Functies

Functies *(Eng. functions)* zijn één van de fundamentele bouwstenen in JavaScript. Een functie bevat een verzameling van acties of statements om een welbepaalde taak uit te voeren. Acties die een functie uitvoert zijn afhankelijk van elkaar. Indien deze onafhankelijk zijn van elkaar kan je best twee functies beschrijven.

Om een functie te gebruiken moet deze eerst gedefinieerd worden in een bepaalde scope. Binnen deze scope kunnen we deze functie **aanroepen** *(Eng. call).* Een functie is zelf een object (`Function` object) en bevat zelf eigenschappen en methoden.

JavaScript bevat redelijk wat voorgedefinieerde functies, zoals:

- `decodeURIComponent`;
- `decodeURI`;
- `encodeURI()`;
- `encodeURIComponent()`;
- `eval()`;
- `isFinite()`;
- `isNaN()`;
- `parseFloat()`;
- `parseInt()`;
- `uneval()`;
- ...

Definiëren van functies

Declaratie

Een functie bestaat uit:

- het **trefwoord** *(Eng. keyword)* `function`
- de **naam** *(Eng. name)* voor de functie.

- een reeks van **parameters** omsloten door **ronde haakjes** *(Eng. parenthesis, pl. parentheses,* `(` *en* `)` *)* en elke parameter wordt gescheiden door een **komma** *(Eng. comma,* `,` *)*
- een **blokstatement** *(Eng. block statement)* `{…}` met daarbinnen de statements.

Het volgende voorbeeld definieert een eenvoudige functie met de naam `addition` :

./js_essentials/functions/addition.js

```js
function addition(x, y) {
  return x + y;
}

console.log("addition(6, -2):", addition(6, -2));
```

```
$ node addition.js
addition(6, -2): 4
```

De functie `addition` definieert 2 parameters, namelijk `x` en `y` . Het `return` -statement specificeert de waarde die de functie teruggeeft. In dit geval geven we de som van de waarden die de parameters `x` en `y` bevatten terug.

Parameter en Argument

- Een **parameter** is een variabele die als input van de functie gedefinieerd is.
- Een **argument** is een waarde die je meegeeft bij het aanroepen van de functie.

In bovenstaand voorbeeld zijn `x` en `y` de parameters voor de respectievelijke argumenten `1` en `2` .

./js_essentials/functions/swap_person.js

```js
const personA = "John Doe";

function swap(person) {
  person = "Jane Doe";
  return person;
}

const personB = swap(personA);
console.log("personA:", personA);
console.log("personB:", personB);
```

```
$ node swap_person.js
personA: John Doe
personB: Jane Doe
```

Pass by Value

Argumenten als primitief datatype worden doorgegeven aan de functie door de waarde ***(Eng. pass by value)*** ervan. Wanneer in dit geval de waarde wordt aangepast van dit argument binnen het blok statement, dan zal de waarde slechts lokaal en dus niet globaal aangepast worden.

./js_essentials/functions/double_salary.js

```js
function doubleSalary(s) {
  s *= 2;
}

let salary = 1999;
console.log(`Your salary is € ${salary}.`);

doubleSalary(salary);
console.log(`Your salary is € ${salary}.`);
```

```
$ node double_salary.js
Your salary is € 1999.
Your salary is € 1999.
```

Pass by Reference

Geven we een niet-primitieve waarde of object door als een argument en passen we daarna een eigenschap van dit object aan binnen de functie, dan is deze verandering ook zichtbaar buiten de functie, zoals in het volgende voorbeeld:

./js_essentials/functions/double_salary_for_person.js

```js
function doubleSalaryForPerson(p) {
  p.salary *= 2;
}
```

```
4
5   let person = {
6     firstName: "Phil",
7     salary: 1999,
8   };
9   console.log(`Your salary is € ${person.salary}.`);
10
11  doubleSalaryForPerson(person);
12  console.log(`Your salary is € ${person.salary}.`);
```

```
$ node double_salary_for_person.js
Your salary is € 1999.
Your salary is € 3998.
```

Expressies

Anoniem

Functies kunnen ook aangemaakt worden via een functie expressie. Deze expressies kunnen zowel anoniem (zonder een naam) of met een naam geïmplementeerd worden, bijv.:

./js_essentials/functions/sq.js — js

```
1   const sq = function(number) {
2     return number * number;
3   };
4
5   console.log("sq(3):", sq(3));
```

```
$ node sq.js
sq(3):  9
```

Naam

Wanneer we een naam voorzien in de functie expressie, dan kunnen we binnen het blok statement van deze expressie terug deze expressie aanspreken via de naam, bijvoorbeeld:

./js_essentials/functions/fac.js

```js
const fac = function factorial(n) {
  return n < 2 ? 1 : n * factorial(n - 1);
};

console.log("fac(3):", fac(3));
```

```
$ node fac.js
fac(3):  6
```

Aanroepen van een functie

Een functie wordt niet uitgevoerd door gewoonweg de functie te definiëren. Tijdens het definiëren geven we een naam aan de functie en specificeren wat er gedaan moet worden (acties) wanneer de functie wordt aangeroepen. Door de functie **aan te roepen** ***(Eng. call, invocation)*** worden deze acties uitgevoerd al dan niet met beïnvloeding door de doorgegeven parameters.

