

User Interfaces 1


JavaScript

Functies

Functions

- Definitie
- `var` **VS** `let/const`
- Parameters (Call by value/Call by reference)
- Optionele parameters
- `arguments` object
- Recursive functies
- Functie als parameter
- Callback functies
- Self invoking functies
- Standaard functies

Wat is een Functie?

- Een functie (function) is één van de bouwstenen van JavaScript
- Het is een verzameling van "statements" met al dan niet parameters en/of een returnwaarde.
- Opm: in JavaScript is elke functie ook een object! 

Waarom functies?

- Hergebruik van code
- Standaard functies
- Beter leesbare code, overzichtelijker
- Beter te onderhouden code
- Betere *encapsulation*, locale variabelen mogelijk
- ...

Een eenvoudig voorbeeld

De definitie:

```
function sayHello() {  
    console.log("hello");  
}
```

Het oproepen:

```
sayHello();
```

als een functie expressie

```
let sayHello = function () {  
    console.log("hello");  
}
```

Een functie expressie genereert een anonieme functie die bijvoorbeeld kan bijgehouden worden in een variabele om dan via die variabele aan te roepen.

Vergeet de ronde haakjes niet!!! Test uit...

~~sayHello;~~

In dit voorbeeld is er géén **returnwaarde**, een **returntype** wordt **nooit** geschreven.

Conventie: De naam begint met een kleine letter en vervolgens "camelCase"

Arrow functie (arrow-notatie / lambda-expressie)

- Een Arrow functie is vereenvoudigde schrijfwijze van een functie expressie

1. Functie expressie

```
let sayHello = function () {  
  console.log("hello");  
}
```

2. Arrow functie

```
let sayHello = () => {  
  console.log("hello");  
}
```

- als de body maar uit 1 statement bestaat, geen {} nodig:

```
let sayHello = () => console.log("hello");
```

=> het resultaat van dit statement is tevens de returnwaarde van de arrow functie!

- indien er maar 1 parameter is, geen () nodig:

```
let print = param => console.log(param);
```

var, let en const

```
var x = 1;
function func() {
  var x = 2;
  console.log(x);
}
func();
console.log(x);
```

Output:

2

1

```
let x = 1;
function func() {
  let x = 2;
  console.log(x);
}
func();
console.log(x);
```

Output:

2

1

```
const x = 1;
function func() {
  const x = 2;
  console.log(x);
}
func();
console.log(x);
```

Output:

2

1

Functie met parameters

De definitie:

```
function square(number) {  
    return number * number;  
}
```

arrow notatie

```
let square = (number) => {  
    return number * number;  
}
```

indien 1 parameter, géén () nodig:

```
let square = number => {  
    return number * number;  
}
```

indien 1 statement, geen {} nodig:

```
let square = number => number * number;
```

Het oproepen:

```
let kwadraat = square(4);
```

In dit voorbeeld is er wél een **returnwaarde**, ze wordt toegekend aan de variabele *kwadraat*.

Een object als parameter

De definitie:

```
function double(tabel) {  
    for (let i = 0; i < tabel.length; i++) {  
        tabel[i] = 2 * tabel[i];  
    }  
}
```

Het oproepen:

```
let getallen = [1, 2, 3];  
double(getallen);  
console.log(getallen);
```

De inhoud van getallen is nu: [2, 4, 6]

Overdracht Parameters

- Primitieve datatypes
 - **Call by value**, er wordt een kopie meegegeven
 - De waarde van de variabele zelf wijzigt niet
- Objecten (en dus ook arrays)
 - **Call by reference**, het adres (een verwijzing naar de parameter) wordt meegegeven
 - De inhoud van de array of het object kan wijzigen

Argument type

Parameters hebben geen type en er is geen automatische controle op het type van de actuele parameters!

Voordeel: Je kan flexibele functies schrijven.

Nadeel: Zonder controle kan je vreemde resultaten bekomen.

Een theoretisch voorbeeld

```
function increment(getallen) {  
  if (typeof(getallen) === "number") {  
    return ++getallen;  
  }  
  for (let i = 0; i < getallen.length; i++) {  
    getallen[i]++;  
  }  
  return getallen;  
}
```

"boolean"
"number"
"string"
"object"

noodzakelijk? waarom (niet)?

```
for (let getal of getallen) {  
  getal++;  
}
```

NIET GEBRUIKEN BIJ BEWERKINGEN
→ GEBRUIKT KOPIE = 'BY VALUE'

Kan je oproepen met

```
let x = 10;  
x = increment(x);
```

x → 11

```
let tabel = [1,2,3];  
increment(tabel);
```

tabel → [2,3,4]

tabel → [1,2,3]

Return type

- Ook het return type is niet vastgelegd
- Een functie kan verschillende datatypes teruggeven → praktisch nut?
- Opmerking: Een functie kan ook een functie als returnwaarde teruggeven (en kan zichzelf herschrijven).

