Javascript Principles

The perfect start for React, Angular and Vue

Javascript traineeship 2021 - Charlotte Huygen

_Charlotte Huygen

Frontend developer

Age:	28 years old
Domicile:	Willebroek, Antwerp
Career:	Webdesigner for 1 year Axxes Consultant for nearly 4 years
Client:	Corilus 1.5 years DPGMedia 2 years
Hobbies	Doggy school with my dog Marley Gaming (when I can) Working out (same, when I can)

Welcome to Axxes!!!

Javascript traineeship 2021 - Charlotte Huygen

ES

ECMASCRIPT

Explain new features of the modern **Ecmascript** version.

TS

TYPESCRIPT

Features of the typed language called **Typescript**

_How will we do this?

_ Theory

Theoretical explanation with examples.

_ Conversation

Asking questions to keep you awake.

_ Exercises

Hands-on exercises to try out the theory yourselves.

YOU KNOW NOTHING
JON SNOW
memegenerator.net

ES

Ecmascript

ECMASCRIPT

European **C**omputer **M**anufacturers **A**ssociation

NETSCAPE

- Brendan Eich
- JS core features

STANDARDIZATION

✗ Every browser own version JS

✓ ECMA

1997

From this year forward the official name is **ECMA**script

ECMASCRIPT

Javascript is a **scripting** language invented to make webpages more **dynamically**. We could manipulate the DOM at runtime.

WHO ALREADY USED JAVASCRIPT?

WHICH SUBJECTS ARE WE GOING TO TOUCH?

> VAR, LET and CONST
> Hoisting
> Functions
> Arrow functions
> Template literals
> Object literals
> Object destructuring
> Classes
> Maps
> Loops
> Promises
> Async/Await

When you just graduated and you have to follow yet another course:

VAR, LET and CONST

VAR, LET and CONST

VAR (variable)

```
var x = 5;
```

Declaration variable Variable name = Value of variable;

VAR, LET and CONST

JAVASCRIPT DATATYPES

var x = 5;	Number
var x = "hello";	String
var x = true;	Boolean (true/false)
var x = 123470997y;	BigInt
var x = {};	Object
var x = Symbol("hello");	Symbol
var x;	undefined
var x = null;	null

VAR, LET and CONST

VAR (variable)

<CODE INPUT>

```
var x = 5;

console.log(x);
```

<CODE OUTPUT>

```
5
```

VAR, LET and CONST

VAR - The global scope

<CODE INPUT>

```
var x = 5;

if(true)  {

    x = 2;

}

console.log(x);
```

<CODE OUTPUT>

WHAT WILL THE OUTPUT BE?

VAR, LET and CONST

VAR - The global scope

<CODE INPUT>

```
var x = 5;

if(true)  {

    x = 2;

}

console.log(x);
```

<CODE OUTPUT>

```
2
```

VAR, LET and CONST

LET

```
var x = 5;

let x = 5;
```

VAR, LET and CONST

LET - The block scope

<CODE INPUT>

```
let x = 5;
if(true) {
    x = 2;
}

console.log(x);
```

<CODE OUTPUT>

WHAT WILL THE OUTPUT BE?

VAR, LET and CONST

LET - The block scope

<CODE INPUT>

```
let x = 5;
if(true) {
    x = 2;
}

console.log(x);
```

<CODE OUTPUT>

```
2
```

VAR, LET and CONST

LET - The block scope

<CODE INPUT>

```
if(true) {
    let x = 2;
}

console.log(x);
```

<CODE OUTPUT>

WHAT WILL THE OUTPUT BE?

VAR, LET and CONST

LET - The block scope

<CODE INPUT>

```
if(true) {
    let x = 2;
}

console.log(x);
```

<CODE OUTPUT>

```
Uncaught ReferenceError: x is not defined
```

VAR, LET and CONST

LET - The block scope

<CODE INPUT>

```
if(true) {

    let x = 5;
    x = 2;
    console.log(x);

}
```

<CODE OUTPUT>

```
2
```

VAR, LET and CONST

CONST

```
var x = 5;

const x = 5;
```

VAR, LET and CONST

CONST

<CODE INPUT>

```
const x = 5;
if(true)  {
    x = 2;
}

console.log(x);
```

<CODE OUTPUT>

WHAT WILL THE OUTPUT BE?

VAR, LET and CONST

CONST

<CODE INPUT>

```
const x = 5;
if(true)  {
    x = 2;
}

console.log(x);
```

<CODE OUTPUT>

```
Uncaught TypeError:
Assignment to constant
variable
```

VAR, LET and CONST

CONST

<CODE INPUT>

```
const x = 5;
if(true)  {
    console.log(x);
}

console.log(x);
```

<CODE OUTPUT>

```
5
5
```

VAR, LET and CONST

CONST - The block scope

<CODE INPUT>

```
if(true) {

    const x = 5;
    x = 2;
    console.log(x);

}
```

<CODE OUTPUT>

WHAT WILL THE OUTPUT BE?

