Bronnen – dag 2

Functions; parameters and arguments

The Coding Train - Function Basics (video)

p5 library: bestaat uit vaste woorden voor JS, zoals function, var, if, background()

background(0) = zwarte achtergrond

als dit in een functie staat, maar de functie wordt nog aangeroepen, wordt de achtergrond alsnog zwart, omdat background() deel is van de p5 library

hiervoor heeft background wel per se een argument nodig

modularity

opdelen in meerdere functies om overzicht in je code te brengen

reusability

→ arguments parameters

The Net Ninja - Modern JavaScript functions (video)

function expression

let hier op de puntkomma aan het einde:

```
const speak = function() {
};
```

calling deze function:

```
speak();
```

function expressions worden niet gehoist, dus je kunt niet de functie callen voordat je de functie hebt geexpressed

handig om function expressions te gebruiken, want dan wordt je verplicht om eerst te callen, wat je code leesbaarder maakt

parameters

```
function (name) {
}
```

name = local (binnen functie) variabele aanmaken die binnen dit code block (van de function) kan wordt gebruikt

callen kan alleen als je bij het callen de parameter een value geeft

er kan ook een default value worden gebruikt in een parameter (dan hoef je ook geen value per se in te vullen bij het callen:

```
function (name = 'mario') {
}
```

arrow function

```
const calcArea = (radius) => {
      return 3.14 * radius**2;
};
```

of

```
const calcArea = () => {
};
```

of (met één parameter kun je de haakjes weghalen + single line, dus de return kan ook weg):

```
const calcArea = radius => 3.14 * radius**2;
```

callback functions

```
let people = ['mario', 'luigi', 'bowser'];

const logPerson = (person, index) => {
      console.log('${index} - hello ${person}');
};

people.forEach(logPerson);

// -> 0 - hello mario
1 - hello luigi
2 - hello bowser
```

Eloquent JavaScript - Functions (book)

Bindings and scopes

```
const halve = function(n) {
  return n / 2;
};

let n = 10;
console.log(halve(100));
// → 50
console.log(n);
// → 10
```

(verschillen tussen var, let en const: JavaScript Let, Const & Var: A Complete Guide)

arrow functions (alleen ES6)

The arrow comes *after* the list of parameters and is followed by the function's body. It expresses something like "this input (**the parameters**) produces this result (**the body**)".

call stack

The place where the computer stores this context is the *call stack*. Every time a function is called, the current context is stored on top of this stack. When a function returns, it removes the top context from the stack and uses that context to continue execution.

calling too little parameters

```
function minus(a, b) {
  if (b === undefined) return -a;
  else return a - b;
}

console.log(minus(10));
// → -10
console.log(minus(10, 5));
// → 5
```

closure
= being able to reference a specific instance of a local binding in an enclosing scope

In the example, `multiplier` is called and creates an environment in which its `factor` parameter is bound to 2. The function value it returns, which is stored in `twice`, remembers this environment. So when that is called, it multiplies its argument by 2. (zie hieronder)

```
function multiplier(factor) {
  return number => number * factor;
}

let twice = multiplier(2);
console.log(twice(5));
// → 10
```

recursive functions
= a function that calls itself

```
function findSolution(target) {
  function find(current, history) {
    if (current == target) {
      return history;
    } else if (current > target) {
      return null;
    } else {
      return find(current + 5, `(${history} + 5)`) ||
             find(current * 3, `(${history} * 3)`);
    }
  }
  return find(1, "1");
}

console.log(findSolution(24));
// → (((1 * 3) + 5) * 3)
```

A useful principle is to not add cleverness unless you are absolutely sure you're going to need it. It can be tempting to write general "frameworks" for every bit of functionality you come across. Resist that urge. You won't get any real work done—you'll just be writing code that you never use.

Exercises

Recursion

```
// Your code here.
function isEven(number) {
  if (number%2 == true) {
      return false;
  }
  else if (number < 0) {
      return "??";
  }
  else {
      return true;
  }
}
console.log(isEven(50));
// → true
console.log(isEven(75));
// → false
console.log(isEven(-1));
// → ??
```

Bean counting

You can get the Nth character, or letter, from a string by writing `"string"[N]`. The returned value will be a string containing only one character (for example, `"b"`). The first character has position 0, which causes the last one to be found at position `string.length - 1`. In other words, a two-character string has length 2, and its characters have positions 0 and 1.

Write a function `countBs` that takes a string as its only argument and returns a number that indicates how many uppercase “B” characters there are in the string.

Next, write a function called `countChar` that behaves like `countBs`, except it takes a second argument that indicates the character that is to be counted (rather than counting only uppercase “B” characters). Rewrite `countBs` to make use of this new function.

```
// Your code here.

console.log(countBs("BBC"));
// → 2
console.log(countChar("kakkerlak", "k"));
// → 4
```

hint:

Your function will need a loop that looks at every character in the string. It can run an index from zero to one below its length (`< string.length`). If the character at the current position is the same as the one the function is looking for, it adds 1 to a counter variable. Once the loop has finished, the counter can be returned.

Take care to make all the bindings used in the function *local* to the function by properly declaring them with the `let` or `const` keyword.

Higher-order functions

[The Coding Train - Higher Order Functions (video)]

higher order functions

= functies die óf een functie als argument gebruiken, óf een functie als output geven (return …)

bijv. map(), sort(), reduce(), filter()

[Eloquent JavaScript - Higher Order (book)]

filtering arrays

filter *loopt* door de hele array

```
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", …}, …]
```

transforming with map

```
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", …]
```

summerizing with reduce

```
function reduce(array, combine, start) {
  let current = start;
  for (let element of array) {
    current = combine(current, element);
  }
```

```
  return current;
}

console.log(reduce([1, 2, 3, 4], (a, b) => a + b, 0));
// → 10
```

The `some` method tests whether any element matches a given predicate function. And `findIndex` finds the position of the first element that matches a predicate.

[Higher-order functions - Part 1 of Functional Programming in JavaScript]

filtering arrays

```
var isDog = function(animal) {
     return animal.species === 'dogs';
};

var dogs = animals.filter(isDog);
var otherAnimals = animals.reject(isDog);
```