27 July 2021

- alert (“Hello world!”) – Enter

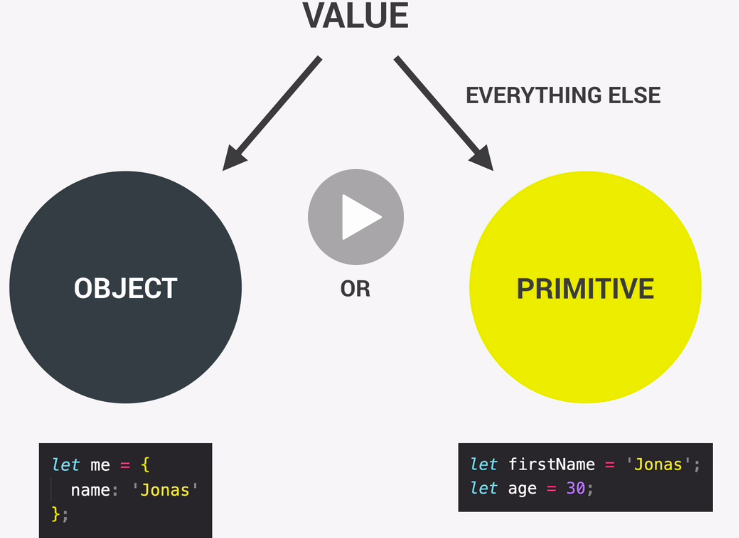
- ‘if’ commands

- using JS as a calculator

- What is Javascript?

* High level, object oriented, multi paradigm programming language
  + Programming language – the language that computers speak in
  + High level – complexities are hidden
  + Object oriented – based on objects for storing data
  + Multi paradigm – different styles (structuring code) of programming is applicable

- Why use Javascript?

* The Three Pillars – html, CSS, javascript
  + Html – content – text, photos, links etc
  + CSS – aesthetics / presentation
  + Javascript – programming language to build web applications
* Dynamic effects – application through Twitter example
* Libraries / tools such as React/Angular/Vue frameworks – makes modern, large scale dev. easier, but is based fundamentally in Javascript
  + Libraries might disappear, but Javascript would remain as the foundation
* Outside of browser application / web applications on web servers – nodeJS
* Native mobile / desktop applications – ReactNative, Ionic, electron
* Javascript releases - ECMAScript
  + ES5
  + ‘Modern Javascript’
    - ES6 - 2015
    - ES11 – 2020
  + Jonas msg – **learn modern Javascript from the beginning**
* Console.log(40+8)
  + Explicit instruction for JS to know to display the mathematical calculation into the browser console
* ‘Values’ – a piece of data
  + Declaring variables –
  + *let* firstName = 'Jonas';
  + Conventions
    - Camelcase
      * firstName
      * firstNamePerson
      * first\_name\_person
    - Must start with a letter
    - Space around operators (= + - \* / ,)
    - Legal symbols - Numbers, letters, underscores or $
    - Reserved Javascript keywords – ‘new’, ‘function’, ‘name’
    - Do not start with uppercase letter
    - The variable name should make sense and give a hint to what data is being contained
* A Value can either be: a) an object or b) primitive data
  + 
* The 7 primitive data types
  + Number (floating point numbers) – used for decimals, integers
    - Let age = 23;
  + Strings (sequence of characters) – used for text, *must be enclosed in quotation marks* otherwise they get confused for variables
    - Let firstName = ‘Jonas’;
  + Boolean (logical type that can only be true or false) – used for taking decisions
    - Let fullAge = true;
  + Undefined (empty value placeholder)
    - Let children;
  + Null (also empty value placeholder)
  + Symbol (unique and cannot be changed)
  + BigInt (larger integers than the Number type can hold)
* Dynamic typing – to use a data type, you do not have to declare explicitly what the type of each data point needs to be
  + It is the value that has the type, not the variable
  + Variable X can be a string or later converted into a number
* Commenting in JS - // or control + /, /\* and \*/
* Figure out what the data type is – typeof
* Changing the value of an already existing variable
  + Let javascriptIsFun = true;
  + javascriptIsFun = ‘YES!’
* Declaring variables that store ‘undefined’ values and displaying to console
  + Undefined can be both the *type* and *value*
* Console.log(typeof null);
  + Javascript weirdly declares null as an ‘object’ type
  + Legacy bug
* Let, const and var
  + Let – variable declaration
    - Variable reassignment or muted – changing the value after declaration
* *let* age = 30;
* age = 31;
* console.log(age);
  + const – value that cannot be changed
    - Cannot declare empty/undefined/null const variable
    - When to use const? Most of the time as common practice, use let if variable value needs to change later down the track
  + Var – function scope vs let (block scoped), generally avoid using var
* Is let / const really required?
* lastName = 'programmer';
* console.log(lastName);
  + Directly assigning a value is possible but not good practice
* Basic operators – transformation / combination of values
  + Mathematical - +, -, \*, /, \*\* (power)
  + Using console.log (x, y, z …) to display multiple variables
  + Concatenation of strings - Combining strings with +
* *const* firstName = 'Jonas';
* *const* lastName = 'Schmedtmann';
* console.log(firstName + lastName);
  + Assignment operators - =, +=, \*=, /=, ++, --
* *let* x = 10 + 5;
* x += 5; *// x = x + 5*
* console.log(x);
  + Comparion operators - >, <, >=, <=
  + Bimdas / operator precedence
* console.log (now - 1991 >  now - 2018);

28/7/21

* Operator precedence - <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence>
  + Generally, arithmetic operators take precedence over comparison operators
  + Assignments operate from right to left
* *let* x,y;
* x = y = 25 - 10 - 5;
  + Highest priority – grouping ()
* Declaring two variables at the same time: let x,y;
* Strings and template literals
  + Template literals – makes concatenating strings easier (putting in spaces etc)
  + Establish by using `` and ${} – introduced in ES6
* *const* jonas = "I'm " + firstName + ' ,a ' +  (year - birthYear) + ' years old ' + job + '!';

*const* jonasNew = `I'm ${firstName}, a ${year - birthYear} years old ${job}!`;

* Multiline strings
* console.log('String with \n\
* multiple \n\
* lines');
* console.log(`String with
* multiple
* lines`);
* Taking decisions – if/else statements
  + If / else control structure - Not necessarily linear execution of code
* if (age >= 18){
* console.log(`Sarah can start driving license`);
* } else{
* *const* yearsLeft = 18 - age;
* console.log(`Sarah is too young. Wait another ${yearsLeft} years.`);
* }
  + Note that variables declared within code blocks {} are not accessible to elements outside of the block
    - Block-exclusion is necessary for if/else to work in the first place
* Type conversion and coercion – converting between types manually / automatically
  + String -> number
* *const* inputYear = '1991';
* console.log(Number(inputYear));
  + Nan – not a number
  + Number -> string
* console.log(String(23));
  + Coercion
* console.log(1995 - '2001');
  + + operator – converts numbers to strings
* console.log('I am ' + 23 + ' years old');
  + – operator – converts strings to numbers
* console.log ('23' - '10' - 3);
  + \* or / or comparison operators – converts strings to numbers
* console.log('23' \* '2');
  + The Test
* *let* n = '1' + 1;
* n = n - 1;
* console.log(n);
* *//my guess is 10*

29/7/21

Truthy and falsy

* Falsy values return false when converted into a Boolean - 0, ‘’, undefined, null, NaN
* Truthy values return true when converted into a Boolean
* Most of the time these cases are handled by value coercion by JS
* *const* money = 0;
* if(money) {
* console.log("Don't spend it all.");
* } else{
* console.log("You should get a job");
* }
* If / else statements – by design accepts Boolean
* How to test if a variable exists?

