**JS From zero to hero**

**Datatypes**

There are different types of **values**

Firstly: **Objects**

secondly: **primitive datatypes** (everything else)

there are 7 types of **primitive datatypes**:

**1 number**these are ALWAYS floating point numbers

**2 string**sequence of characters. Always put them in paranthesese

**3 boolean**

True or false.

**4 undefined**an empty value. A value that has not yet been assigned a value.

**5 Null**also means empty value but a bit different

**6 symbol**Symobl. Value that is unique and cannot be changed.

**7 BigInt**Very large integers. Larger than number could hold.

Javascript has dynamic typing. This means that The Value has Type, not the variable. This also means that any variable we store can have its type changed later in the porgram.

**Let, const and var**

Let and const are modern javascript and var is old.

**Let** are mutable. (let age. Because age can change)

**Const** is immutable (const birthyear for example cannot change)  
the course and general proper practice is that any value should be initiallized as a const and **only** when you it turns out you want it to be mutable, than turn it into a **let.**

**Var.** This is the old way of creating variables and is a lot like **let. Let** is block scoped and **var** is function-scoped. **We should never use** var though.

**Basic operators**

Basic stuff. You know this

**Operator precedence**

Google mdn operator precedence.

Hoe hoger het getal hoe hoger de precedence. Let ook op de middelste column die aangeeft of code van links-rechts gaat of rechts-links gaat.

**String and Template literals**

**Type Conversion and Coercion**

Conversion = when we manually converse types.

Coercion is when javascript does it (accidentily) behind the curtains

You can convert numbers, strings and booleans to each other. Converting anthing to undefined or null doesnt make sense.

**Truthy and Falsy statements**

Or conversion with booleans,  
5 falsy values. Values that are not exactly false but turn false when we convert them to boolean.

The 5 falsy values are: 0, ‘’ (empty string), undefined, null, NaN

These are not false but become false when converted to boolean.

Any positive number or filled string becomes true when converted to a boolean

**Equality Operators == vs ===**

=== strict equality operator. Only returns true when both sides are EXACTLY the same.

== loose equality operator. It will use typecoercion to make things similar.

This is confusing for many people. Especially the loose operator results in many strange bugs.

**Boolean logic**

And, Or and NOT operators

**The Switch Statement**

Switch vs else if basically

**Statements and expressions**

An expression is anything that produces a value.

**The Conditional**

**Functions overview**

3 types of functions

1. function declaration – a function that can be used before its declared.

*function* percentageOfWorld1(*populationMillions*) {

return (populationMillions / 7900 \* 100);

}

1. Function expression – Essentially a function *value* stored in a variable

*const* percentageOfWorld2 = *function* (*populationMillions*) {

    return (populationMillions / 7900 \* 100);

}

1. Arrow function. Great for a quick one-line function.

*const* percentageOfWorld3 = *populationMillions* =>

   populationMillions / 7900 \* 100;

**92 Scoping and scope in javascript**  
  
Scope  
How our progams variables are organized and accessed

Lexical scoping  
Scoping is controlled by placement of function and blocks in code

Scope  
Space or environment in which a certain variable is declared. There is **global** scope, function scope and block scope.

scope of a variable  
Region of our code where a certain variable can be declared.

99 primitives vs objects   
AKA  
storing in the call stack vs storing in the heap

A screenshot of a computer

Description automatically generated

Section 9

103 destructuring arrays

Destructuring arrays is unpacking an arrays in different variabled.

!!!  
remember declaring a variable creates the variabled. It declares to exist  
 initalizing a variable, is assigning a value to a variable (for the first time)

So: first you *declare*, than you initalize (or you do both at the same time)

Arrays : [ ]

Objects: { }

104 destructuring object:

Pretty coplex. Or badly explained

**105 Spread operator**

Basically if youra rray is named Array1.

And you type ...Array1 it will unpack the entire array.

This spread operator ... works on ANY iterables:

Arrays, strings, maps, sets etc

It really helps when you dont know your array length!