./js_essentials/functions/addition.js

```js
function addition(x, y) {
  return x + y;
}

console.log("addition(6, -2):", addition(6, -2));
```

```
$ node addition.js
addition(6, -2): 4
```

De scope van een functie is de plek waar het wordt gedeclareerd, dat kan binnen een andere functie zijn, binnen een object, globaal …

Goed

```js
console.log("addition(6, -2):", addition(6, -2));

```

```
function addition(x, y) {
  return x + y;
}
```

```
$ node addition_good.js
addition(6, -2): 4
```

Functie declaraties worden **gehesen** ***(Eng. hoisted).*** Dit betekent dat we deze kunnen aanroepen voordat ze gedeclareerd worden.

Fout

```js
console.log("add(6, -2):", add(6, -2));

const add = function addition(x, y) {
  return x + y;
};
```

```
$ node addition_bad.js
console.log("add(6, -2):", add(6, -2));
                           ^

ReferenceError: Cannot access 'add' before initialization
```

Functie expressies daarentegen worden niet gehesen. Dit resulteert in de foutboodschap: `ReferenceError: Cannot access 'add' before initialization`.

Recursie

Een **recursieve functie** is een functie die zichzelf aanroept. Het is mogelijk om elke recursief algoritme te converteren naar een niet recursief algoritme, maar de code wordt dan minder leesbaar.

Opgelet

Een recursieve functie moet een conditie bevatten om **oneindige recursie** *(Eng. infinite recursion)* te vermijden.

./js_essentials/functions/factorial.js

```js
function factorial(n) {
  n = Math.abs(Math.round(n));
  let result = 1;
  if (1 < n) {
    result = n * factorial(n - 1);
  }
  return result;
}

console.log("factorial(0):", factorial(0));
console.log("factorial(1):", factorial(1));
console.log("factorial(2):", factorial(2));
console.log("factorial(3):", factorial(3));
console.log("factorial(4):", factorial(4));
console.log("factorial(5):", factorial(5));
console.log("factorial(6):", factorial(6));
```

```
$ node factorial.js
factorial(0): 1
factorial(1): 1
factorial(2): 2
factorial(3): 6
factorial(4): 24
factorial(5): 120
factorial(6): 720
```

Naamgeving van een functie

Functies zijn acties, zodat de naam bij voorkeur begint met een **werkwoord** *(Eng. verb).* De naam beschrijft kort wat functie doet. Het geeft een indicatie aan andere programmeurs wat de functionaliteit is van deze functie. De naamgeving van functies wordt bepaald door een development team binnen een digital agency.

Functies kunnen bijvoorbeeld **starten met** *(prefix):*

- `get…`
  Haal iets op en geeft dit terug via het trefwoord `return`.
  Bijv. `getFullName()`, ***get the full name of the person and return it.***
- `set…`
  Stel een bepaalde eigenschap in.
  Bijv. `setFullName()`, ***set the full name of the person.***
- `calc…`
  Bereken iets.
  Bijv. `calcSum()`, ***calculate the sum of a couple of values and return the result.***
- `check…`
  Controleer iets.
  Bijv. `checkPermission()`, ***check the permission of a person for a certain resource, it returns true or false.***
- `create…`
  Aanmaak van iets.
- `fetch…`
  Haal data op uit een externe bron.
  Bijv. `fetchRandomUsers()`, ***fetch data from a certain online resources in order to get random users.***
- `show…`
  Toon iets.
  Bijv. `showMessage()`, ***show a certain message.***

Scope van een functie

Variabelen gedefinieerd in een functie zijn niet toegankelijk buiten de functie, omdat de variabele enkel in de scope van de functie is gedeclareerd. Een functie heeft wel toegang tot alle variabelen en functies in dezelfde scope van deze functie. Dit betekent dat wanneer een functie gedefinieerd is in de globale scope toegang heeft tot alle code die eveneens gedefinieerd is in dezelfde globale scope. Definiëren we een functie binnen een andere functie, dan heeft deze **geneste functie** *(Eng. nested function)* toegang tot alle variabelen uit de ***parent function*** en ook tot alle variabelen waarvan de ouder of parent ook toegang tot heeft.