Equality operators (loose and strict) - == or ===

* = - assignment operator
* == - loose equality operator, enables type coercion
  + ‘18’ == 18 -> true
* === - strict equality operator, exactly equal to
  + ‘18’ === 18 -> false
* Prompt command
* Good practice – use strict operator

Logical operators – AND &&, OR ||, NOT !, if/else

2 August 2021

The Switch Statement

* *const* day = 'Friday';
* switch(day){
* case 'Monday':
* console.log('Plan my course structure');
* console.log('Go to coding meetup');
* break;
* case 'Tuesday':
* console.log('Prepare theory videos');
* break;
* case 'Wednesday':
* case 'Thursday':
* console.log('Write code examples.');
* break;
* case 'Friday':
* console.log('Record videos');
* break;
* case 'Saturday':
* case 'Sunday':
* console.log('Enjoy the weekend');
* break;
* default:
* console.log('Not a valid day!');
* break;
* }
* Important to remember the break; otherwise the code keeps continuing down the cascade
* If / else equivalent
* if (day === 'Monday'){
* console.log('Plan my course structure');
* console.log('Go to coding meetup');
* } else if(day === 'Tuesday'){
* console.log('Prepare theory videos');
* } else if((day === 'Wednesday') || (day ==='Thursday')){
* console.log('Write code examples.');
* } else if(day === 'Friday'){
* console.log('Record videos');
* } else if((day ==='Saturday') || (day ==='Sunday')){
* console.log('Enjoy the weekend');
* } else{
* console.log('Not a valid day!');
* }

Statements and expressions

* Expression – code that produces a value e.g ‘2 + 3’
* Statements – code that does not e.g ‘if/else structures’

The conditional (ternary) operator

* Conditional operators – if/else, switch, ternary operators
* *const* age = 15;
* age >= 18 ? console.log('I like to drink wine 🍷.'):
* console.log('I like to drink water.');
* An operator **is an expression**
* Ternary operator in comparison to if/else
* *const* drink = age >= 18 ? 'wine 🍷' : 'I like to drink water';
* console.log(drink);
* *let* drink2;
* if(age>=18){
* drink2 = 'wine';
* } else{
* drink2 = 'I like to drink water';
* }
* console.log(drink2);
* Ternary operators are expressions and can be used within a template literal (where if/else structure cannot go – statement within a statement, no good)
* console.log(`I like to drink ${age >= 18 ? 'wine 🍷' : 'I like to drink water'}`);
* Not necessarily replacements of if/else

Coding challenge #4:

*const* bill = 430;

*const* tip = (bill >= 50 && bill <=300) ? (0.15 \* bill) : (0.2 \* bill);

*const* total = bill + tip;

console.log(`The bill was ${bill}, the tip was ${tip} and the total value ${total}`);

Javascript releases

* Created in 1995
  + The desire for more interactive websites
  + Netscape and Brandon Eich – created Javascript in **10 days**
* 1996 – Mocha changed from Livescript and then to Javascript
  + Java has nothing to do with Javascript
  + Microsoft copied and created Jscript for internet explorer
* 1997 – Creation of ECMAscript out of a need to standardise the languages (ECMAscript is the standard, Javascript is the language in practice)
* 2009 – ES5 released with new features
* 2015 – ES6 / ES2015– biggest update ever
* ‘Don’t break the world’ principle - Javascript has backwards compatability – latest JS compatible with ES1
  + Old features never removed, new versions are just incremental upgrades
* Good practice
  + During development - Always use the latest Chrome
  + During production – conversion back to ES5 to ensure web compatibility for all users, transpile and polyfill (via Babel)
* Do not forget tradition
  + You will better understand how JS works
  + You will likely encounter old code, so you’d better be equipped to understand them

**Fundamentals II**

* Strict mode
  + First line in JS file “ use ‘strict’; “
  + Forbids certain actions
  + Makes transparent ‘silent failures’ of JS
* Functions
  + Reusable chunks of code, consisting of smaller sections of code
  + Calling / writing / invoking
    - Logger();
* *function* fruitProcessor(*apples*, *oranges*){
* console.log(apples, oranges);
* *const* juice = `Juice with ${apples} apples and ${oranges} oranges`;
* return juice;
* }
  + Apples/oranges are called the arguments of the function
  + Note how juice cannot exist outside of the function (statement)
* Function declaration vs expressions
  + Parameter is the placeholder for arguments – ()
  + Declaration – can call function before its declaration
  + Function expression – essentially a function value stored within a variable
* *//function declaration*
* *const* age1 = calcAge1(1995);
* *function* calcAge1(*birthYear*){
* return 2037 - birthYear;
* }
* *//function expression*
* *const* calcAge2 = *function* (*birthYear*){
* return 2037 - birthYear;
* }
* *const* age2 = calcAge2(1991);
* console.log(age1, age2);

Arrow functions (added in ES6)

* Special form of function expression via =>
* *const* calcAge3 = *birthYear* => 2037 - birthYear;
* console.log(calcAge3(1995));
* Another example with 2 arguments:
* *const* yearsUntilRetirement = (*birthYear*, *firstName*) => {
* *const* age = 2037 - birthYear;
* *const* retirement = 65 - age;
* *// return retirement;*
* return `${firstName} retires in ${retirement}`;
* }
* console.log(yearsUntilRetirement(1995, 'John'));

First data structure – Arrays

* *const* friend1 = 'Michael';
* *const* friend2 = 'Steven';
* *const* friend3 = 'Peter';
* *const* friend = ['Michael', 'Steven', 'Peter'];
* *const* years = new Array(1991, 1984, 2008, 2020);
* Array length
* console.log(friends[0]);
* console.log(friends.length);
* console.log(friends[(friends.length-1)]);
* Mutating array values
  + Even with const declarations
  + Only primitive values are immutable, which arrays are not
  + But the following is illegal
* *friends = ['Bob', 'Alice', 'Jay'];*

Basic array operations

* Push function – adds elements to end of array.
  + Read as – ‘push function added onto the friends array’
  + Actually returns the new length of the new array
* friends.push('Jay');
* Unshift function – adds elements to the front of array
* Pop function / shift function – removes last element on the array / removes first element on the array
* Position of value within array
* console.log(friends.indexOf('Steven'));
* Check if value is within array – returns Boolean
  + Does not utilise type coercion, strict equality
* friends.push(23);
* console.log(friends.indexOf('Steven'));
* console.log(friends.includes('23'));

Objects

* Individual values within arrays are only identifiable by their order number as opposed to ‘categories’ or explicit headings, which is where Objects come in
* *const* jonas = {
* firstName: 'Jonas',
* lastName: 'Schmedtmann',
* age: 2037 - 1991,
* job: 'teacher',
* friends: ['Michael', 'Peter', 'Steven']
* };
* The Object Jonas has 5 properties
  + With individual values
* Objects preferable for unordered data, arrays for ordered data

Dot vs Bracket notation – grabbing properties/values from objects

* console.log(jonas.lastName);
* console.log(jonas['lastName']);
* Bracket notation - Leveraging expressions.
* *const* nameKey = 'Name';
* console.log(jonas['first' + nameKey]);
* console.log(jonas['last' + nameKey]);
* [‘first’ + nameKey] -> firstName
* Dot notation – have to use the exact property name
* Adding onto objects
* jonas.location = 'Portugal';
* jonas['twitter'] = '@jonasschmedtman';

