**Eloquent javascript**

<https://eloquentjavascript.net/01_values.html>

H1.

**Operators:** symbols like + and \*

The % symbol is used to represent the **remainder operation**. 314 % 100 produces 14, and 144 % 12 gives 0 (rest)

You’ll also often see this operator referred to as **modulo.**

There are three special values in JavaScript that are considered numbers but don’t behave like normal numbers.

* Infinity and -Infinity,
* NaN (don’t yield a meaningful result.)

**String**

whenever a backslash (\) is found inside quoted text, it indicates that the character after it has a special meaning. This is called *escaping* the character.

If two backslashes follow each other, they will collapse together, and only one will be left in the resulting string value.

Backtick-quoted strings, usually called *template literals*, can do a few more tricks. Apart from being able to span lines, they can also embed other values. `half of 100 is ${100 / 2}`

**binary operators** Operators that use two values

**unary operators** Operators that take one value

console.log(- (10 - 2)) // = -8

**>=** greater than or equal to

**<=**  less than or equal to

**==**  equal to

**!=**  not equal to

**NaN** the only value in JavaScript that is not equal to itself

== Implicet cohorsion

=== explicet cohorsion

0 false

1 true

!0 true

!!0 false

The **&&** operator represents logical *and*.

The **||** operator denotes logical *or*.

console.log(output || n);

**conditional operator** (or sometimes just the **ternaryoperator** since it is the only such operator in the language). The value on the left of the question mark “picks” which of the other two values will come out. When it is true, it chooses the middle value, and when it is false, it chooses the value on the right.

**console.log(true ? 1 : 2)**;

// → 1

**console.log(false ? 1 : 2);**

// → 2

**Null**/ **undefine**d that are used to denote the absence of a *meaningful* value.

***Type coercion***: When an operator is applied to the “wrong” type of value, JavaScript will quietly convert that value to the type it needs,

H2.

**Statements:**  a program is built out of

**Expressions:** Statements tend to contain

**Bindings** can be used to file pieces of data under a name, and they are useful for tracking state in your program.

Binding names must not start with a **digit** and can’t be a **reserved keyword.** Bindings mogen geen space bevatten.

Var caught = 5 \* 5;

Var = Keyword –

Caught = variabele name

= assignment operator

\* arithmetic operator

**environment** is the set of bindings that are defined.

**Functions** are special values that encapsulate a piece of program.

**Expression:** A fragment of code that produces a value

After a binding has been defined, its name can be used as an **expression**.

Anything that produces a value is an expression in JavaScript

**binary operator** : applied to two expressions

**unary operator**

**Side effects**: change the internal state of the machine in a way that will affect the statements that come after it. It prints a line.

Showing a dialog box or writing text to the screen is a **side effect.**

**Binding** To catch and hold values

**Let:** keyword

**Digits:** tekens zoals - . Een binding mag hier niet mee beginnen.

**conditional execution**: the program takes the proper branch based on the situation at hand

**Loop:** The word while is followed by an expression in parentheses and then a statement,

**While loop**

let result = 1;

let counter = 0;

while (counter < 10) {

result = result \* 2;

counter = counter + 1;

}

console.log(result);

// → 1024

**Do loop**

let yourName;

do {

yourName = prompt("Who are you?");

} while (!yourName);

console.log(yourName);

**! operator** will convert a value to Boolean type before negating it, and all strings except ""convert to true. This means the loop continues going round until you provide a non-empty name.

**Do loop** is a control structure similar to a while loop. It differs only on one point: **a do loop** always executes its body at least once, and it starts testing whether it should stop only after that first execution. To reflect this, the test appears after the body of the loop.

**For loop**

for (let number = 0; number <= 12; number = number + 2) {

console.log(number);

}

// → 0

// → 2

// … etcetera

let result = 1;

for (let counter = 0; counter < 10; counter = counter + 1) {

result = result \* 2;

}

console.log(result);

// → 1024

**Parentheses** ( )

**Semicolons** ;

**Break**  has the effect of immediately jumping out of the enclosing loop.

**Remainder (%) operator** is an easy way to test whether a number is divisible by another number. If it is, the remainder of their division is zero.

switch (prompt("What is the weather like?")) {

case "rainy":

console.log("Remember to bring an umbrella.");

break;

case "sunny":

console.log("Dress lightly.");

case "cloudy":

console.log("Go outside.");

break;

default:

console.log("Unknown weather type!");

break;

}

H3.