VAR, LET and CONST

CONST - The block scope

<CODE INPUT>

```
if(true) {

    const x = 5;
    x = 2;
    console.log(x);

}
```

<CODE OUTPUT>

```
Uncaught

TypeError: Assignment to

constant variable
```

VAR, LET and CONST

TO CONCLUDE

VAR	LET	CONST
GLOBAL SCOPE	BLOCK SCOPE	BLOCK SCOPE
CAN REDECLARE	CAN REDECLARE	CANNOT REDECLARE

TEMPLATE LITERALS

TEMPLATE LITERALS

HOW TO USE VARIABLES IN STRINGS

<HOW IT WAS>

```
let name = "Charlotte";
let age = 27;

console.log("Hi, my name is " + name + " and I am " + age + " years old");
```

<OUTPUT>

```
Hi, my name is Charlotte and I am 28 years old
```

TEMPLATE LITERALS

HOW TO USE VARIABLES IN STRINGS

<WITH TEMPLATE LITERALS>

```
let name = "Charlotte";
let age = 27;

console.log(`Hi, my name is ${name} and I am ${age} years old`);
```

<OUTPUT>

```
Hi, my name is Charlotte and I am 27 years old
```

Multiline template literals

The hacky Javascript ways...

```
Let multiLine = "This is \
multiline"

Let multiLine = "This is"
+ "multiline"
```

```
console.log(`This is my first line.
This is my second line`)
```

Now smoother with template literals!

TEMPLATE LITERALS

ALSO POSSIBLE FOR CALCULATIONS!

<CALCULATIONS IN TEMPLATE LITERALS>

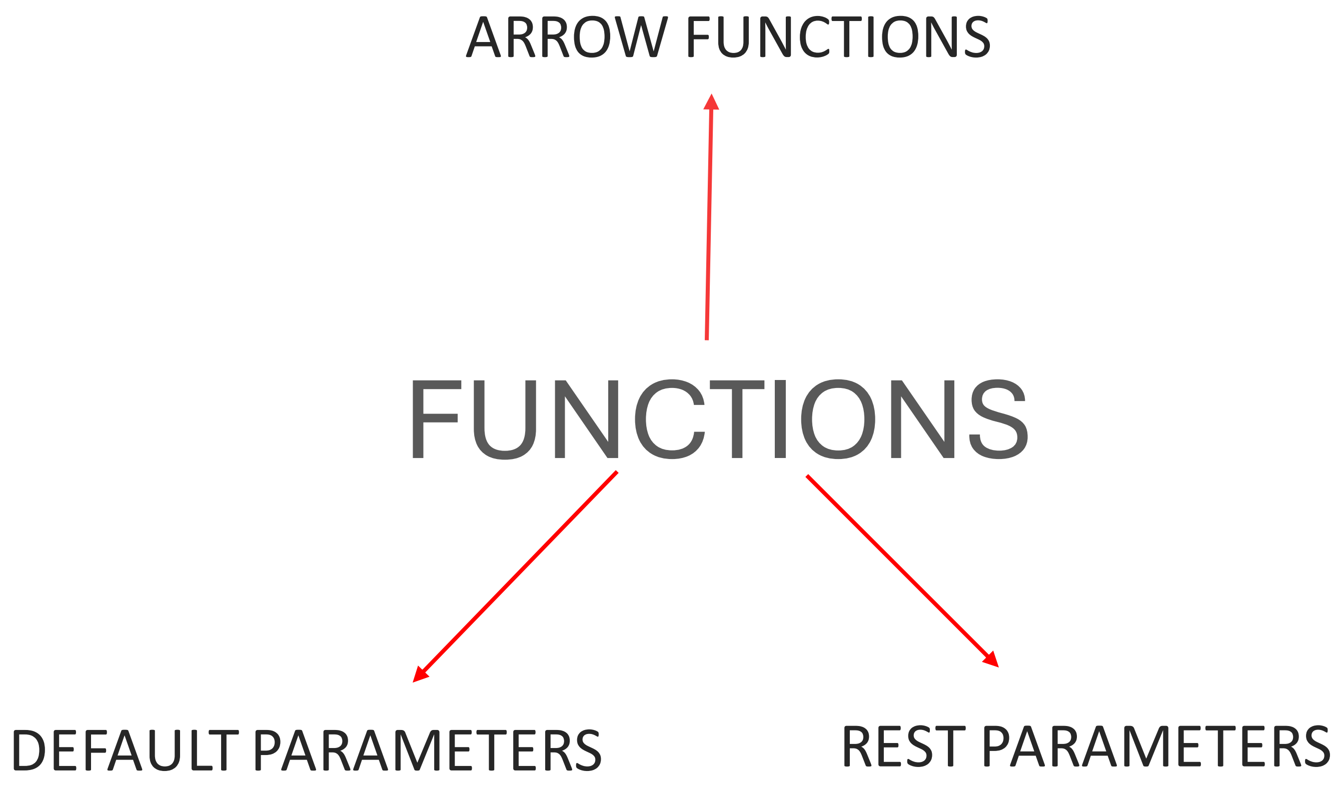
```
let num1 = 5;
let num2 = 5;

console.log(`A decade is ${num1 + num2} years long!`);
```

<OUTPUT>

```
A decade is 10 years long!
```

FUNCTIONS


ARROW FUNCTIONS
FUNCTIONS
DEFAULT PARAMETERS
REST PARAMETERS

FUNCTIONS

BUILD-UP

```
function name(x,y) {}
```

Declaration function Function name(Function parameters){}

Functions

Default parameters

```
function call(x = 5) {
    console.log(x);
}

call();
```

OUTPUT?