3 August 2021

Functions calling other functions

* Don’t Repeat Yourself (DRY) principle
* *function* cutFruitPieces(*fruit*){
* return fruit \* 4;
* }
* *function* fruitProcessor(*apples*, *oranges*){
* *const* applePieces = cutFruitPieces(apples);
* *const* orangePieces = cutFruitPieces(oranges);
* *const* juice = `Juice with ${applePieces} apple pieces and ${orangePieces} orange pieces`;
* return juice;
* }

Review Arrow functions

Coding Challenge #1

4 August 2021

Object Methods

* Methods – actions that can be performed on objects
* A function can be stored as a value within an Object property
* *const* jonas = {
* firstName: 'Jonas',
* lastName: 'Schmedtmann',
* age: 2037 - 1991,
* job: 'teacher',
* friends: ['Michael', 'Peter', 'Steven'],
* hasDriversLicense: true,
* calcAge: *function*(*birthYear*){
* return 2037 - birthYear;
* }
* };
* This – keyword to call the encapsulating function
* *const* jonas = {
* firstName: 'Jonas',
* lastName: 'Schmedtmann',
* birthYear: 1991,
* job: 'teacher',
* friends: ['Michael', 'Peter', 'Steven'],
* hasDriversLicense: true,
* *// calcAge: function(birthYear){*
* *//     return 2037 - birthYear;*
* *// }*
* calcAge: *function*(){
* return 2037 - *this*.birthYear;
* }
* };

Iteration and loops

* Template string - ``
* for(*let* rep = 1; rep <= 10; rep ++){
* console.log(`Lifting weights repetition ${rep}`);
* }
* For structure (initial value, limit, increment)
  + Counter, the condition and the counter update
* *const* jonas = [
* 'Jonas',
* 'Schmedtman',
* 2037 - 1991,
* 'teacher',
* ['Michael', 'Peter', 'Steven']
* ];
* *const* types = [];
* for(*let* i = 0; i < jonas.length; i++){
* console.log(jonas[i], typeof jonas[i])
* types[i] = typeof jonas[i];
* }

Loops – continue and breaks

* Continue – log only strings from array
* for(*let* i = 0; i < jonas.length; i++){
* if(typeof jonas[i] !== 'string') continue;
* console.log(jonas[i], typeof jonas[i])
* }
* Force loop to stop
* *const jonas = [*
* *'Jonas',*
* *'Schmedtman',*
* *2037 - 1991,*
* *'teacher',*
* *['Michael', 'Peter', 'Steven']*
* *];*
* *for(let i = 0; i < jonas.length; i++){*
* *if(typeof jonas[i] === 'number') break;*
* *console.log(jonas[i], typeof jonas[i])*
* *}*

While loop

* For use when the number of iterations Is unknown
  + E.g dice roll scenario

5 August 2021

*//Coding challenge #4*

*const* bills = [22, 295, 176, 440, 37, 205, 10, 1100, 86, 52];

*const* tips = [];

*const* totals = [];

*function* calcTip(*bill*){

*let* tip;

    if(bill >= 50 && bill <=300){

        tip = 0.15 \* bill;

    } else {

        tip = 0.2 \* bill;

    }

    return tip;

}

for (*let* i=0; i<=9; i++){

    tips.push(calcTip(bills[i]));

    totals.push(bills[i] + tips[i]);

}

console.log(bills, tips, totals);

*function* calcAverage(*arr*){

*let* sum = 0;

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

        sum = arr[i] + sum;

    }

    return sum / arr.length;

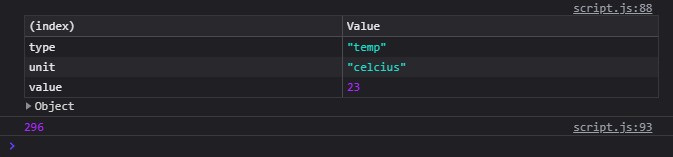
}

console.log(calcAverage([2, 3, 6]));

console.log(calcAverage(totals));

11 August 2021

Dev environment

* Setting up opinionated editor – Prettier
* Custom snippets – TODO highlight
* NodeJS live browser
* Learn how to code
  + How to fail?
    - No clear goals in the beginning
      * What am I trying to get to?
        + Feedforward learning community prototype
        + Upskilling
    - Blatant copy and paste code
    - No reinforcing learnings – through side projects, hobbies etc
    - Quickly getting frustrated / losing motivation
    - Learning in isolation
      * #100DaysofCode
      * #CodeNewbie
      * #webdev
* How to think developer – problem solving
  + “In an array of GPS coordinates, find the two closest points”
  + Common mistakes
    - Jumping into the problem without thinking first
    - Implements solution in unstructured way
    - Gets frustrated when things don’t work
    - Too proud to ask for help
  + The Four Steps
    - Make sure you understand the whole problem 100%
    - Divide and conquer – break up the big problem into smaller quests
    - Don’t be afraid to research as much as possible
      * Do not hit your head against the wall more than necessary
    - Write pseudocode before the actual code
      * Writing code in normal language to check the logic first before writing it in actual JS
* Google, StackOverflow and MDN
* Debugging
  + Identify
  + Find
  + Fix
  + Prevent
* Debugging with the console and breakpoints
  + Console.warn()
  + Console.error()
  + Console.table(*Object name*)
  + 
  + Debugger;
  + Placeholders – hold points in the execution of the code
* *//Coding challenge*
* *const* printForecast = *function*(*arr*){
* *let* forecast = "";
* for(*let* i = 0; i <= arr.length-1; i++){
* forecast = forecast + ` ${arr[i]}ºC in ${i+1} days...`;
* }
* return(forecast);
* }
* *const* data1 = [17, 21, 23];
* *const* data2 = [12, 5, -5, 0, 4];
* console.log(printForecast(data1));
* console.log(printForecast(data2));

Dom and Events Fundamentals

* Project #1 – Guess My Number
  + DOM – interfacing with html/CSS elements
    - What is DOM and DOM manipulation?
    - Document Object Model
      * The DOM tree structure
      * Document.QuerySelector()
      * The DOM !== Javascript
        + DOM are part of web APIs which are accessible through Javascript – libraries written in JS
* Selecting and manipulating elements
* document.querySelector('.message').textContent = 'Correct number!';
  + Note .message is a CSS ‘class selector’ and the overall JS structure to this line
    - Document. – dot notation as property accessor
    - querySelector – function
    - textContent – function
* Handling click events
* document.querySelector('.check').addEventListener('click', *function* (){
* console.log(document.querySelector('.guess').value);
* })
* Selecting CSS elements via Javascript
* document.querySelector('body').style.backgroundColor = '#60b307';
  + The value to be changed in the CSS code has to be contained in a string
  + These adjustments create in-line styles in the html file itself and leave the CSS untouched
  + Note the camel case notation of the CSS terminology ‘background-color’
* The DRY Principle – refractoring
* *const* displayMessage = *function*(*message*){
* document.querySelector('.message').textContent = message;
* }

*// document.querySelector('.message').textContent = guess > secretNumber ? '📈 Too high!' : '📉 Too low!';*

            displayMessage(guess > secretNumber ? '📈 Too high!' : '📉 Too low!');