**Functie:** set van statements die een taak uitvoeren of een value berekenen.

Voordelen

* Geen herhaling nodig
* Clustert code > leesbaar
* Werkt zonder andere code te beinvloeden

Manieren van schrijven:

1. Function a(){}
2. Const square = function(x){}

manier 2 > wordt pas later gelezen door js geheugen. > kan pas gelezen worden als het aan de beurt is

**API call:** opvragen gegevens

**Progressive enhancement** Bruikbaar voor alle leveranciers

**Invoking/Calling/ Applying**: Het uitvoeren van een functie

**Parameter:** function ()

**Argument** function(…)

**Binding**

**Scope** is the part of the program in which the binding is visible. Each binding has a scope.

**Global bindings**: bindings defined outside of any function or block, the scope is the whole program.

**localbindings**: declared inside a function can be referenced only in that function

**var** is global zichtbaar - global scope. visible throughout the whole function that they appear in—or throughout the global scope, if they are not in a function.

**let** alleen in de local environment – block scope

let x = 10;

if (true) {

let y = 20;

var z = 30;

console.log(x + y + z);

// → 60

}

// y is not visible here

console.log(x + z);

// → 40

const halve = function(n) {

return n / 2;

};

let n = 10;

console.log(halve(100));

// → 50

console.log(n);

// → 10

**Nested scope/ lexical scoping** **:** Blocks and functions can be created inside other blocks and functions. Each local scope can also see all the local scopes that contain it, and all scopes can see the global scope.

const hummus = function(factor) {

const ingredient = function(amount, unit, name) {

let ingredientAmount = amount \* factor;

if (ingredientAmount > 1) {

unit += "s";

}

console.log(`${ingredientAmount} ${unit} ${name}`);

};

ingredient(1, "can", "chickpeas");

ingredient(0.25, "cup", "tahini");

ingredient(0.25, "cup", "lemon juice");

ingredient(1, "clove", "garlic");

ingredient(2, "tablespoon", "olive oil");

ingredient(0.5, "teaspoon", "cumin");

};

Als een functie in een value zit kan je deze veranderen in een nieuwe **binding**

let launchMissiles = function() {

missileSystem.launch("now");

};

if (safeMode) {

launchMissiles = function() {/\* do nothing \*/};

}

**function declaration** create a function binding with the function keyword at the start of a statement. Function declarations are not part of the regular top-to-bottom flow of control. They are conceptually moved to the top of their scope and can be used by all the code in that scope.

**Arrow functions** =>

Als er 1 expressie is in de parameters kunnen de parameters weggelaten worden. Als de body een single expression is kunnen braces ook worden weggelaten. Zie 2.

1. const square1 = (x) => { return x \* x; };
2. const square2 = x => x \* x;

Wanneer een functie geen parameters bevat, staan er gewoon lege parameters.

const horn = () => {

console.log("Toot");

};

const power = (base, exponent) => {

let result = 1;

for (let count = 0; count < exponent; count++) {

result \*= base;

}

return result;

};

**Call stack**: de plaats waar de computer de volgorde van context opslaat. Every time a function is called, the current context is stored on top of this stack.

Als er meer parameters worden aangeroepen dan er bestaan, worden ze genegeerd. Als er te weinig worden aangeroepen staat er: undefined.

function minus(a, b) {

if (b === undefined) return -a;

else return a - b;

}

console.log(minus(10));

// → -10

console.log(minus(10, 5));

// → 5

Als er een = operator achter een parameter staat gevolgd door een expression, wordt deze expressie gebruikt wanneer de parameter niet wordt gegeven.

function power(base, exponent = 2) {

let result = 1;

for (let count = 0; count < exponent; count++) {

result \*= base;

}

return result;

}

console.log(power(4));

// → 16

console.log(power(2, 6));

// → 64

**Closure**: de mogelijkheid om een local binding op te roepen uit een enclosing scope.

function wrapValue(n) {

let local = n;

return () => local;

}

let wrap1 = wrapValue(1);

let wrap2 = wrapValue(2);

console.log(wrap1());

// → 1

console.log(wrap2());

// → 2

function multiplier(factor) {

return number => number \* factor;

}

let twice = multiplier(2);

console.log(twice(5));

// → 10

**Recursive**: A function that calls itself

function power(base, exponent) {

if (exponent == 0) {

return 1;

} else {

return base \* power(base, exponent - 1);

}

}

console.log(power(2, 3));

// → 8

const makeNoise = function() {

console.log("Pling!");

};

makeNoise();

// → Pling!