Functions

Default parameters

```
function call(x = 5) {
    console.log(x);
}

call();
```

```
5
```

Functions

Default parameters

```
function call(x, y = 4) {
    console.log(x + y);
}

call(2, 8);
call(7);
```

OUTPUT?

Functions

Default parameters

```
function call(x, y = 4) {
    console.log(x + y);
}

call(2, 8);
call(7);
```

```
10
11
```

Functions

REST parameter

```
function name(...REST) {}
```

Functions

REST PARAMETERS

```
function add(...numbers) {
    return numbers.reduce((a, b) => {
        return a + b
    });
}

console.log(add(1,2,3));
console.log(add(1,2,3,4))
```

```
6
10
```

Functions

REST PARAMETERS

WHAT THE HELL DOES THAT NUMBERS ARRAY LOOK LIKE THEN?

[1, 2, 3]

[1, 2, 3, 4]

let numbers = [1, 2, 3]

let numbers = [1, 2, 3, 4]

```
function add(...numbers) {
    return numbers.reduce((a, b) => {
        return a + b
    });
}

console.log(add(1,2,3));
console.log(add(1,2,3,4))
```

```
6
10
```

Functions

REST PARAMETERS

```
function info(name, age, ...hobbies) {
    return console.log(`Hi my name is ${name}, I'm ${age} years old and my
hobbies                          are ${hobbies});
}

info("Sjarel", 25, "gamen", "voetbal", "whatever");
```

OUTPUT?

Functions

REST PARAMETERS

```
function info(name, age, ...hobbies) {
    return console.log(`Hi my name is ${name}, I'm ${age} years old and my hobbies are
${hobbies}.`);
}

info("Sjarel", 25, "gamen", "voetbal", "whatever");
```

```
Hi, my name is Sjarel, I'm 25 years old and my hobbies are gamen voetbal
whatever .
```

Functions

REST PARAMETERS

```
let myArray = ["this", "is", "my", "array"];

function info(...data) {
    return console.log(data);
}

info(myArray);
```

Functions

ARROW FUNCTIONS

HOW IT WAS:

```
function (params) {
    return
}
```

HOW IT'S GOING:
(SINGLE LINE)

```
(params)  => value;
```

HOW IT'S GOING:
(MULTILINE)

```
(params)  => {
    return value;
}
```

Functions

ARROW FUNCTIONS

SINGLE LINE:

```
(params)  => value;
```

```
let log = (message) => console.log(message);
```

MULTILINE

```
(params)  => {
    return value;
}
```

```
let log = (name) => {
      let message = `Hi ${name}`;
      return message;
}
```

HOISTING

Hoisting

What is hoisting in variables...

```
var a = "First";
var b = "Second";
var c = "Third";

console.log(a + b + c);

//FirstSecondThird
```

```
var a = "First";
var b = "Second";
var c = "Third";

console.log(d);

var d;

// Which output do you expect?
```

What happens inside the browser?

Hoisting

What is hoisting in variables...

```
var a = "First";
var b = "Second";
var c = "Third";

console.log(a + b + c);

//FirstSecondThird
```

```
var a = "First";
var b = "Second";
var c = "Third";

console.log(d);

var d;

// undefined
```

Browser loads variable declarations first

FIX HOISTING IN VARIABLES

Use your variables inside functions! NOT in the root of your file.
Do not use them outside functions, when you will be using them inside of them.

Hoisting

What is hoisting in functions...

```
function example() {
    var a = 10;
    return a;
}

console.log(example());
```

```
console.log(example());

function example() {
    var a = 10;
    return a;
}

// 10
```

Browser loads function declarations first

FIX HOISTING IN FUNCTIONS

Use anonymous functions and attach them to a variable.

Hoisting

Fix it with anonymous functions

```
let example = function() {
    var a = 10;
    return a;
}

console.log(example());
```

```
console.log(example());

let example = function() {
    var a = 10;
    return a;
}

// TypeError: example is not a function
```

Hoisting no longer possible!

OBJECT LITERALS

_OBJECT LITERALS

BUILD-UP

```
function createTrainee(name, age) {
    return {
        name,
        age,
        job: {
            description: "consultant",
            companyName: "Axxes"
        },
        getJobDescription() {
             return `${name} is a ${job.description} at ${job.companyName}`;
        }
    }
}
```

Properties
Methods

_OBJECT LITERALS

USE OBJECT LITERAL

```
function createTrainee(name, age) {
    return {
        name,
        age,
        job: {
            description: "consultant",
            companyName: "Axxes"
        },
        getJobDescription() {
             return `${this.name} is a ${this.job.description} at
${this.job.companyName}`;
        }
    }
}
```

```
Let trainee = createTrainee("Jane Doe",
23);

Let traineeTwo = createTrainee("John
Doe", 26);


trainee.getJobDescription();

traineeTwo.getJobDescription();
```