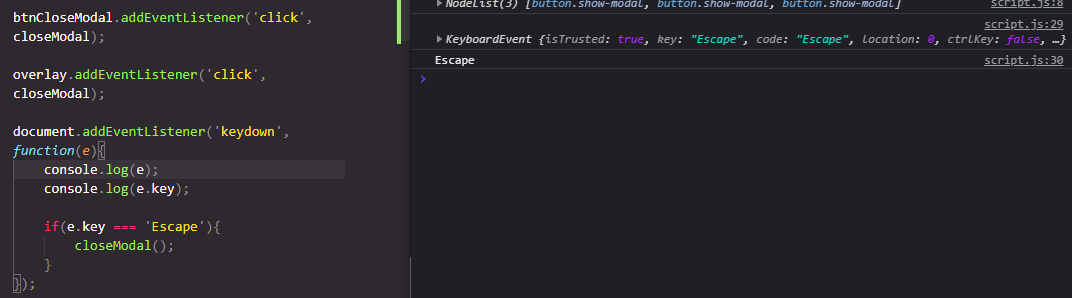
* Refractoring via functioning out the document.querySelector and the use of ternary operators

The Modal Window

Working with classes

* Limitations of QuerySelector
  + Document.querySelectorAll
* Node list – unpacking the btnsOpenModel ‘array’
* *const* btnsOpenModal = document.querySelectorAll('.show-modal');
* for(*let* i = 0; i < btnsOpenModal.length; i++){
* btnsOpenModal[i].addEventListener('click', *function* (){
* console.log('Button clicked');
* });
* }
* 

Keyboard events



13 August 2021

Project 3 – Pig Game

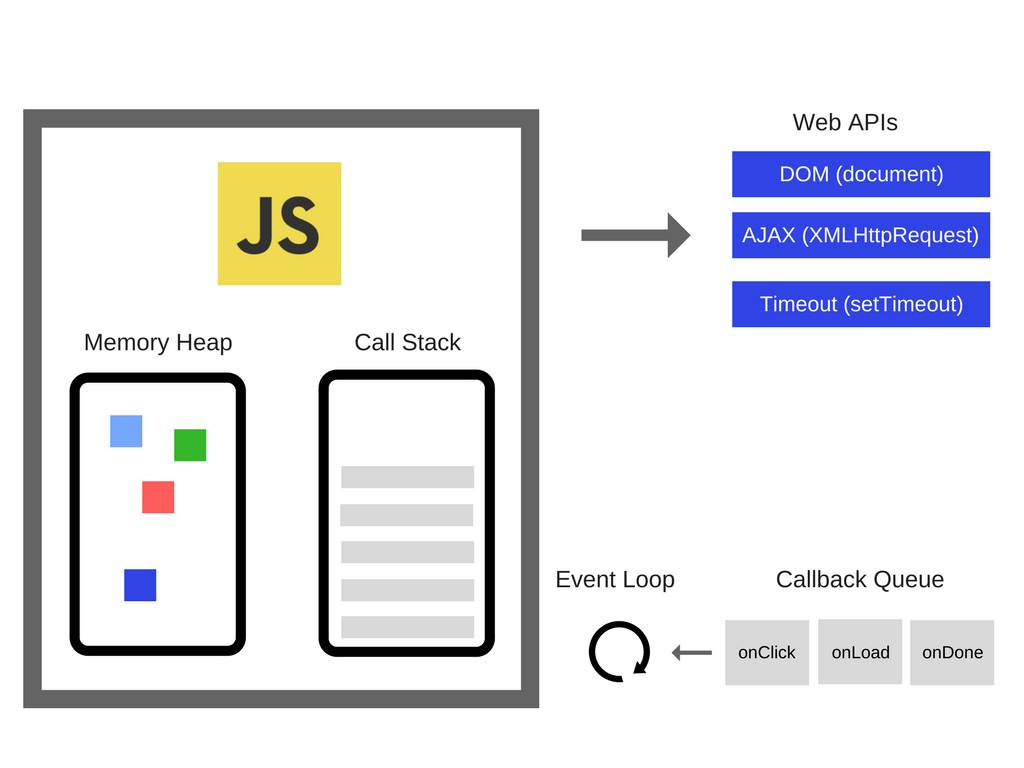
* getElementById
* *const* score1 = document.getElementById('score--1');

19 August 2021

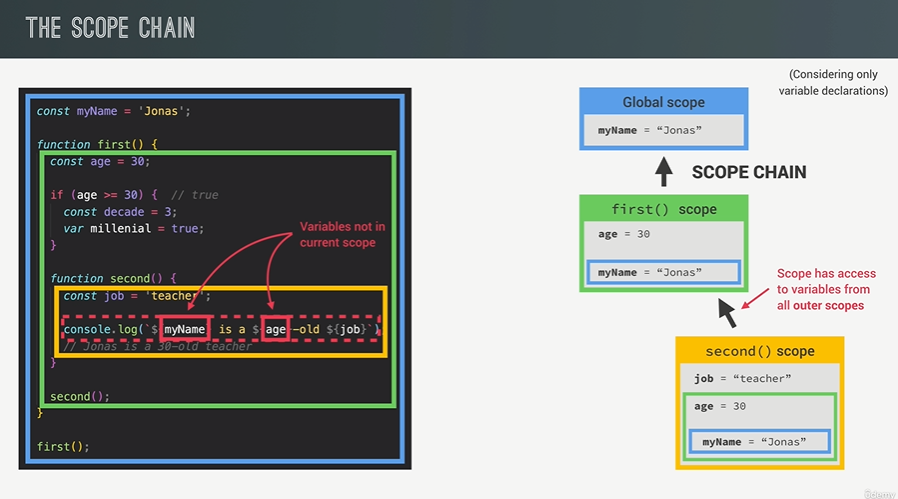
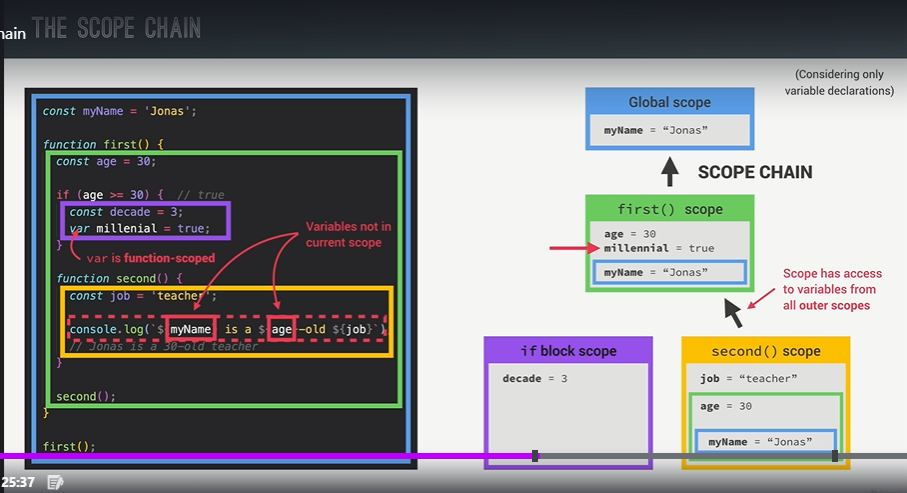
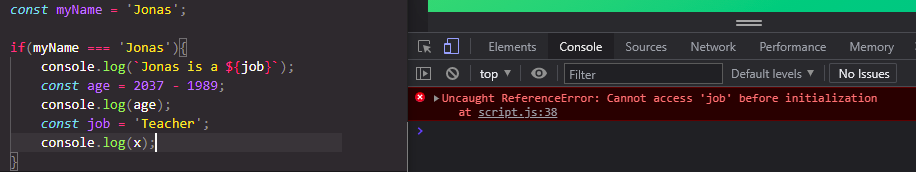
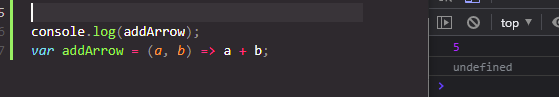
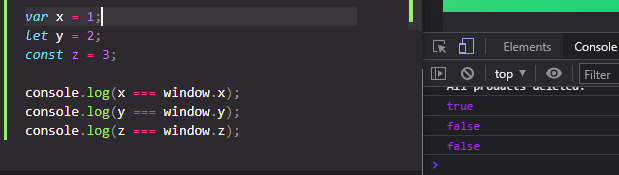
How Javascript works behind the scenes