//10x +2 bij Result

const power = function(base, exponent) {

let result = 1;

for (let count = 0; count < exponent; count++) {

result \*= base;

}

return result;

};

console.log(power(2, 10));

// → 1024

**purefunction** is a specific kind of value-producing function that not only has no side effects but also doesn’t rely on side effects from other code

Functions that create values

Functions that create a line : side effect

We want to write a program that prints two numbers: the numbers of cows and chickens on a farm, with the words Cows and Chickens after them and zeros padded before both numbers so that they are always three digits long.

**Side effect:**

function printZeroPaddedWithLabel(number, label) {

let numberString = String(number);

while (numberString.length < 3) {

numberString = "0" + numberString;

}

console.log(`${numberString} ${label}`);

}

function printFarmInventory(cows, chickens, pigs) {

printZeroPaddedWithLabel(cows, "Cows");

printZeroPaddedWithLabel(chickens, "Chickens");

printZeroPaddedWithLabel(pigs, "Pigs");

}

printFarmInventory(7, 11, 3);

**Pure function**

function zeroPad(number, width) {

let string = String(number);

while (string.length < width) {

string = "0" + string;

}

return string;

}

function printFarmInventory(cows, chickens, pigs) {

console.log(`${zeroPad(cows, 3)} Cows`);

console.log(`${zeroPad(chickens, 3)} Chickens`);

console.log(`${zeroPad(pigs, 3)} Pigs`);

}

printFarmInventory(7, 16, 3);

H4.

**Serialize the data:** save data or send it by converting

data structures are built from Numbers, Booleans, and strings

// Haal een waarde uit een array

let listOfNumbers = [2, 3, 5, 7, 11];

console.log(listOfNumbers[2]);

// → 5

console.log(listOfNumbers[0]);

// → 2

console.log(listOfNumbers[2 - 1]);

// → 3

**Objects** allow us to group value together

**Properties:** een variabele (in een object), bijvoorbeeld naam: van het hotel

**Acces to properties**: value**.**color (the word after the dot is the literal name of the property)

value[i] (the expression between the brackets is **evaluated*t***o get the property name)

Almost all JavaScript values have properties. The exceptions are null and undefined.

null.length;

// → TypeError: null has no properties

**Expressions:** een stukje code die een value produceert zoals:

Var color = ‘beige’;

Var area = 3\*2;

Math.max

**Methods** zijn functies die in properties leven. Bijvoorbeeld .toUpperCase.

They (usually) act on the value they are a property of.

//Voorbeeld method om een array te manipuleren:

**POP & PUSH**

let sequence = [1, 2, 3];

sequence.push(4);

sequence.push(5);

console.log(sequence);

// → [1, 2, 3, 4, 5]

console.log(sequence.pop());

// → 5

console.log(sequence);

// → [1, 2, 3, 4]

**A stack**, in programming, is a data structure that allows you to push values into it and pop them out again

**Object:** een verzameling properties omringd door braces.

let day1 = {

squirrel: false,

events: ["work", "touched tree", "pizza", "running"]

};

console.log(day1.squirrel);

// → false

console.log(day1.wolf);

// → undefined

day1.wolf = false;

console.log(day1.wolf);

// → false

Hier wordt een property toegekend aan day1

**braces have *two* meanings** in JavaScript.

1. they start a block of statements.
2. they describe an object.

**Delete operator**

let anObject = {left: 1, right: 2};

console.log(anObject.left);

// → 1

delete anObject.left;

console.log(anObject.left);

// → undefined

console.log("left" in anObject);

// → false

console.log("right" in anObject);

// → true

**Object.keys** function:  find out what properties an object has

console.log(Object.keys({x: 0, y: 0, z: 2}));

// → ["x", "y", "z"]

**Object.assign** function: copies all properties from one object into another

let objectA = {a: 1, b: 2};

Object.assign(objectA, {b: 3, c: 4});

console.log(objectA);