OBJECT AND ARRAY DESTRUCTURING

Object and array destructuring

HOW DO WE DESTRUCTURE OBJECTS

```
let trainee = createTrainee("Jane Doe", 23);

let {name, age} = trainee;
console.log(`trainee's name is: ${name}`);

let {job} = trainee;
console.log(`trainee works at
${job.companyName}`);
```

HOW DO WE DESTRUCTURE ARRAYS

```
let myArray = [000, 111, 222];

let [,,valueName] = myArray
console.log(valueName);

// 222
```

DESTRUCTURING ARRAYS WITH REST PARAM

```
let [, ...otherValues] = myArray
Console.log(otherValues)

// [111, 222]
```

LOOPS

$$\begin{tikzcd}
 & \text{LOOPS} \arrow[dl] \arrow[dr] & \\
\text{FOR LOOP} & & \text{WHILE/DO... WHILE}
\end{tikzcd}$$

Loops

FOR LOOP

```
for (loop variable; loop condition; incrementExpression) {}

for (var i = 0; i < 5; i++) {}
```

LOOPS

FOR LOOP

<FOR LOOPS>

```
function loop() {
    for(var i = 0; i < 5; i++) {
        console.log(i);
    }
    console.log(i);
}
```

<OUTPUT>

```
> 0
> 1
> 2
> 3
> 4

> 5
```

LOOPS

FOR LOOP

<FOR LOOPS>

```
let myArray = ["One", "Two", "Three"];

function loop() {
    for(var i = 0; i < myArray.length; i++) {
        console.log(i);
    }
}
```

<OUTPUT>

```
>  0
>  1
>  2
```

Loops

WHILE LOOP

```
while (while condition) {}

while (var i = 0; i < 5; i++) {}
```

LOOPS

WHILE LOOP

<WHILE LOOPS>

```
let loading;
while (loading < 10) {
   console.log("Still looping!");
   loading++;
}
```

<OUTPUT>

```
>   Still looping!
>   Still looping!
>   Still looping!
>   Still looping!
>   ...
```

Loops

DO... WHILE LOOP

```
do{} while (while condition)

do{} while (var i = 0; )
```

LOOPS

WHILE LOOP

<WHILE LOOPS>

```
let loading;
do {
   console.log("Still looping!");
   loading++;
} while (loading < 10);
```

<OUTPUT>

```
>   Still looping!
>   Still looping!
>   Still looping!
>   Still looping!
>   ...
```

CLASSES

Classes

THE MAKING OF...

THE CONSTRUCTOR

- Make objects based on class

```
Class Car {
    constructor(name, color) {
        this.name = name;
        this.color = color;
    }
}
```

Classes

THE MAKING OF...

GETTERS AND SETTERS

- GET info out of our Object
- Adjust info inside our Object using SETTERS

```
Class Car {
    constructor(name, color) {
        this.name = name;
        this.color = color;
    }

    get name() {
        return this.name;
    }

    get color() {
        return this.color;
    }

    set name(name) {
         this.name = name;
    }

    set color(color) {
        this.color = color;
    }
}
```

Classes

THE MAKING OF...

METHODS

- Static methods
- "normal" methods

```
Class Car {
    constructor(name, color) {
        this.name = name;
        this.color = color;
    }

    ... (getters and setters)

    static makeCar(name, color) {
        return new Car(name, color);
    }

    getInfo() {
        return `${this.name} is a car with color:   ${this.color}`;
    }


}
```

Classes

USE YOUR CLASS

MAKING AN OBJECT BASED ON A CLASS

CONSTRUCTOR:

```
let BMW = new Car(“BMW”, “black”);
```

STATIC METHOD

```
let BMW = Car.makeCar(“BMW”, “black”);
```

```
Class Car {
    constructor(name, color) {
        this.name = name;
        this.color = color;
    }

    … (getters and setters)

    static makeCar(name, color) {
        return new Car(name, color);
    }

    getInfo() {
        return `${this.name} is a car with color:   ${this.color}`;
    }


}
```

Classes

THE MAKING OF...

ADJUSTING YOUR OBJECT THROUGH SETTERS

```
let BMW = new Car("BMW", "black");
BMW.name = "Mercedes";
```

OR YOU CALLING DATA THROUGH GETTERS

```
console.log(`The color of my car is ${BMW.color}`);
```

```
Class Car {
    constructor(name, color) {
        this.name = name;
        this.color = color.
    }

    get name() {
        return this.name;
    }

    get color() {
        return this.color;
    }

    set name(name) {
         this.name = name;
    }

    set color(color) {
        this.color = color;
    }
```

```
}
```

EXTENDING CLASSES

Classes

EXTENDING CLASSES

THE SUPER METHOD

- Calling constructor of EXTENDED Class

```
Class Hybrid extends Car {
    constructor(name, color, isHybrid) {
        super(name, color);
        this.isHybrid = isHybrid;
    }

    get isHybrid() {
        return this.isHybrid;
    }

   set isHybrid(isHybrid) {
       this.isHybrid = isHybrid;
    }

    getInfo() {
        return `${this.name} is a hybrid.`;
     }
}
```