* JS is a high level, object oriented, multi paradigm programming language.
  + High level – uses abstractions to shortcut
    - Low level – dealing with memory resource allocation etc etc
  + Garbage collected – removes memory of old data
  + Interpreted / just-in-time compiled – we write human written JS code that gets converted to machine code (0s and 1s)
  + Multi paradigm
    - What is a paradigm? An approach or programming mindset
    - Procedural
    - Object oriented (OOP)
    - Functional programming (FP)
  + Prototype based object oriented approach
    - Most in JS are objects except for primitive values (e.g numbers, strings etc) like arrays, objects
  + First class functions
    - Functions treated as regular variables – using functions within functions
  + Dynamically typed
    - No data type definitions – let x = 23; (*number)*
    - Automatic declaration of data type – x = “Jonas” (*string*)
  + Single threaded
    - Concurrency model – handling multiple tasks at the same time
    - JS runs on a single thread, one at a time
    - Non blocking event loop – an event loop takes long running tasks and executes them in the background, putting them into the main loop once complete
* Javascript engine and runtime
  + JS engine – a program that executes JS code
    - V8 engine – Google Chrome, nodeJS
  + What is in an engine?
    - A call stack – where code is executed through execution context
    - A heap – unstructured memory pool where objects are stored
  + How is code compiled to machine code?
    - Compilation vs interpretation
      * Compilation – entire code compiled to machine code at once, written to a binary file that can be executed by a computer
      * Source code – (compilation) – Portable file: Machine code – (execution, can happen way after compilation) – Program running
      * Interpretation – interpreter runs through the source code and executes it line by line
      * Modern JS uses a mix of compilation & interpretation for efficiency -> Just in time compilation
    - Just in time compilation
      * Code -> engine
        + Parse the code (read), into the Abstract Syntax Tree (AST)

Is the AST related to DOM? No.

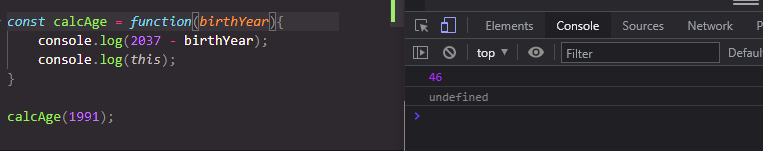
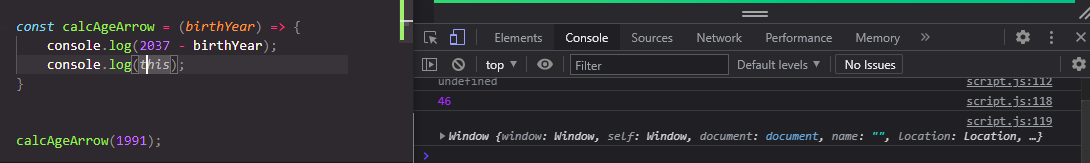
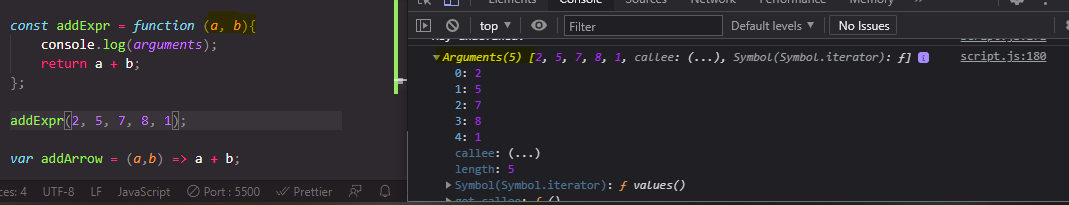
* + - * + The AST then gets compiled into machine code -> Execution straight away
      * The execution stage happens in ‘optimisation strategies’ – executes an unoptimized version of machine code just to get some legs, then optimised and recompiled later down the track
  + Javascript runtime in the browser– container that includes all we need to run JS (like Google Chrome)
  + 
* Execution contexts and the call stack
  + Stages
    - Compilation
      * Creation of global execution context (only for top level code e.g variable declaration, function declaration, not the context within a function etc)
        + Execution of top level code inside global EC

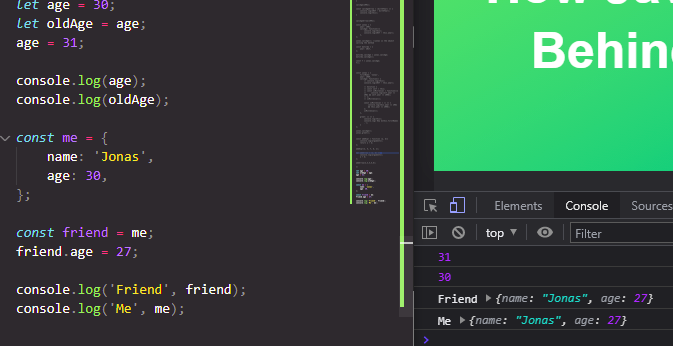
Execution of functions and waiting for callbacks

* + Execution context (EC)– environment in which a piece of JS is executed, stores all the necessary information to allow the execution
    - What’s inside EC?
      * Variable environment – let, const, var declarations
        + Functions
        + Arguments objects
      * Scope chain – references to variables outside of the current function, stored within each function
      * *This* keyword
  + The call stack - a place where ECs get stacked on top of each other to keep track of where we are in the execution
* Scope and the scope chain
  + What is scoping? How program variables are accessed and organised. Where do variables live and can be accessed and where not?
  + Lexical scoping – scoping is controlled by placement of functions and blocks in the code
  + Scope – environment in which a certain variable is declared. Global scope, function scope, block scope
  + Scope of a variable – region of our code where a certain variable can be accessed
  + The three types of scope
    - Global – for top level code, for any code outside of any function or block, accessible everywhere
    - Function (local scope) – variables accessible only inside the function
    - Block – anything held within {}, variables only accessible within the block (ONLY for let, const; var can be accessible outside of the block)
      * Functions are also block scoped
  + Scope chain – child scopes can always access variables within parent scopes, looks up from bottom to top until the right variable is found, if not the code stops
  + 
  + Note that the var ‘millenial’ variable is not block scoped but instead function scoped, in the first () function
  + 
  + Note that the block scope cannot access the second() scope – scope chain can only operate up/down, not sideways. The block scope is on the same level as the function scope second()
  + Scope chain vs call stack – the call stack is the ‘up-to-down’ order in which global scope / function scope / block scopes are *called* in the code. The scope chain determines the parent/child relationship of the scope, the order in which functions are *written* in the code.
  + 
  + Scoping – where do variables live and where and where can’t we access them?
  + Three types of scope – global scope, scopes defined by functions, scopes defined by blocks
  + Only let and const are block scoped, var end up in the closest function scope
  + Lexical scoping – rules where we can access variables based on exactly where the function and block scopes are written
  + Every scope always has access to variables written from all its outer scopes – this is the scope chain
  + Variable lookup – when a variable is not in the current scope, it looks up towards the parent scopes until it finds what it is looking for. This is a one way street.
* Scoping in practice
* Hoisting in Javascript – makes some variables accessible before they are actually declared
  + ‘Let example;’ - ?
  + Behind the scenes – the code is scanned for variable declarations and a new property is created in the *variable environment object*
  + Functions – yes, initial value is set based on actual function
  + Var variables – yes, initial value is ‘undefined’
  + Let/const – no, initial value is ‘uninitialised’, Temporal Dead Zone
  + Function expressions / arrows – depends on whether using var, let/const
* Temporal Dead Zone – JS knows about the variable being present but does not allow its use until its initialisation
  + 
  + Why have a TDZ? Accessing variables before initialisation is bad practice and should be avoided & make const actually work (to be assigned at the beginning and left alone)
* Hoisting and TDZ in practice
  + 
  + Var gets hoisted (allows for use before declaration) but is misleading bc initial value defaults to ‘undefined’
  + 