107 && and shortcircuiting

|| OR will return the first truthy value

If everything is falsy it will return the last value (which is falsy)  
**Practically this is used to set default values**

&& If everything is truthy, it will return the last value.

If anything is falsy it return this falsy value.  
Practically, you can execute some code when the first operand is true.

108 Nullish coalescent operator

Is literally the same as the || or operator except is works with Nullish values rather than Falsy values  
Falsy values: Null + undefined + 0 + ‘ ‘ (empty string)

Nullish values: Null + undefined (not 0 and ‘ ‘ (empty string)

// 109 Logical Assignment operators

Very new and modern 3 new operators

restaurant.numGuests = 23;

const guests1 = restaurant.numGuests ? restaurant.numGuests :10;

console.log(guests1);

==

const guests2 = restaurant.numGuests||10;

console.log(guests2);

==

rest1.numGuests ||= 10;

//FIRSTLY based of  || -> the OR assignment operator

// Using the || operator to set a default

rest2.numGuests = rest2.numGuests || 10

rest1.numGuests = rest1.numGuests || 10

// This is literally the same but even Shorter!

rest1.numGuests ||= 10;

rest2.numGuests ||= 10;

console.log(rest1);

console.log(rest2);

//SECONDLY variant of ?? -> Nullish assignment operator

rest1.numGuests ??= 10;

rest2.numGuests ??= 10;

console.log(rest1);

console.log(rest2);

// Thirdly:variant of && -> the AND assignment operators

// If the restaurant has an owner, we want to replace it with the string anonymous

rest1.owner = rest1.owner && '<ANONYMOUS>';     // returns undefined, because rest1.owner is undefined

rest2.owner = rest2.owner && '<ANONYMOUS>';     // returns anonymous, because rest2.owner is truthy

rest1.owner &&= '<ANONYMOUS>'

rest2.owner &&= '<ANONYMOUS>'

113 optional chaining

Restaurant.mon?.open?

De? Is basically een optional check.

De ? checkt of het element links bestaat. Als dat niet zo is, dan wordt het gedeelte rechts ook niet gelezen/uitgevoerd

De optional chain wordt bijna altijd gebruikt in combinatie met de nullish asignment operator. Like zo:

Console.log(users[o]?.name?) ?? “Array is empty”

Als users array een 1e entree heeft en die heeft een naam. Dan is alles ok.   
Maar als een van beide niet bestaat, dan krijg je bericht: “Array ios empty” terug. Dit is super handig en nuttig.

114 Looping Objects, keys, values and entries

An entrie exist of a key/value pair. It can be used to loop over objects.

Object.keys(days) // day is a list of days the restaurant is open

Returns

116 SETS

A set is a collection of unique values (no duplicates). You can add duplicates and it will work, but it won’t actually show the same value twice.

This work with any **iterable**. So making a set of a word just pulls the word apart in different letters.

Conclusion at the end. Sets are very similar to arrays, but they are handy if you need to have unique values. Otherwise, just keep using arrays.

117 MAPS

Maps are a lot more useful than sets. Joans said: maps are very powerful. They store information in key-value pairs, like objects, however contrary to objects map can store **keys** of any type even objects (whereas objects virtually always have string values.)

119 **Which data structure should to use**

**Where can data come from:**

* from the program itself. Data written in the source code, status messages
* From the UI. Data input from the user or information from the DOM
* External sources – a web api
* This often results in **COLLECTIONS OF DATA**
* **DATA STRUCTURES** is where we store these collections.

**Simple list? 🡪 Array or sets**

**Key/value pairs 🡪 Objects or Maps (keys allow us to describe values**

Firstly the simple structures of values

**Arrays** use when you need ordered list of values (might contain duplicates

Use when you need to manipulate data

**Sets** Use when you need to work with unique values

Use when high performance is important

Use to remove dupicates from arrays

Sets are not meant to replace arrays, instead they are meant to complement each other.