Classes

EXTENDING CLASSES

CREATE AN OBJECT WITH OUR HYBRID CLASS

```
let Niro = new Hybrid("KIA", "Grey", true);
```

```
Class Hybrid extends Car {
    constructor(name, color, isHybrid) {
        super(name, color);
        this.isHybrid = isHybrid;
    }

    get isHybrid() {
        return this.isHybrid;
    }

   set isHybrid(isHybrid) {
       this.isHybrid = isHybrid;
    }

    getInfo() {
        return `${this.name} is a hybrid.`;
     }
}
```

Classes

DYNAMICALLY INHERITING WITH CLASSES

```
function getType(carType) {
    if(carType === “hybrid”) {
        return Hybrid
    } else {
        return Car
    }
}


Class KIA extends getType(“hybrid”) {
    constructor(name, color) {
        super(name, color);
    }
}


let kia = new KIA(“KIA”, “white”);
console.log(`${kia.getInfo()}`
```

MAPS

Maps

MAPS

- A collection of key/value pairs
- Getters and setters like Classes
- Iterable

```
var translations = new Map();
translations.set(“hello”, “hallo”);
translations.set(“bye”, “dag”);

console.log(`The translation of “bye” is
${translations.get(“bye”)}`);

// "bye"
```

```
translations.forEach((key, value) => {
    console.log(`${key}: ${value}`);
}
```

PROMISES

Promises

BASICS

PROMISES

- A function parameter with two variables
- RESOLVE/REJECT

```
let promise = new Promise((resolve, reject) => {
    let sum = 1+1;

    if(sum == 2) {
        resolve(“Yaay, we succeeded.”)
    } else {
        reject(“FAIL!”)
    }
});
```

Promises

BASICS

PROMISES

- A function parameter with two variables
- RESOLVE/REJECT
- RESOLVE = ACTION -> Then method
- REJECT = ACTION -> Catch method

```
let promise = new Promise((resolve, reject) => {
    let sum = 1+1;

    if(sum == 2) {
        resolve("Yaay, we succeeded.")
    } else {
        reject("FAIL!")
    }
});

promise.then((res) => {
    console.log('We get here when the promise is
resolved: + ${res}');
}).catch((res) => {
    console.log('We get here when the promise is
rejected: + ${res}');
});
```

Promises

BASICS

PROMISES

- Promise.all()

```
let promise1 = new Promise((resolve, reject) => {
    resolve("First promise resolved);
);

let promise2 = new Promise((resolve, reject) => {
    resolve("Second promise resolved);
);

let promise3 = new Promise((resolve, reject) => {
    resolve("Third promise resolved);
);

promise.all([
   promise1,
   promise2,
   promise3
]).then((messages)  => {
   console.log(messages)
});
```

Promises

BASICS

PROMISES

- Promise.race()

```
let promise1 = new Promise((resolve, reject) => {
    resolve("First promise resolved);
);

let promise2 = new Promise((resolve, reject) => {
    resolve("Second promise resolved);
);

let promise3 = new Promise((resolve, reject) => {
    resolve("Third promise resolved);
);

promise.race([
   promise1,
   promise2,
   promise3
]).then((messages) => {
   console.log(messages)
});
```

ASYNC/AWAIT

Await/async

BASICS

ASYNC/AWAIT

- Waits for promises to be resolved
- Code continues in background
- Code inside async waits for promise!

```
fetch("https://pokeapi.co/api/v2/pokemon/mew")
    .then(res => {
        console.log(`This pokémon is ${res.name}`);
    })
    .catch(err => {
        console.log(`Could not fetch pokémon ${err}`);
    })
```

WITH ASYNC/AWAIT

```
async function getPokemon() {
  await fetch("https://pokeapi.co/api/v2/pokemon/mew");
  console.log(`This pokémon is ${res.name}`);
}
```