// → {a: 1, b: 3, c: 4}

**Array van objecten**

let journal = [

{events: ["work", "touched tree", "pizza",

"running", "television"],

squirrel: false},

{events: ["work", "ice cream", "cauliflower",

"lasagna", "touched tree", "brushed teeth"],

squirrel: false},

{events: ["weekend", "cycling", "break", "peanuts",

"beer"],

squirrel: true},

/\* and so on... \*/

];

**Var** kan door het hele document aangepast worden

**Let** kan niet door het hele document aangepast worden

**Const** kan niet door het hele document aangepast worden

Behalve als let en const een object zijn: const b={b:20} dan kan dit op 1 manier veranderd worden

const score = {visitors: 0, home: 0};

// This is okay

score.visitors = 1;

// This isn't allowed

score = {visitors: 1, home: 1};

Comparing different objects will return false, even if they have identical properties. They have not the same identity

**no “deep” comparison** operation built into JavaScript, which compares objects by contents

let object1 = {value: 10};

let object2 = object1;

let object3 = {value: 10};

console.log(object1 == object2);

// → true

console.log(object1 == object3);

// → false

let journal = [];

function addEntry(events, squirrel) {

journal.push({events, squirrel}); // if a property name in brace notation isn’t followed by a value, its value is taken from the binding with the same name.

}

addEntry(["work", "touched tree", "pizza", "running",

"television"], false);

addEntry(["work", "ice cream", "cauliflower", "lasagna",

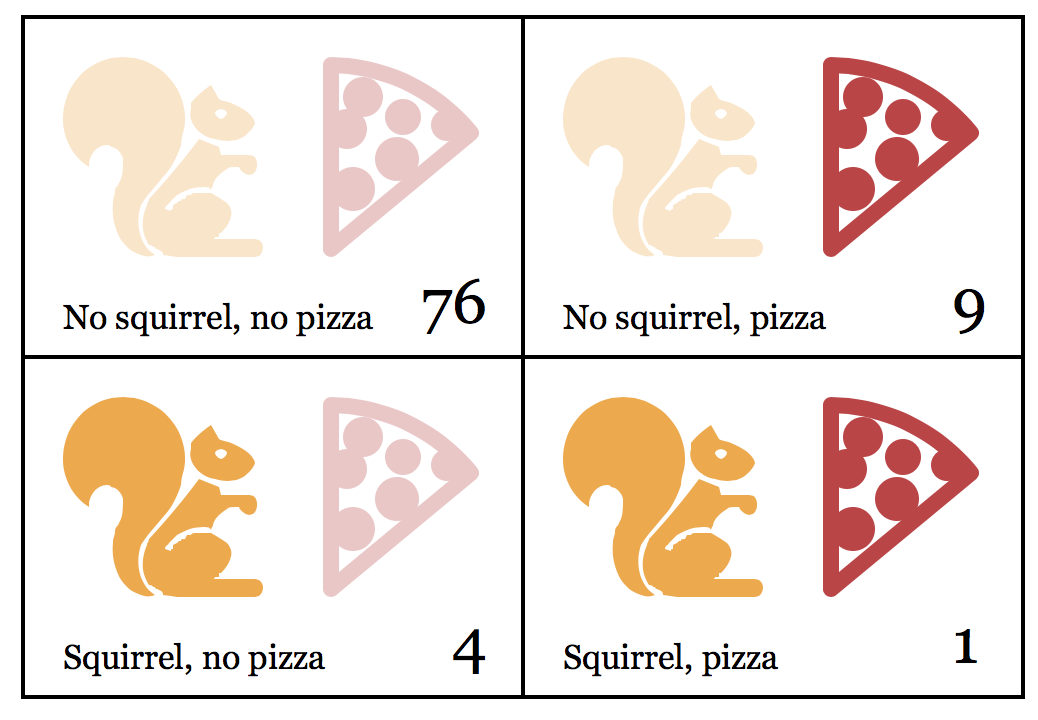
"touched tree", "brushed teeth"], false);

**Correlation** is een maat voor de afhankelijkheid tussen statistische variabelen. Correlation between variables is usually expressed as a value that ranges from -1 to 1.

0 correlation means the variables are not related.

1 indicates that the two are perfectly related.