./js_essentials/functions/nested_function.js

```js
const num1 = 20;
const num2 = 30;
const name = "Olivier";

// This function is defined in the global scope
function multiply() {
  return num1 * num2;
```

```
  }

  // A nested function example
  function getScore() {
    const num1 = 2;
    const num2 = 3;

    function add() {
      return `${name} scored ${num1 + num2} goals. He's a great soccer player!`;
    }

    return add();
  }

  console.log("multiply():", multiply());
  console.log("getScore():", getScore());
```

```
$ node nested_function.js
multiply(): 600
getScore(): Olivier scored 5 goals. He's a great soccer player!
```

Een geneste functie is een functie in een functie. De geneste functie is 'private' wat betekent dat deze niet kan aangesproken worden buiten de parent functie. Een geneste functie is een **closure** wat betekent dat de argumenten en de variabelen van de parent functie kan aangesproken worden.

./js_essentials/functions/add_squares.js — js

```
function addSquares(a, b) {
  function square(x) {
    return x * x;
  }
  return square(a) + square(b);
}

console.log("addSquares(2, 3):", addSquares(2, 3));
console.log("addSquares(3, 4):", addSquares(3, 4));
console.log("addSquares(4, 5):", addSquares(4, 5));
```

```
$ node add_squares.js
addSquares(2, 3): 13
addSquares(3, 4): 25
addSquares(4, 5): 41
```

Een geneste functie kan ook teruggegeven worden aan de aanvrager en dit via het `return` statement.

Arguments object

De argumenten van een functie worden gehandhaafd in een object gelijkaardig met een array object. We kunnen deze argumenten opvragen met `arguments[i]` waarbij `i` de index (startend met `0`) is van een specifiek argument. Het eerste argument kunnen we opvragen via `arguments[0]`. Het totaal aantal argumenten kan opgevraagd worden via `arguments.length`.

./js_essentials/functions/concat.js

```js
function concat(separator) {
  let str = "";
  for (let i = 1; i < arguments.length; i++) {
    str += arguments[i] + (i < arguments.length - 1 ? separator : "");
  }
  return str;
}

let result;
result = concat(" | ", "Computer Systems", "Programming 1", "Web Design");
console.log(result);
result = concat(", ", "@Work 1", "Programming 2", "UI Design");
console.log(result);
```

```
$ node concat.js
Computer Systems | Programming 1 | Web Design
@Work 1, Programming 2, UI Design
```

> **Opmerking**
>
> De waarden voor de parameters (hier de separator) behoren ook tot het arguments object, daarom begint de lus hier vanaf index `1` en niet `0`.

Functieparameters

Vanaf ES6 kunnen we gebruik maken van twee nieuwe parameters, namelijk **default parameters** en **rest parameters**. De default waarde van parameters is `undefined`. In sommige situaties kan het handig zijn om de default waarde aan te passen.

Veronderstel de volgende code:

./js_essentials/functions/addition.js (00) js

```
function addition(a, b) {
  return a + b;
}

console.log("addition(8):", addition(8));
```

```
$ node addition.js
addition(8): NaN
```

> **Tip**
>
> NaN staat voor **Not a Number**.

We kunnen dit opvangen door na te gaan of waarde van b al dan niet `undefined` is:

./js_essentials/functions/addition.js (01) js

```
function addition(a, b) {
  b = typeof b !== "undefined" ? b : 0;
  return a + b;
}

console.log("addition(8):", addition(8));
```

```
$ node addition.js
addition(8): 8
```

Default Parameters

De check of de variabele `b` al dan niet `undefined` is kunnen we vermijden door gebruik te maken van een **default parameter** *(Ned. standaardparameter)*.

./js_essentials/functions/addition.js (02) js

```js
function addition(a, b = 0) {
  return a + b;
}

console.log("addition(8):", addition(8));
```

```
$ node addition.js
addition(8): 8
```

Maken we van parameter `a` een default parameter en van `b` een gewone parameter dan resulteert de volgende code in `NaN`.

./js_essentials/functions/addition.js (03) js

```js
function addition(a = 0, b) {
  return a + b;
}

console.log("addition(8):", addition(8));
```

```
$ node addition.js
addition(8): NaN
```

Dit werkt ook met object deconstruction.