20 August 2021

The ‘this’ keyword

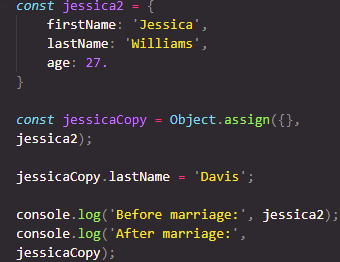
* This variable – One of the 3 ingredients for an execution context
  + Variable environment
  + Scope chain
  + This keyword
* This – special keyword created for every EC. Initial value points to the ‘owner’ of the function in which the *this* keyword is used
  + It is not static, and depends on how the function is called
* *const* jonas = {
* name: 'Jonas',
* year: 1989,
* calcAge: *function*() {
* return 2037 - *this*.year;
* }
* };
* jonas.calcAge();
  + Method – this takes on the object that is calling the method (the method being calcAge)
  + Simple function call – undefined
  + Arrow functions – this = <this of surrounding function (lexical this)>
  + Event listener – this = <DOM element that the handler is attached to>
* Simple function call example
* 
* Arrow function – grabs the *this* keyword of the parent function
* 
* Regular functions vs arrow functions
  + Arguments keyword can only exist in regular functions but not arrow. It is legal to have more arguments than what a function recognises:
* 
* Primitive vs Objects
  + Primitive data types
    - Number
    - String
    - Boolean
    - Undefined
    - Null
    - Symbol
    - BigInt
  + Objects
    - Object literal
    - Functions
    - Arrays
  + When talking about memory management - Primitive types & reference types
  + JS Engine – call stack & heap
    - Reference types – stored in heap
    - Primitive types – stored in call stack



|  |  |  |
| --- | --- | --- |
| CALL STACK | | |
| Identifier | Address | Value |
| Age | 0001 | 30 |
| oldAge | 0002 | 31 |
| Me | 0003 | D30F |
| Friend |  |  |

|  |  |
| --- | --- |
| HEAP | |
| Address | Value |
| D30F | Name: ‘Jonas’, age: 30, |
|  |  |

* Note how ‘friend’ is assigned as a const but is not immutable the object property can change. This is because it is the reference value is changing, not the value at location 0003.
* Object.assign({}, <object to be copied> -> Shallow assignment



* Deep clone

21 August 2021

Data structures, modern operators and strings

Deconstructing arrays

* Unpacking and assigning the existing values in the array

*const* arr = [2, 3, 4];

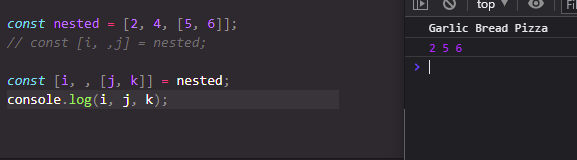
*const* a = arr[1];

*const* b = arr[2];

*const* c = arr[3];

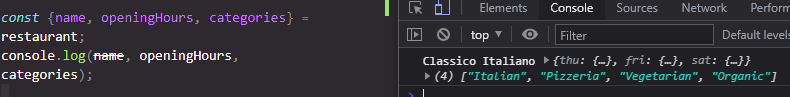
*const* [x, y, z] = arr;

* Reversing the order of an unpacked array
* *let* [main, , secondary] = restaurant.categories;
* [main, secondary] = [secondary, main];
* Nested destructuring

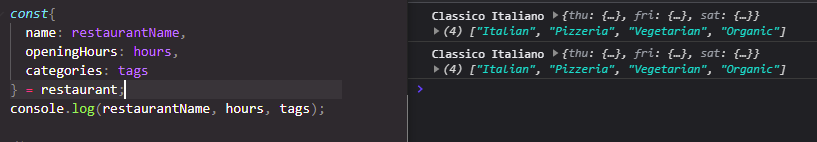


* Setting default values
* *//Default values*
* *const* [p = 1, q = 1, r = 1] = [8, 9];

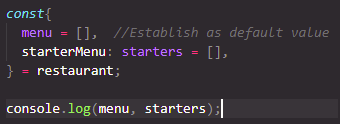
Object destructuring



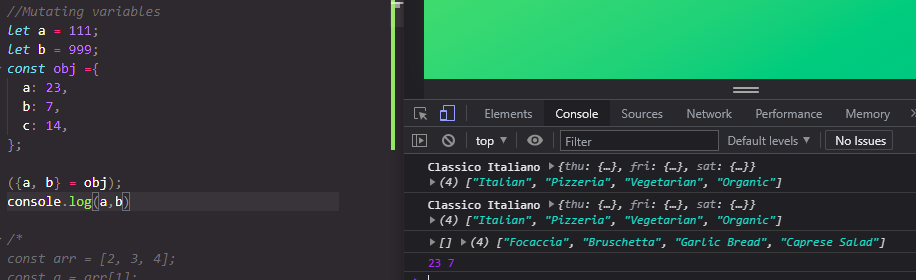
* What about unique variable names?



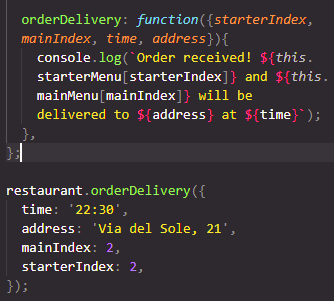
* Setting initial values as arrays



* Mutating variables within objects



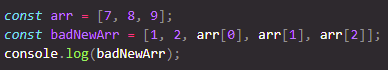
* Received information does not have to match order of function variables



25 August 2021

The Spread Operator

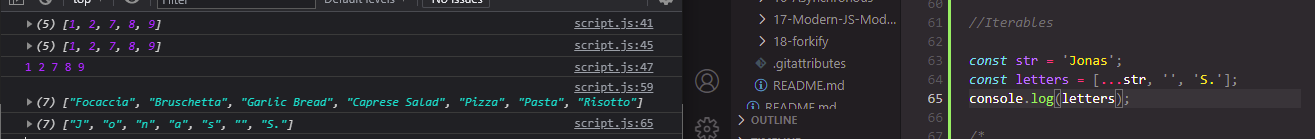
* A more convenient way to add new values to existing arrays

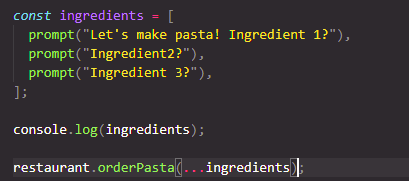


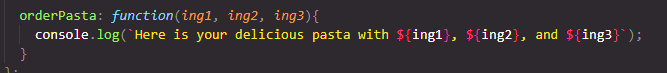
* Spread operator – grabs the values in the array (or more broadly any iterable) and writes them out individually