Optionele parameters

- Je mag een functie oproepen met **minder** actuele parameters dan er formele parameters zijn.
 - actuele parameters = tijdens de aanroep van de functie
 - formele parameters = in de functiedefinitie
- De ontbrekende parameters krijgen dan de waarde **undefined**.
- Je moet er dan wel voor zorgen dat ze in de functie een "default" waarde krijgen

Optionele parameters

```
function whereAreYou(location) {  
  const message = "Ik ben ";  
  if (!location) {  
    return message + "thuis!";  
  }  
  return message + location;  
}
```

Nieuw in ES6: gebruik van default parameter

```
function whereAreYou(location = "thuis") {  
  const message = "Ik ben ";  
  return message + location + "!";  
}
```

```
let text1, text2;  
text1 = whereAreYou("op school!");  
text2 = whereAreYou();
```

parameter ontbreekt...

Ik ben op school!

Ik ben thuis!

Het arguments object

De parameters van een functie worden bijgehouden in het object **arguments**.

Het bevat een array met de argumenten.

Je kan het aanspreken zoals een array:

arguments[0] geeft de eerste parameter terug

arguments.length geeft het aantal parameters terug

Via deze weg kan je een willekeurig aantal parameters bij een functie gebruiken (zie voorbeeld op de volgende slide).

Het object **arguments** bestaat niet in deze vorm voor de parameters van functies geschreven met de arrow notatie.

Het arguments object

```
function myConcat(separator, arg1) {  
    let result = arg1;  
    for (let i = 2; i < arguments.length; i++) {  
        result += separator + arguments[i];  
    }  
    return result;  
}
```

Voorbeelden van gebruik:

```
console.log(myConcat(", ", "rood", "groen", "blauw"));  
console.log(myConcat("-", 1, 2, 3, 4, 5));
```

Resultaten:

```
"rood, groen, blauw"  
"1-2-3-4-5"
```

Rekursieve functies

Bij functies kan je ook recursie toepassen (zichzelf terug aanroepen),
zorg ervoor dat je niet in een oneindige lus belandt...

```
function factorial(n) {  
    if (n <= 1) {  
        return 1;  
    }  
    return n * factorial(n - 1);  
}
```

In dit voorbeeld wordt $n!$ berekend (lees: n faculteit)
(vb. 3 faculteit berekenen: $3! = 3 \cdot 2 \cdot 1$). De functie roept zichzelf telkens
terug op met 1 minder dan de beginwaarde tot de waarde 1 bereikt wordt.

```
factorial(3) → 3 * factorial(2)  
factorial(2) → 2 * factorial(1)  
factorial(1) → 1  
return 3 * 2 * 1 → 6
```

Functies als parameters

Je kan ook functies als parameter gebruiken bij andere functies = **callback functie**

```
let data = [100, 1, 11, 10];  
function compare(a, b) {  
    return a - b;  
}  
data.sort(compare);
```

Kan data hier in plaats van met **let** met **const** geschreven worden?

Ja !!!
dus array.sort() gebruikt een intern sortingsalgoritme door elementen te 'swappen'

ofwel

```
let data = [100, 1, 11, 10];  
data.sort(function (a, b) {  
    return a - b;  
});
```

Anonieme functie

```
let data = [100, 1, 11, 10];  
data.sort((a, b) => a - b);
```

arrays multidimensional sort:

<https://stackoverflow.com/questions/6101475/how-does-one-sort-a-multi-dimensional-array-by-multiple-columns-in-javascript>

Callback functie

```
function invokeAndAdd(a, b) {  
    return a() + b();  
}  
  
function one() { return 1; }  
function two() { return 2; }  
  
let sum = invokeAndAdd(one, two);  
console.log("/" + sum + "/");
```

arrow notatie

```
let invokeAndAdd = (a, b) => a() + b();  
let one = () => 1;  
let two = () => 2;  
  
let sum = invokeAndAdd(one, two);  
console.log("/" + sum + "/");
```

- Bij het doorgeven van een functie als parameter van een andere functie wordt de parameterfunctie dikwijls een **callback** functie genoemd
- In dit theoretisch voorbeeld geven we de functies `one` en `two` als parameters mee aan de functie `invokeAndAdd`.
Na de uitvoering van deze functie heeft `sum` de waarde `/3/`.
- Test ook eens uit wat er gebeurt als je **de ronde haakjes** 'vergeet'...