Next objects and maps, when we need to describe values using keys

**Objects METHODS** use when you need **METHODS**

**JSON** use when you need to use json

more traditional key/value store (“abused” objects)  
Easy to write and access values with . and []

**Maps** Use when you simply need to map keys to values

Use when you need keys that are not strings

Better performance  
Keys can have **any datatype**Easy to iterate   
Easy to compute size

A screenshot of a computer

Description automatically generated

**129 Default parameters**

One can easily define default parameters for function in the function definition like so:

const createBooking = function (

  flighNum,

  numPassengers = 1,

  price = 199 \* numPassengers

) {

  Code

  };

};

createBooking('LH123');

createBooking('LH123', 2, 800);

createBooking('LH123', undefined, 1000);

**130 passing arguments value vs reference**

You know this already:

If you call a primitive value, then you create a copy that you can manipulate.   
However if you call a complex value like a value on an object, and you manipulate it than you actually change the value on the object.

**131 first class vs higher order functions**

Little theory lesson:

First class functions:  
Js has first class orders, which means that functions are technically values and can be treated and used as values.

Higher order functions:  
Are functions that can take, or return, functions.

**132 Functions accepting callback functions (functions returning functions)**

const oneWord = function (str) {

  return str.replace(/ /g, '').toLowerCase();

};

const upperFirstWord = function (str) {

  const [first, ...others] = str.split(' ');

  return [first.toUpperCase(), ...others].join(' ');

};

// higher-order function

const transformer = function (str, fn) {

  console.log(`Original string: ${str}`);

  console.log(`Transformed string: ${fn(str)}`);

  console.log(`Transformed by: ${fn.name}`); // name is a baked in property of methods

};

Both first order functions do something similar, namely transform a string.  
The transformer will take either of these functions and transform the string appropriately. The transformer itself doesn’t know anything about the functions.

addEventListener is also a higher order functions. Cool right!

const high5 = function () {

  console.log('👋');

};

document.body.addEventListener('click', high5);

and here the method gets return for each string in the array.

['Harry', 'Pieternella', 'Pomni'].forEach(high5);

**133 functions returning functions**

Deze functie returns a function

const greet = function(greeting) {

    return function(name) {

        console.log(`${greeting} ${name}`);

    }

}

So you can use it like this.

greet('Hello')('Rooie Rinus');

je kunt ook de gereturnde function opslaan als een value met een soort standaard value eraan gehangen:

const greeterHey = greet('Hey');

greeterHey('Harry');

greeterHey('Rinus');

**134 the call and apply methods**

We can use the call method to specift the ’.this’ keyword for a method.

This object had the book method.

const lufthansa = {

    airline: 'Lufthansa',

    iataCode: 'LH',

    bookings: [],

    book(flighNum, name) {

    console.log(`${name} booked a seat on ${this.airline} flight ${this.iataCode}${flighNum}`);

    this.bookings.push({flight: `${this.iataCode}${flighNum}`, name})

    },

}

If we want to use this book function on an object that doesn’t have this function:

const eurowings = {

    airline: 'Eurowings',

    iataCode: 'EW',

    bookings: [],

    // we want the same method here. Butcopying the method is bad practice.

};

We can do

book.call(eurowings)

And the ‘.this’ keyword will be set to eurowings.

And you can also set the other parameters like so:

book.call(eurowings, 23, 'Histibe Rachelvoet ');

**the apply method** does something similar but it accepts an array of values.

const flightdata = [555, 'Trudy Griezel'];

book.apply(swiss, flightdata);

this is however not really used anymore, because we can just use the call method and destructure the array:

book.call(swiss, ...flightdata);

**135 the bind method**

The bind method is very similar to the call method with one big difference

The call method immediately calls the functions

The bind method returns a new function with the newely set ‘.this’ keyword.

So call performs the functions, bind returns a new, more specific, function.