-1 means that the variables are perfectly related but that they are opposites—when one is true, the other is false.



|  |  |
| --- | --- |
| *ϕ* = | *n*11*n*00 − *n*10*n*01  √ *n*1•*n*0•*n*•1*n*•0 |

0 false

1 true

The value *n*1• refers to the sum of all measurements where the first variable is true

1×76−4×9 = 40

5×85×10×80

*ϕ* ≈ 0.069

function phi(table) {

return (table[3] \* table[0] - table[2] \* table[1]) /

Math.sqrt((table[2] + table[3]) \*

(table[0] + table[1]) \*

(table[1] + table[3]) \*

(table[0] + table[2]));

}

console.log(phi([76, 9, 4, 1]));

// → 0.068599434

Math.sqrt berekend de wortel

tableFor figures out which box in the table each journal entry falls into by checking whether the entry contains the specific event it’s interested in and whether the event happens alongside a squirrel incident. The loop then adds one to the correct box in the table.

function tableFor(event, journal) {

let table = [0, 0, 0, 0];

for (let i = 0; i < journal.length; i++) {

let entry = journal[i], index = 0;

if (entry.events.includes(event)) index += 1;

if (entry.squirrel) index += 2;

table[index] += 1;

}

return table;

}

console.log(tableFor("pizza", JOURNAL));

// → [76, 9, 4, 1]

**Index** is de plaats in de array

**includes method** that checks whether a given value exists in the array.

for (let i = 0; i < JOURNAL.length; i++) {

let entry = JOURNAL[i];

// Do something with entry

}

Nu dezelfde code maar dan modern geschreven (meer info H6.) the word of after a variable definition, it will loop over the elements of the value given after of

for (let entry of JOURNAL) {

console.log(`${entry.events.length} events.`);

}

Find every type of event.

function journalEvents(journal) {

let events = [];

for (let entry of journal) {

for (let event of entry.events) {

if (!events.includes(event)) {

events.push(event);

}

}

}

return events;

}

console.log(journalEvents(JOURNAL));

see all the correlations

for (let event of journalEvents(JOURNAL)) {

console.log(event + ":", phi(tableFor(event, JOURNAL)));

}

// → carrot: 0.0140970969

// → exercise: 0.0685994341

// → weekend: 0.1371988681

// → bread: -0.0757554019

// → pudding: -0.0648203724

// and so on...

Show only correlations greater than 0.1 or less than -0.1.

for (let event of journalEvents(JOURNAL)) {

let correlation = phi(tableFor(event, JOURNAL));

if (correlation > 0.1 || correlation < -0.1) {

console.log(event + ":", correlation);

}}

// → weekend: 0.1371988681

// → brushed teeth: -0.3805211953

// → candy: 0.1296407447

Uitkomst

for (let entry of JOURNAL) {

if (entry.events.includes("peanuts") &&

!entry.events.includes("brushed teeth")) {

entry.events.push("peanut teeth");

}

}

console.log(phi(tableFor("peanut teeth", JOURNAL)));

// → 1

**Unshift** add thing to the array in front of the queue

**shift** gets and removes things from the array

let todoList = [];

function remember(task) {

todoList.push(task);}

function getTask() {

return todoList.shift();}

function rememberUrgently(task) {

todoList.unshift(task);}

remember("rabarber")

remember("vis")

remember("pollen")

rememberUrgently("vis")

todoList

(4) ["vis", "rabarber", "vis", "pollen"]

getTask()

"vis"

todoList

(3) ["rabarber", "vis", "pollen"]

**indexOf**  method: noemt de plaats van de array of letter bij een string.

console.log([1, 2, 3, 2, 1].indexOf(2));

// → 1

console.log("coconut".indexOf("u"));

// → 5

**lastIndexOf** is hetzelfde maar zoekt vanaf achteraan

**Slice** method : geeft getallen die tussen de slice (array)nummers. Laatste nummer is exclusief. Wanneer er geen laatste arraynummer wordt gegeven, krijg je alle nummers

console.log([10, 11, 12, 13, 14].slice(2, 4));

// → [12, 13]

console.log([10, 11, 12, 13, 14].slice(2));

// → [12, 13, 14]

console.log("coconuts".slice(4, 7));

// → nut

**Concat()** method: glue arrays together to create a new array

function remove(array, index) {

return array.slice(0, index)

.concat(array.slice(index + 1));

}

console.log(remove(["a", "b", "c", "d", "e"], 2));

// → ["a", "b", "d", "e"]

**Trim** method: removes whitespace

console.log(" okay \n ".trim());

// → okay

**padStart** takes the desired length and padding character as arguments.

console.log(String(6).padStart(3, "0"));