Await/async

BASICS

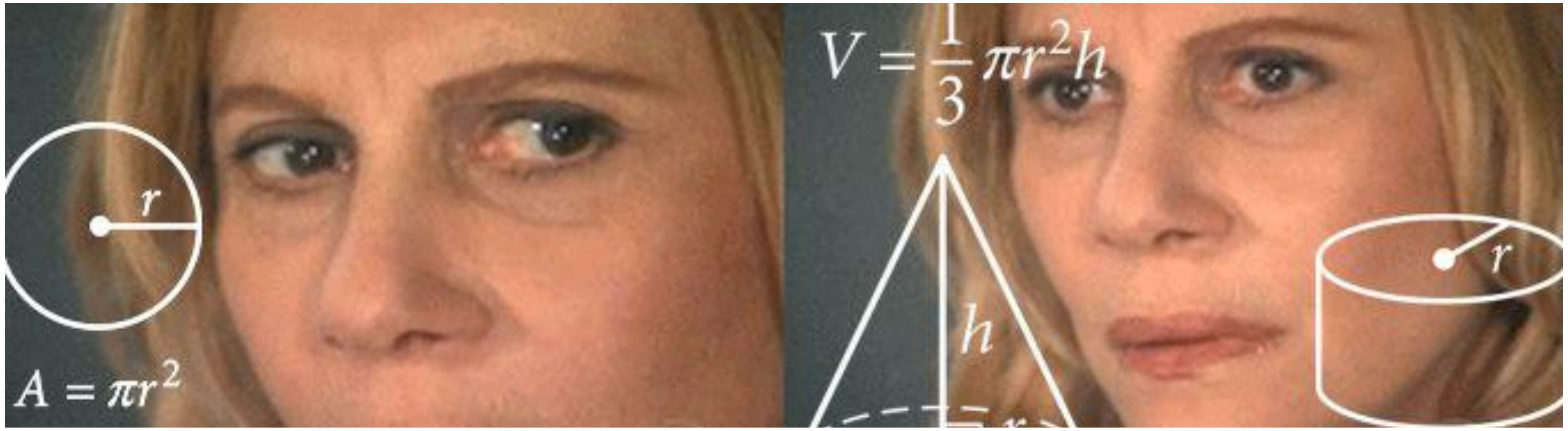
ASYNC/AWAIT ERROR HANDLING

- Instead of .catch => try, catch

```
fetch("https://pokeapi.co/api/v2/pokemon/mew")
   .then(res => {
      console.log(`This pokémon is ${res.name}`);
   })
   .catch(err => {
      console.log(`Could not fetch pokémon ${err}`);
   })
```

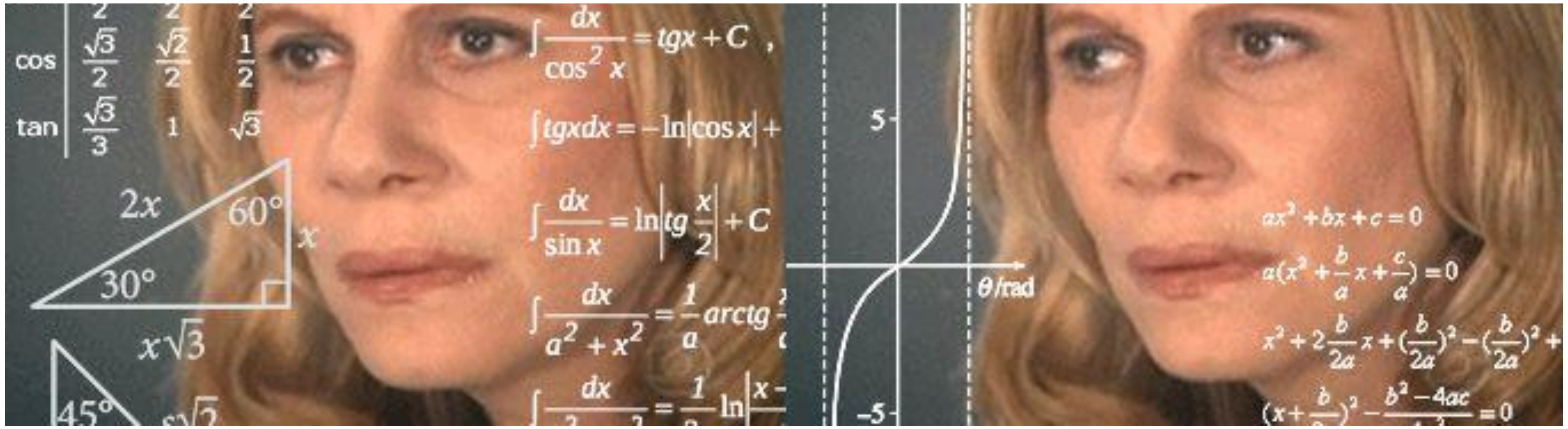
WITH ASYNC/AWAIT

```
async function getPokemon() {
  try {
     await fetch("https://pokeapi.co/api/v2/...");
     console.log(`This pokémon is ${res.name}`);
  } catch (err) {
     console.log(`Could not fetch pokémon ${err}`);
  }
}
```



$A = \pi r^2$
r
$V = \frac{1}{3}\pi r^2 h$
h
r

EXERCISES!



cos
tan
2x
60°
x
30°
$x\sqrt{3}$
45°
$\int \frac{dx}{\cos^2 x} = tgx + C$
$\int tgxdx = -\ln|\cos x| +$
$\int \frac{dx}{\sin x} = \ln\left|tg\frac{x}{2}\right| + C$
5
-5
θ/rad
$ax^2 + bx + c = 0$
$a(x^2 + \frac{b}{a}x + \frac{c}{a}) = 0$

https://github.com/CharlieHuygen/axxes-traineeship-javascript-exercises

```
cd exercises

npm install
```

Exercises

JEST

RUN THE TEST

```
npm test hoisting
```

TEST FILES

hoisting.spec.js

FIX THE CODE INSIDE .JS FILES.

FOLLOW THE TODOS

Exercises

JEST

```
expect()
.toBe()
.toEqual()
```

```
expect(value).toBe(5);
expect(object).toEqual({name: 'Charlotte Huygen'});
```

TS

TYPESCRIPT

_BUT FIRST

1. INSTALL NODEJS

https://nodejs.org/en/

2. INSTALL NPM

```
npm install  -g
```

3. INSTALL TYPESCRIPT

```
npm install  -g typescript
```

WHICH SUBJECTS ARE WE GOING TO TOUCH?