Callback functie

```
function multiplyByTwo(a, b, c, callback) {  
  let array = [];  
  for (let i = 0; i < 3; i++) {  
    array[i] = callback(arguments[i] * 2);  
  }  
  return array;  
}  
function addOne(a) {  
  return a + 1;  
}  
let myArray = multiplyByTwo(10, 20, 30, addOne);  
console.log(myArray.toString());
```

Dit geeft de afdruk [21, 41, 61]

subtiel verschil in output:
21,41,61 (zonder haken)

2 functies → 1 for-lus voor beide!

```
[ 21, 41, 61 ]  
21,41,61
```

Anonieme callback functie

```
function multiplyByTwo(a, b, c, callback) {  
  let array = [];  
  for (let i = 0; i < 3; i++) {  
    array[i] = callback(arguments[i] * 2);  
  }  
  return array;  
}  
  
let myArray = multiplyByTwo(10, 20, 30, (a) => a + 1);  
console.log(myArray);
```

Dit geeft ook de afdruk `[21, 41, 61]`

De functie `addOne` is nu vervangen door een *anonymous callback function*

Callback Functie

Voordelen:

- minder globale variabelen
- betere performantie
- minder code te schrijven
- *kan asynchroon (we kunnen ondertussen andere dingen in de browser doen en hoeven bv. niet te wachten op het 'antwoord' van een API call)*

Nadelen:

- complexer
- moeilijker te lezen en te debuggen

Self-invoking functies

Tussen de eerste ronde haken staat een anonieme functie.
De tweede set ronde haken zegt "voer de functie uit" en kan parameters bevatten.

```
(  
    function () {  
        console.log("Hello World!");  
    }  
)();
```

```
(  
    name => console.log("Hello " + name)  
)("Jos");
```

```
(  
    function (name) {  
        console.log("Hello " + name);  
    }  
)("Dude");
```

Voordelen: geen event(s) nodig om te starten
 geen globale variabelen

Nadeel: je kunt de functie slechts één keer uitvoeren

Standaard functies

- `isFinite` → `false` bij `NaN` en `Infinity`
- `isNaN` → getal of niet
- `parseInt` → geeft `NaN` of geheel getal
- `parseFloat` → geeft `NaN` of decimaal getal
- `Number` → converteer naar getal
- `String` → converteer naar string
- ...

security issue:

zet string om naar waarde, als de (zelfs gedeeltelijke) inhoud van de string via de user wordt bekomen, kan hiermee eender welke functie geïnjecteerd worden...

performance issue:

bovendien genereert elke eval een nieuwe instantie van de js-interpreter...

- **`eval`** → `'evaluate'`: tekst als JS-code interpreteren en uitvoeren!

Standaard functies

Enkele voorbeelden

```
let waarde = "10";  
console.log(isNaN(waarde));
```

false

```
let getal = parseInt(waarde);  
console.log(isNaN(getal));
```

false

```
waarde = "10$";  
console.log(isNaN(waarde));
```

true

← is dus geen getal

```
getal = parseInt("010", 10);  
console.log(isNaN(getal));
```

false

← Anders geïnterpreteerd als 8
(octaal) door 0 vooraan

Oefening functies 1



- Schrijf een functie **keerOm(woord)** die de letters van het woord in omgekeerde volgorde teruggeeft.
- Stel om te testen **woord** gelijk aan "hamerhaai"
- Toon het resultaat via **console.log**

Invoer:

hamerhaai

Uitvoer:

iaahremah

Oefening functies 2



- Schrijf een functie `initialen(naam)` die de initialen van een ingevoerde naam teruggeeft (splits op een spatie). Als de naam geen spaties bevat geef je enkel die naam terug.
- Toon het resultaat via `console.log`
- Test met onderstaande voorbeelden:

Invoer: Jos Van den Bulcke
Uitvoer: J.V.d.B.

Invoer: Jos
Uitvoer: Jos

uitbreiding: eventuele spaties wegwerken die door de gebruiker vóór en na 'Jos' werden ingegeven

Oefening functies 3



- Schrijf een functie `eersteLetters(lijst)` die een array met de eerste letters van een woordenlijst teruggeeft.

Gebruik

```
const lijst = ["aap", "noot", "mies"];
```

als woordenlijst.

- Toon het resultaat via `console.log`
- Verwachte uitvoer (bij rechtstreeks afdrukken van de arrayvariabele):

```
["a", "n", "m"]
```