// → 006

**Split** method. Split de woorden uiteen

**Join** zet de woorden week aan elkaar

let sentence = "Secretarybirds specialize in stomping";

let words = sentence.split(" ");

console.log(words);

// → ["Secretarybirds", "specialize", "in", "stomping"]

console.log(words.join(". "));

// → Secretarybirds. specialize. in. Stomping

**Repeat** method

console.log("LA".repeat(3));

// → LALALA

**Pak een letter uit een string**

console.log(string[1]);

// → b

**…** om een array aan te geven of aan te roepen

let words = ["never", "fully"];

console.log(["will", ...words, "understand"]);

// → ["will", "never", "fully", "understand"]

function max(...numbers) {}

console.log(max(4, 1, 9, -2));

**Math.random** Getal tussen 0 en 1

console.log(Math.random());

// → 0.36993729369714856

**Math.floor** rond het nummer af

console.log(Math.floor(Math.random() **\* 10**));

// → 2

**Math.ceil** naar boven afronden

**Math.round** afronden naar dichtsbijgelegen

**Math.abs**

**serialize the data** = if you want data in a file for later or send it to another computer converted into a flat description

dat kan met JSON

**JSON.stringify**

**JSON.parse**

let string = JSON.stringify({squirrel: false,

events: ["weekend"]});

console.log(string);

// → {"squirrel":false,"events":["weekend"]}

console.log(JSON.parse(string).events);

// → ["weekend"]

var word = ‘bootcamp’;

word.length;

// 8

H5.

**Abstractions** hide details and give us the ability to talk about problems at a higher (or more abstract) level. It is a useful skill, in programming, to notice when you are working at too low a level of abstraction.

Voorbeeld:

console.log(sum(range(1, 10)));

de loop wordt hier bijvoorbeeld weggelaten.

**higher-order functions**. Functions that operate on other functions, either by taking them as arguments or by returning them. Handig bij samenstellen van operaties. One area where higher-order functions shine is data processing.

**Higher-order functions** allow us to abstract over actions, not just values. They come in several forms. For example,

* **we can have functions that create new functions.**

function greaterThan(n) {

return m => m > n;

}

let greaterThan10 = greaterThan(10);

console.log(greaterThan10(11));

// → true

* **And we can have functions that change other functions.**

function noisy(f) {

return (...args) => {

console.log("calling with", args);

let result = f(...args);

console.log("called with", args, ", returned", result);

return result;

};

}

noisy(Math.min)(3, 2, 1);

// → calling with [3, 2, 1]

// → called with [3, 2, 1] , returned 1

**Merk op dat als er een tweede parameter wordt meegegeven** noisy(Math.min)(3, 2, 1);

**deze doorschuift. In dit geval …args**

**forEach** array method. Loop as a higher-order function.

["A", "B"].forEach(l => console.log(l));

// → A

// → B

**Filters out the elements in an array that don’t pass a test.**

Merk op dat er een functie aan een argument wordt gegeven.

Dit is een **pure function.** Het wijzigt niet de bestaande array maar bouwt een nieuwe.

function filter(array, test) {

let passed = [];

for (let element of array) {

if (test(element)) {

passed.push(element);

}

}

return passed;

}

console.log(filter(SCRIPTS, script => script.living));

// → [{name: "Adlam", …}, …]

The **filter()** method creates a new array with all elements that pass the test implemented by the provided function.

var words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter( i => i.length > 6);

console.log(result);

// expected output: Array ["exuberant", "destruction", "present"]

i is in dit geval dus een waarde uit de array

\_\_

function map(array, transform) {

let mapped = [];

for (let element of array) {

mapped.push(transform(element));

}

return mapped;

}

let rtlScripts = SCRIPTS.filter(s => s.direction == "rtl");

console.log(map(rtlScripts, s => s.name));

// → ["Adlam", "Arabic", "Imperial Aramaic", …]

The **map()** method creates a new array with the results of calling a provided function on every element in the calling array.

var array1 = [1, 4, 9, 16];

// pass a function to map

const map1 = array1.map(x => x \* 2);

console.log(map1);

// expected output: Array [2, 8, 18, 32]

