderlecht', 52);
originalMap.set('Charleroi', 51);
console.log(originalMap);
// Map(3) {"Club Brugge" => 64, "Anderlecht" => 52, "Charleroi" => 51}

const mappedMap = new Map([...originalMap].map(([k, v]) => [k, v + 3]));
console.log(mappedMap);
// Map(3) {"Club Brugge" => 67, "Anderlecht" => 55, "Charleroi" => 54}
// stap 1 --> [...originalMap]
// stap 2 --> [...originalMap].map(([k, v]) => [k, v + 3])
// stap 3 --> new Map([...originalMap].map(([k, v]) => [k, v + 3]))
```



# Rest en spread operator

- Nog een voorbeeld, maar nu voor filter()

```
// Geef de ploegen met meer dan 60 punten
const filteredMap = new Map([...originalMap].filter(([k, v]) => v > 60));
console.log(filteredMap); // Map(1) {"Club Brugge" => 64}
// stap 1 --> [...originalMap]
// stap 2 --> [...originalMap].filter(([k, v]) => v > 60)
// stap 3 --> new Map([...originalMap].filter(([k, v]) => v > 60))
```

```
// Je kan de spread operator ook gebruiken om bijvoorbeeld
```

```
// maps te combineren
```

```
const anotherMap = new Map();
anotherMap.set('AA Gent', 47);
anotherMap.set('RC Genk', 44);
anotherMap.set('Standard', 41);
```

```
const combinedMap = new Map([...originalMap, ...anotherMap]);
console.log(combinedMap);
// Map(6) {"Club Brugge" => 64, "Anderlecht" => 52, "Charleroi" => 51,
// "AA Gent" => 47, "RC Genk" => 44, ...}
```