* Iterables – strings, arrays, maps and sets EXCEPT objects

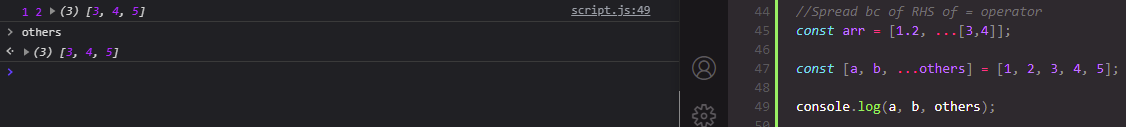






Rest pattern and parameters

* Rest pattern – does the opposite of spread, collects individual elements into an array



Short circuiting && and || (and & OR operators)

* Use any data type and return any data type



* Short circuiting – returns the very first truthy value
  + Returns ‘Hello’
* console.log(undefined || 0 || '' || 'Hello' || 23 || null);
* further examples
* *const* guests1 = restaurant.numGuests ? restaurant.numGuests : 10;
* console.log(guests1);
* *const* guests2 = restaurant.numGuests || 10;
* && - returns on the very first falsy value, if not, returns the last
* if (restaurant.orderPizza){
* restaurant.orderPizza('mushrooms', 'spinach');
* }
* *//Easy shortcut to if statement*
* restaurant.orderPizza && restaurant.orderPizza('mushrooms', 'spinach');

Nullish coalescing operator

restaurant.numGuests = 0;

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

console.log(guests1);

* In this case, 10 is returned since restaurant.numGuests would be a falsy value
* The nullish coalescing operator overcomes this
* *const* guestCorrect = restaurant.numGuests ?? 10;
* console.log(guestCorrect);
* Nullish values – null and undefined are considered falsy , a 0 would be treated as truthy

Coding Challenge #1

*//Coding Challenge #1*

*const* game = {

  team1: 'Bayern Munich',

  team2: 'Borrussia Dortmund',

  players: [

  [

  'Neuer',

  'Pavard',

  'Martinez',

  'Alaba',

  'Davies',

  'Kimmich',

  'Goretzka',

  'Coman',

  'Muller',

  'Gnarby',

  'Lewandowski',

  ],

  [

  'Burki',

  'Schulz',

  'Hummels',

  'Akanji',

  'Hakimi',

  'Weigl',

  'Witsel',

  'Hazard',

  'Brandt',

  'Sancho',

  'Gotze',

  ],

  ],

  score: '4:0',

  scored: ['Lewandowski', 'Gnarby', 'Lewandowski',

  'Hummels'],

  date: 'Nov 9th, 2037',

  odds: {

  team1: 1.33,

  x: 3.25,

  team2: 6.5,

  },

  };

*const* players1 = game.players[0];

  console.log(players1);

*const* players2 = game.players[1];

  console.log(players2);

*const* [gk, ...fieldPlayers] = players1;

*const* allplayers = [...players1, ...players2];

  console.log(allplayers);

*const* players1Final = [...players1, 'Thiango', 'Coutinho', 'Perisic'];

*const* {

    team1: team1,

    x: draw,

    team2: team2,

  } = game.odds;

*const* printGoals = *function*(...*numberPlayers*){

*const* [...totalGoals] = [...numberPlayers];

    console.log(totalGoals);

    console.log(`${numberPlayers} at total goals of ${totalGoals[0].length}`);

  };

  printGoals(players1);

  printGoals(game.scored);

  team1 < team2 && console.log('Team 1 is more likely to win');

  team2 < team1 && console.log('Team 2 is more likely to win');

26 August 2021

Looping array – the for-of loop

*const* menu = [...restaurant.starterMenu,...restaurant.mainMenu];

for (*const* item1 of menu) console.log(item1);

for (*const* item1 of menu.entries()) console.log(item1);

Background pattern

Description automatically generated

Enhanced object literals

* Easier way to ‘literally’ write an object

Optional chaining – checking if objects/arrays are present

if (restaurant.openingHours && restaurant.openingHours.mon){

  console.log(restaurant.openingHours.mon.open);

};

*//WITH optional chaining*

console.log(restaurant.openingHours.mon?.open);

*//Methods*

console.log(restaurant.order?.(0, 1) ?? 'Method does not exist');

console.log(restaurant.orderRisotto?.(0, 1) ?? 'Method does not exist');

* Optional chaining & nullish coalescing operator often used hand in hand

Looping objects – object.keys, object.values, object.entries

* For-of loop & Object.keys
* for(*const* day of Object.keys(openingHours)){
* console.log(day);
* }

*//Property values*

*const* values = Object.values(openingHours);

console.log(values);

*const* entries = Object.entries(openingHours);

for(*const* [key, {open, close}] of entries){

  console.log(`On ${key} we are open at ${open} and closed at ${close}`);

};

3 September 2021

Sets

* What is a set? A collection of unique values
* *const* ordersSet = new Set([
* 'Pasta',
* 'Pizza',
* 'Pizza',
* 'Risotto',
* 'Pasta',
* 'Pizza',
* ]);
* console.log(ordersSet);
* console.log(ordersSet.size);
* .has, .size, .add operations, .clear, .delete, .values
* Cannot retrieve specific values from a set
* Sets are treated as iterables – can use for-of, while, if/or
* Making an array out of a set
* *const* staff = ['Waiter', 'Chef', 'Waiter', 'Manager', 'Chef', 'Waiter'];
* *const* staffUnique = [...new Set(staff)];
* console.log(staffUnique);

Maps

* Data structure that maps values to keys

*const* rest = new Map();

rest.set('name', 'Classico Italiano');

rest.set(1, 'Firenze, Italy');

rest.set(2, 'Lisbon, Portugal');

console.log(rest);

rest

  .set('categories', ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'])

  .set('open', 11)

  .set('close', 23)

  .set(true, 'We are open')

  .set(false, 'We are closed');

  console.log(rest);

  console.log(rest.get('name'));

  console.log(rest.get(true));

  console.log(rest.get(1));

*const* time = 21;

  console.log(rest.get(time > rest.get('open') && time < rest.get('close')));

Maps – iteration

* Convert object to map
* *const* question = new Map([
* ['question', 'What is the best programming language in the world?'],
* [1, 'C'],
* [2, 'Java'],
* [3, 'JavaScript'],
* ['correct', 3],
* [true, 'Correct 👏'],
* [false, 'Try again!'],
* ]);
* console.log(question);
* console.log(Object.entries(openingHours));
* *const* hoursMap = new Map(Object.entries(openingHours));

console.log(question.get('question'));

for (*const* [key, value] of question) {

*// console.log(typeof (key));*

  if (typeof key === 'number') console.log(`Answer ${key} : ${value}`);

}

*const* answer = 3;

*// const answer = Number(prompt('Your answer'));*

console.log(answer);

console.log(question.get(question.get('correct') === answer));

*//Convert map to array*

console.log([...question]);

console.log(question.entries());

console.log(...question.keys());

console.log(...question.values());

Which data structure to use?