**Reduce** geeft een waarde uit een array door elke waarde langs te gaan (higher-order operation) je hebt nodig:

- array

- variabele met de functie reducer

- **Accumulator (acc)** onthoudt de waarde voor elke iteratie in de array

**- Current Value (cur)** het huidige element uit de array

**Optellen met reducer**

const array1 = [1, 2, 3, 4];

const reducer = (accumulator, currentValue) => accumulator + currentValue;

console.log(array1.reduce(reducer, 0));

// 0 + 1 + 2 + 3 + 4

Merk op dat de 0 in dit geval er bij wordt opgeteld. Dit is het start getal. Zie ook volgend vb. Als de array waardes bevat kan dit ook worden weggelaten. Dan begint de telling bij het eerste element.

function reduce([1, 2, 3, 4], (a, b) => a + b, start) {

let current = start;

for (let element of [1, 2, 3, 4]) {

current = function (a, b) => current + element;

}

return current;

}

console.log(reduce([1, 2, 3, 4], (a, b) => a + b, 0));

// → 10

\_\_

Vind het gemiddelde jaar van bestaande en niet bestaande scripts.

Start with all scripts, filter out the living (or dead) ones, take the years from those, average them, and round the result.

function average(array) {

return array.reduce((a, b) => a + b) / array.length;

}

console.log(Math.round(average(

SCRIPTS.filter(s => s.living).map(s => s.year))));

// → 1188

console.log(Math.round(average(

SCRIPTS.filter(s => !s.living).map(s => s.year))));

// → 188

Kan ook zo:

let total = 0, count = 0;

for (let script of SCRIPTS) {

if (script.living) {

total += script.year;

count += 1;

}

}

console.log(Math.round(total / count));

// → 1188

\_\_\_

**Uit de les** voorbeeld van sporters die aanwezig waren

var topAttendee = players.reduce(function(acc, cur) {

if(!acc) {

return cur;

}

if(cur.attendence > acc.attendence) {

return cur;

}

}, null);

**some() method** test of een element in de array de test heeft gehaald (true)

function isNegative(element, index, array) {

return element < 0;

}

const int8 = new Int8Array([-10, 20, -30, 40, -50]);

const positives = new Int8Array([10, 20, 30, 40, 50]);

console.log(int8.some(isNegative));

// expected output: true

console.log(positives.some(isNegative));

// expected output: false

\_\_

JavaScript-strings zijn gecodeerd als een reeks van 16-bits getallen. Dit worden Code-units genoemd. Omdat de codeuints in een eenheid moesten passen die te klein was is UTF-16 uitegvonden. Het bevat een hoop fouten. Er moeten bijvoorbeeld 2units characters worden gebruikt, gelukkig wordt dat nu vaker gebruikt.

charCodeAt() method geeft de code unit, een getal tussen 0 en 65535

codePointAt() method geeft een Unicode character

Om alle tekens in een tekenreeks te doorlopen, moeten we ons nog steeds bezighouden met de vraag of een teken een of twee code-eenheden in beslag neemt.

// Two emoji characters, horse and shoe

let horseShoe = "🐴👟";

console.log(horseShoe.length);

// → 4

console.log(horseShoe[0]);

// → (Invalid half-character)

console.log(horseShoe.charCodeAt(0));

// → 55357 (Code of the half-character)

console.log(horseShoe.codePointAt(0));

// → 128052 (Actual code for horse emoji)

\_\_

let roseDragon = "🌹🐉";

for (let char of roseDragon) {

console.log(char);

}

// → 🌹

// → 🐉

\_\_

## Recognizing tekst (2e stuk overgeslagen)

**findIndex() method**: like indexOf, but instead of looking for a specific value, it finds the first value for which the given function returns true. Like indexOf, it returns -1 when no such element is found.

function countBy(items, groupName) {

let counts = [];

for (let item of items) {

let name = groupName(item);

let known = counts.findIndex(c => c.name == name);

if (known == -1) {

counts.push({name, count: 1});

} else {

counts[known].count++;

}

}

return counts;

}

console.log(countBy([1, 2, 3, 4, 5], n => n > 2));

// → [{name: false, count: 2}, {name: true, count: 3}]

H6.

Object-oriented programming, a set of techniques that use objects (and related concepts) as the central principle of program organization.

**Note**: Methods are nothing more than properties that hold function values.

let rabbit = {};

rabbit.speak = function(line) {

 console.log(`The rabbit says '${line}'`);

};

rabbit.speak("I'm alive.");

// → The rabbit says 'I'm alive.