- > Data types
- > Conversion
- > Interfaces
- > Arrays
- > Looping
- > Functions
- > Classes
- > Variables in Classes
- > Generic Functions
- > Generic Classes
- > Destructuring

JUST DO IT!

DATA TYPES

_DATA TYPES

BUILD-UP

```
var name: datatype = value;

var name: string = "Charlotte";
```

_DATA TYPES

ECMASCRIPT

```
var x = 5;

var x = "hello";

var x = true;

var x = [1, 2, 3]

var x = {};

var x = "anything really";

var x = null;
```

TYPESCRIPT

?

_DATA TYPES

ECMASCRIPT

```
var x = 5;
var x = "hello";
var x = true;
var x = [1, 2, 3]
var x = {};
var x = "anything really";
var x = null;
```

TYPESCRIPT

```
var x: number = 5;
var x: string = "hello";
var x: boolean = true;
var x: number[] = [1, 2, 3]
var x: object = {};
var x: any = "anything really";
var x: null = null;
```

CONVERSION

_CONVERSION

FROM ONE VARIABLE TYPE TO THE OTHER

String <-> Number

Number <-> String

Conversion

CONVERT STRING TO NUMBER

METHOD:

- .parseInt()

```
let number = "5";

let number: number = parseInt("5");
```

Conversion

CONVERT NUMBER TO STRING

METHOD:

- .parseInt() // string to number
- .toString() // number to string

```
let number = "5";

let number: number = parseInt("5");


let text: string = number.toString();
```

INTERFACES

Interfaces

WHAT ARE THEY?

- Used for complex data types
- = a model for something
- Consist of typed properties

```
Interface Address {
    street: string;
    houseNumber: number;
    areaCode: number;
    city: string;
}


Let myAddress: Address = {
    street: “Meir”,
    houseNumber: 1,
    areaCode: 2000,
    city: “Antwerp”
}
```

Interfaces

INTERFACES AS A MODEL FOR CLASSES

- Used for complex data types
- = a model for something
- Consist of typed properties

```
Interface Valuta {
    getValue(): any;
}

Class CryptoCurrency implements Valuta {
    constructor(name: string, value: number) {}


    getValue(): void {
        console.log('$(this.name) has a current value of
${this.number}');
    }
}

Const bitcoin = new CryptoCurrency("Bitcoin", 30000);
Const ethereal = new CrypoCurrency("Ethereal",
12000);

bitcoin.getValue();
ethereal.getValue();
```

ARRAYS

_ARRAYS

BUILD-UP

```
var name: datatype = value;
```

HOW DO WE MAKE A TYPED ARRAY???

_ARRAYS

BUILD-UP

```
var name: datatype = value;

let students: string[] = ["student1", "student2", "student3"]
```

_ARRAYS

BUILD-UP

```
var name: datatype = value;

let students: string[] = ["student1", "student2", "student3"]
let numbersArray: number[] = [123, 456, 789]
```

ARRAYS

INTERFACE AS A MODEL FOR AN ARRAY

- Account for all the required vars
- Multiple values in array? Add them in like an object.

```
Interface Address {
    street: string;
    houseNumber: number;
    areaCode: number;
    area: string;
}


Let myAddress: Address[] = [
{
    street: “Meir”,
    houseNumber: 1,
    areaCode: 2000,
    area: “Antwerp”
}, {
    street: “Amerikalei”,
    houseNumber: 2,
    areaCode: 2000,
    area:“Antwerp”
}]
```

LOOPING

_ LOOPS

1. FOR IN

2. FOR OF

_ LOOPS

LOOPING OVER AN ARRAY WITH THE FOR IN LOOP

```
var numberArray = [1,2,3,4,5];

for(var number in numberArray) {
    console.log(number);
}

// WHAT WILL BE LOGGED?
```

_ LOOPS

LOOPING OVER AN ARRAY WITH THE FOR IN LOOP

- FOR IN will log the index of the elements

```
let numberArray = [1,2,3,4,5];

for(let number in numberArray) {
    console.log(number);
}

// 0, 1, 2, 3, 4
```

_ LOOPS

LOOPING OVER AN ARRAY WITH THE FOR OF LOOP

- FOR OF will log the actual elements

```
let numberArray = [1,2,3,4,5];

for(let number of numberArray) {
    console.log(number);
}

// 1, 2, 3, 4, 5
```

FUNCTIONS

_ FUNCTIONS

RETURN TYPES

- You can type parameters
- Functions can have return types

ECMASCRIPT/JAVASCRIPT

```
let getSum = (number1, number2) => {
    return console.log(number1 + number2);
}

Let sum = getSum(5, 2);
console.log(sum);
```

TYPESCRIPT

???

_ FUNCTIONS

RETURN TYPES

- You can type parameters
- Functions can have return types

ECMASCRIPT/JAVASCRIPT

```
let getSum = (number1, number2) => {
    return console.log(number1 + number2);
}

Let sum = getSum(5, 2);
console.log(sum);
```

TYPESCRIPT

```
let getSum =
(number1:number, number2:number):number  => {
    return console.log(number1 + number2);
}

Let sum = getSum(5, 2);
console.log(sum);
```

_ FUNCTIONS

DEFAULT VALUES AND OPTIONALS

- DEFAULT VALUE?