So you can create many similar functions for different objects, without rewrighting the same method for each object (which is bad practice).

const bookEW = book.bind(eurowings);

const bookLM = book.bind(lufthansa);

const bookLX = book.bind(swiss);

and even set other defaults like flightnumber:

const bookEW23 = book.bind(eurowings, 23);

bookEW23('Sjenkie Horgel')  //so we only need to add the name of the passenger :)

this below doesn’t work. Even though Lufthansa ‘owns’ the original method. This is because the addEventListener sets the ‘this’ keyword to the element its listening for (the buy button in this case)

document.querySelector('.buy').addEventListener('click', lufthansa.buyPlanes

So we need to use the bind function to set a new ‘this’ keyword. Like so:

document.querySelector('.buy').addEventListener('click', lufthansa.buyPlanes.bind(lufthansa));

in this case we have a general calculate price after tax method:

const addTax = (rate, value) => value + value \* rate;

We can use bind to make this method more specific. For example make a ‘btw’ calculator function.

const addVAT = addTax.bind(null, 0.23)

The first parameter is the new ‘this’ keyword, however there is no relevant this, so we use ‘null’ instead. The second parameter is actually the first parameter of the original method.x`

# A closer look at Arrays

**Simple array methods**

.Slice(a, b) Returns a shallow copy of a portion of an array into a new array object. The slice starts **before** indexa and ends **before** indexb. (slice (2,4) would return an array with indexes 2 & 3.

arr.Splice(a, b) Similar to splice except. It takes a portion from the original array and returns that as a new array. However the spliced part is removed from the original array. Slice(2,4) would remove 2&3 from the original array and return a new array {2,3}.

Arr.splice(a, b, ’c’) replaces everything between index a and b, with String c.

.reverse Reverses all values position in an array *in place* meaning the original array is mutated.

.toReverse Reverses all values position in an array and returns them as a new array. Leaving the original array intact

.concat arr3 = arr1.concat(arr2) combines two arrays into a third new array. This leaves the original arrays intact

.join(‘a’) joins all elements in a new string with string a in between each element.

.push(‘a’) adds string a to the end of the array.

.pop removes **and return**s the last element from an array.

.shift removes and returns the first element from an array.

.unshift(1) adds value 1 as a new element to the beginning of an array. More values can be added.

.indexOf(‘Cavia’) Returns the index of an arrayelement where the value is ‘Cavia’. Returns -1 if value isnt present.

.indexof(‘Cavia’, 2) Returns the index of the second array element where the value is ‘Cavia.

.includes(‘a’) returns either true or false wether ‘a’ is present as a value in the array.

Arr[a] returns value at index a. arr[arr.length-1] = last value in array

Arr.at(0) does the same. Arr.at(-1) = last value in array. -2 second to last etc.

'Harry'.at(0) strings are arrays too.

**LOOPS**

**forOf usecase: breaking out of a loop.**

for (const movement of movements) {

Do something to each element in the movements array.

for (const [i, movement] of movements.entries()) {

to use the index and value you need to specifically assign them

**forEach usecase: whenever you dont need to break out. (its simpler**

movements.forEach(function(movement) {

do something for each element in the movements aray using a callback function.

movements.forEach(function(movement, i, arr) {

the foreach methods function has (value, index, fullArray) baked in.

**foreach with maps**

currencies.forEach(function(value, key, map){

A map is an array of key value pair. It works similar

**Foreach with sets**

currenciesUnique.forEach(function(value, key, map) {

key == value. Becasue a set is a unique collection indexes are unnecessary.

**.map** Creates a new array populated with the results of calling a provided function on every method of the calling array

const movementsDescriptions2 = movements.map((mov, i, arr) => {

  return `Movement ${i+1} You ${mov >= 0 ? `deposited` : `withdrew`} ${Math.abs(mov)}`

});

Returns a string using index for sequence number. And decides wether the value is positive of negatice and uses if else statement to decide depo or withdr.

**.filter** creates a **shallow** copy of a portion of the given array, filtered down to just the elements that pass the test implemented by the provided function.

const withdrawals = movements.filter(

  mov => mov > 0

)

Filters returns an array with only positive values.

**.reduce** Returns a single value. It executes a “reducer” callback function on each element of the array. Passing the return value of the current calculation to the next calculation.

const maxValue = movements.reduce((acc, cur, i, arr) => cur >= acc ? cur : acc, movements[0]);

method that returns the highest value in the array.