./js_essentials/functions/print_coordinates.js (03) js

```js
function printCoordinates({ x = 0, y = 0, z }) {
  console.log("x:", x, "y:", y, "z:", z);
}

const coordinatesA = {};
```

```
const coordinatesB = { x: 1 };
const coordinatesC = { y: 1 };
const coordinatesD = { z: 1 };

printCoordinates(coordinatesA);
printCoordinates(coordinatesB);
printCoordinates(coordinatesC);
printCoordinates(coordinatesD);
```

```
$ node print_coordinates.js
x: 0 y: 0 z: undefined
x: 1 y: 0 z: undefined
x: 0 y: 1 z: undefined
x: 0 y: 0 z: 1
```

Rest Parameters

Een rest parameter *(Ned. restparameter)* laat toe om een oneindig aantal argumenten voor te stellen met een array. Dit keer een echte array dit in tegenstelling tot het arguments object.

./js_essentials/functions/multiply.js — js

```
function multiply(factor, ...args) {
  let result = 0;
  for (let i = 0; i < args.length; i++) {
    result += args[i] * factor;
  }
  return result;
}

console.log("multiply(2, 1, 2, 3):", multiply(2, 1, 2, 3));
```

```
$ node multiply.js
multiply(2, 1, 2, 3): 12
```

Het eerste argument behoort niet tot de **rest parameter** `args`. De lus kan dus starten vanaf `0`.

Arrow functions

Een **pijlfunctie-expressie** *(Eng. arrow function expression)* heeft een kortere syntaxis in vergelijking met een functie expressie en bevat niet `this`, `arguments`, `super` en `new.target`. Een arrow function is altijd een **anonieme functie** *(Eng. anonymous function).*

./js_essentials/functions/gasses.js — js

```js
const gasses = ["helium", "neon", "argon", "krypton", "xenon", "radon"];

const a1 = gasses.map(function(s) {
  return s.length;
});

const a2 = gasses.map((s) => s.length);

console.log("a1:", a1);
console.log("a2:", a2);
```

```
$ node gasses.js
a1: [ 6, 4, 5, 7, 5, 5 ]
a2: [ 6, 4, 5, 7, 5, 5 ]
```

IIFE

Een ***Immediately-Invoked Function Expression*** (IIFE – spreek uit: 'eyeffee') is een JavaScript-functie die onmiddellijk na het definiëren uitgevoerd wordt. Via een IIFE kunnen we de scope van de function beschermen alsook de variabelen binnen deze functie. Een IIFE is een **zelfuitvoerende anonieme functie** *(Eng. self-executing anonymous function).*

./js_essentials/functions/iife.js (00) — js

```js
(function() {
  // statements
})();
```

De IIFE bestaat uit een groepoperator `(…)` met daarbinnen een anonieme functie expressie. Om een IIFE uit te voeren moeten we deze aanroepen, dit kan door direct na de groepoperator een call te voorzien via `()`.

Om eem IIFE ten volle te snappen zullen we een gewone functiedeclaratie omzetten in een IIFE.

./js_essentials/functions/iife.js (01) js

```js
function add() {
  const x = 5;
  const y = -4;
  return x + y;
}

console.log("add():", add());
```

```
$ node iife.js
add(): 1
```

De variabelen `x` en `y` kunnen niet gewijzigd worden buiten de functie wat betekent dat ze **immutable** ***(Ned. onveranderlijk)*** zijn. De functie `add()` kan wel aangesproken worden. Om dit te vermijden kunnen de functiedeclaratie omsluiten door een groepoperator.

./js_essentials/functions/iife.js (02) js

```js
(function add() {
  const x = 5;
  const y = -4;
  return x + y;
});

console.log("add():", add());
```

```
$ node iife.js
console.log("add():", add());
                      ^

ReferenceError: add is not defined
```

De functie `add()` kan niet meer aangesproken worden buiten de groepoperator resulterende in de fout “ReferenceError: add is not defined”.

We verwijderen vervolgens de naam van de functie wat resulteert in een anonieme functie of functie expressie.

./js_essentials/functions/iife.js (03) js

```
(function() {
  const x = 5;
  const y = -4;
  return x + y;
});
```

Tenslotte moeten we deze functie aanroepen. Dat kan eenvoudig door na de groepoperator een bracket, paar `()` toe te voegen.

./js_essentials/functions/iife.js (04) js

```
(function() {
  const x = 5;
  const y = -4;
  return x + y;
})();
```

Met de arrow function-notatie krijgen we de volgende IIFE:

./js_essentials/functions/iife.js (05) js

```
(() => {
  const x = 5;
  const y = -4;
  return x + y;
})();
```

Aan een IFFE kan je ook argumenten meegeven en parameters definiëren.

./js_essentials/functions/iife.js (06) js

```
((x, y) => {
  return x + y;
})(5, -4);
```

./js_essentials/functions/iife.js (07) js

```
const result = ((x, y) => {
  return x + y;
})(5, -4);
```

```
4
5   console.log("result:", result);
```

```
$ node iife.js
result: 1
```

← Lussen

Objecten →

Laatst bijgewerkt: 8/9/2022 13:39:31