  Typing not needed!
- OPTIONAL?

  Use a question mark.
- IN CASE OF OPTIONALS:

  Don't forget the question mark

```
let getSum = (number1: number, number2 = 2) => {
    return console.log(number1 + number2);
}


let getSum = (number1: number, number2: number, number3?: number) => {

    if(number3 !== undefined) {
        return console.log(number1 + number2 + number3);
    } else {
        return console.log(number1 + number2);
    }
}
```

CLASSES

_ CLASSES

DIFFERENCE JS CLASSES VS TYPESCRIPT CLASSES

```
Class Game {

    constructor(type: string, name: string, price: number) {
        this.type = type;
        this.name = name;
        this.price = number;
    }

}
```

_ CLASSES

PRIVATE - PUBLIC - READONLY VARIABLES IN CLASSES

- Private, public, readonly = Access modifiers
- In which way are your properties available?

```
Class Game {

    private type: string;
    public name: string;
    readonly price:  number;

}
```

1. THE PRIVATE ACCESS MODIFIER

Property only available within the class.

SO: ~~marioKart.type~~

BUT: marioKart.getInfo();

_ CLASSES

PRIVATE - PUBLIC - READONLY VARIABLES IN CLASSES

```
Class Game {

    private type: string;
    public name: string;
    readonly price:  number;

}
```

2. THE PUBLIC ACCESS MODIFIER

Property available within and outside the class. This is the DEFAULT of every property!

You can use the public keyword, but it isn't necessary

_ CLASSES

PRIVATE - PUBLIC - READONLY
VARIABLES IN CLASSES

```
Class Game {

    private type: string;
    public name: string;
    readonly price:  number;

}
```

2. THE READONLY ACCESS MODIFIER

You can READ the property inside and outside the class and ONLY that.

NO SETTING, JUST GETTING

GENERIC FUNCTIONS

_ GENERIC FUNCTIONS

WHAT ARE THEY?

= Reusable blocks of code

- Can be used with different types

```
function getType<T> (val: T): string {
    return typeof(val);
}

let string = "A string";
let number = 7;


console.log(getType(string));
console.log(getType(number));
```

GENERIC CLASSES

GENERIC CLASSES

WHAT ARE THEY?

= Reusable blocks of code

- Can be used with different types

```
class GenericCalculation<T>  {
    add: (val1: T, val2: T) => T
}


let number = new GenericCalculation<number>();
number.add = (x, y) => x +y;


Console.log(number.add(1, 2));


let string = new GenericCalculation<string>();
string.add = (x, y) => x +y;


console.log(string.add(“1”,”2”));
```

DESTRUCTURING

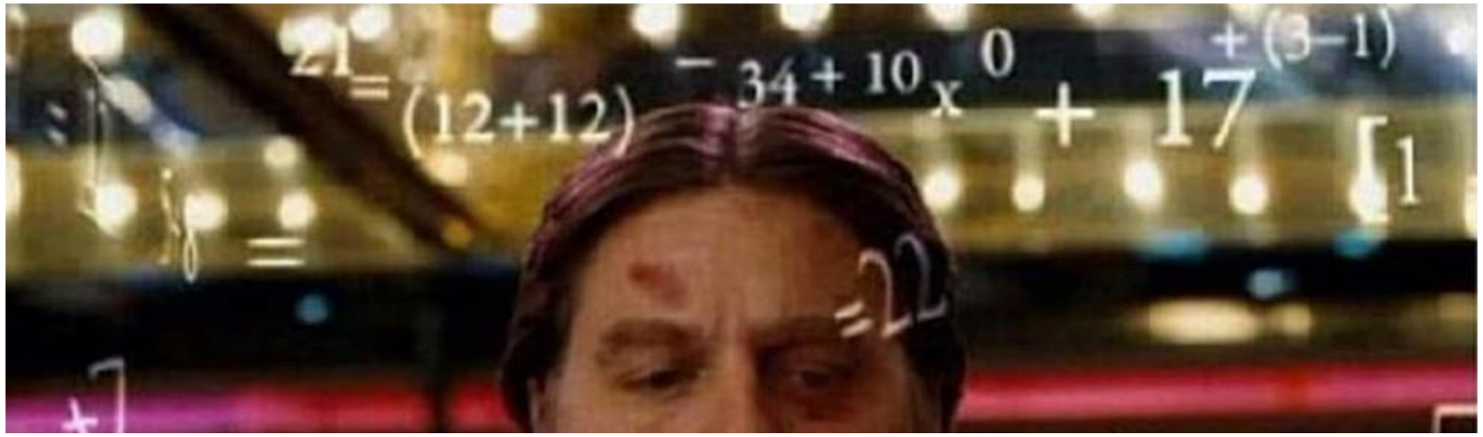
DESTRUCTURING

WHAT IS IT?

We can rearrange how variables or arrays hold their values.

```
let values = {x: 1, y: 2, z: 3};

let {x, y, z} = values;
 console.log( x + y + z);
// 1, 2, 3
```



EXERCISES!

ANY QUESTIONS??