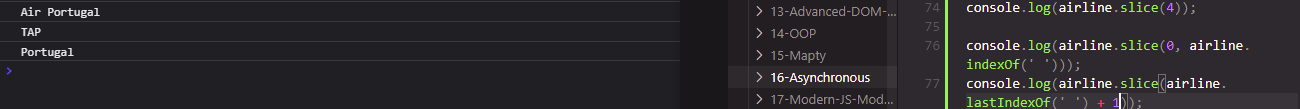
* Arrays / objects / sets / maps
* Sources of data?
  + From the program itself
  + From the user interface
  + From external sources
* Array / sets – simple list of values
* Objects / maps - Key / value pair
* Key differences
  + Arrays - Use when you need ordered data (and when retrieving / manipulating is important)
  + Sets – Use when you need to work on unique values (no duplicates), when high performance really matters
  + Objects – More traditional key/value stores, easier to write and access with . and []
  + Maps – better performance, keys can have any data type, easy to iterate, easy to compute size

4 September 2021

Working with strings

* Extracting individual elements within strings
* *const* airline = 'TAP Air Portugal';
* *const* plane = 'A320';
* console.log(plane[1]);
  + airline.length – 16 characters
* Methods – JS automatically converts the string in question to an object in order for methods to be applied
* console.log(airline.indexOf('r'));
* console.log(airline.lastIndexOf('r'));
* console.log(airline.indexOf('Portugal'));
  + .slice(start-index, end-index)





* Reverse extraction .slice (-X)
* Fixing capitalisation
* *const* passenger = 'jOnAS';
* *const* passengerLower = passenger.toLowerCase();
* *const* passengerCorrect = passengerLower[0].toUpperCase() + passengerLower.slice(1);
* console.log(passengerCorrect
* Check email - .trim email (removing whitespace)
* *const* email = 'hello@jonas.io';
* *const* loginEmail = '  Hello@Jonas.Io \n';
* *const* lowerEmail = loginEmail.toLowerCase();
* *const* trimmedEmail = lowerEmail.trim();
* console.log(trimmedEmail);
* *const* normalisedEmail = loginEmail.toLowerCase().trim();
* console.log(normalisedEmail);
* .replace(X,Y), .replaceAll(X, Y)
* *const* announcement = 'All passengers come to boarding door 23, boarding door 23!';
* console.log(announcement.replace('door', 'gate'). replace('door', 'gate'));
* *// console.log(announcement.replaceAll('door', 'gate'));*
  + Old school way – via regular expressions
* console.log(announcement.replace(/door/g, 'gate'));
* Booleans - .includes, .startsWith, .endsWith
* .split, .join – de-structuring and creating arrays from strings
* *const* [firstName, lastName] = 'Jonas Schmedtmann'.split(' ');

*const* newName = ['Mr', firstName, lastName.toUpperCase()].join(' ');

console.log(newName);

*const* capitaliseName = *function* (*name*) {

*const* nameArray = name.split(' ');

  console.log(nameArray);

*const* namesUpper = [];

  for(*const* word of nameArray){

    namesUpper.push(word[0].toUpperCase() + word.slice(1));

*// console.log(word[0]);*

  }

  console.log(namesUpper.join(' '));

};

capitaliseName('Jessica ann smith davis');

capitaliseName('julian goh');

* .padStart, .padEnd
* *const* maskCreditCard = *function*(*number*){
* *const* str = String(number);
* *const* strLength = str.length;
* console.log(str.slice(-4).padStart(strLength, 'X'));
* };
* Repeat
* *const* message2 = 'Bad weather... All departures delayed... ';
* console.log(message2.repeat(2));
* *const* planesInLine = *function* (*n*) {
* console.log(`There are ${n} planes in line ${'✈'.repeat(n)}`);
* };
* planesInLine(5);
  + Note that the plane repeat was needed to be encapsulated by ${} in order to use methods

Coding challenge

*//Coding Challenge #4*

document.body.append(document.createElement('textarea'));

document.body.append(document.createElement('button'));

document.querySelector('button').addEventListener('click',*function*(){

*const* text = document.querySelector('textarea').value;

  console.log(text);

*const* textArray = text.split('\n');

  console.log(textArray);

  console.log(textArray.entries());

  for(*const* [i, word] of textArray.entries()){

*const* lowerCase = word.toLowerCase().trim();

*const* afterUnderscore = lowerCase.slice(lowerCase.indexOf('\_') + 1);

*const* beforeUnderscore = lowerCase.slice(0,(lowerCase.indexOf('\_')));

*const* capitalise = afterUnderscore[0].toUpperCase() + afterUnderscore.slice(1);

*const* output = beforeUnderscore + capitalise;

    console.log(`${output.padEnd(20)}${'✅'.repeat(i + 1)}`);

  }

});

String Methods practice

*// Data needed for a later exercise*

*const* flights =

  '\_Delayed\_Departure;fao93766109;txl2133758440;11:25+\_Arrival;bru0943384722;fao93766109;11:45+\_Delayed\_Arrival;hel7439299980;fao93766109;12:05+\_Departure;fao93766109;lis2323639855;12:30';

*// 🔴 Delayed Departure from FAO to TXL (11h25)*

*//              Arrival from BRU to FAO (11h45)*

*//   🔴 Delayed Arrival from HEL to FAO (12h05)*

*//            Departure from FAO to LIS (12h30)*

*// const cutFlights = flights.replaceAll('+', '\n').replaceAll(';', ' ').replaceAll('\_', ' ');*

for(*const* flight of flights.split('+')){

*// console.log(flight);*

*const* individualFlight = flight.split(';');

*// console.log(individualFlight);*

*const* descrip = individualFlight[0].replace('\_', '').replace('\_', ' ').replace('Delayed', '🛑 Delayed');

*// console.log(descrip);*

*const* from = individualFlight[1].slice(0,3).toUpperCase();

*// console.log(from);*

*const* to = individualFlight[2].slice(0, 3).toUpperCase();

*// console.log(to);*

*const* time = `(${individualFlight[3]})`;

*// console.log(time);*

  console.log(`${descrip} from ${from} to ${to} ${time}`.padStart(50));

};

5 September 2021

Closer look at functions – higher order method, bind, closure

Default values

*const* createBooking = *function* (*flightNum*, *numPassengers* = 1, *price* = 199) {

*//ES5*

*// numPassengers = numPassengers || 1;*

*// price = price || 199;*

*const* booking ={

        flightNum,

        numPassengers,

        price,

    }

    console.log(booking);

    bookings.push(booking);

};

createBooking('LH123');

createBooking('LH123', 2, 800);

* Leaving function input fields as ‘undefined’
* createBooking('LH123', undefined, 800);
* How passing arguments work – value vs reference
  + Passing a primitive type into an function – is creating a copy
  + Passing an object into a function – affects the original reference values
  + \*\* Javascript can only pass by values
* First class and higher order functions
  + Functions are treated as first class citizens, which means functions are simply values. Because functions are values, they can be stored in variables or properties.
  + Pass functions as arguments to other functions
  + Functions are another type of object – methods can be applied
  + Higher order functions:
    - A function that receives another function as an argument that then returns another function, or BOTH
      * The higher order function – addEventListener
      * Callback function - greet
* *const* btnClose = document.querySelector('.buy');
* btnClose.addEventListener('click', greet);
  + First class & higher order functions mean different things
* Functions accepting callback functions
* *const* oneWord = *function* (*str*) {
* return str.replace(/ /g, '').toLowerCase();
* };
* *const* upperFirstWord = *function* (*str*) {
* *const* [first, ...others] = str.split(' ');
* return [first.toUpperCase(), ...others].join(' ');
* };
* *// console.log(upperFirstWord('Apache Helicopter'));*
* *// const str = 'Apache Helicopter';*
* *// const[first, ...others] = str.split(' ');*
* *// console.log(first, ...others);*
* *//Higher order function*
* *const* transformer = *function* (*str*, *fn*) {
* console.log(`Original string: ${str}`);
* console.log(`Transformed string: ${fn(str)}`);
* console.log(`Transformed by: ${fn.name}`)
* };
* transformer('JavaScript is the best!', upperFirstWord);
  + .forEach(input-function)
* Why are callback functions important? To allow for abstraction (remove clutter, hide the minutiae)
  + Higher / lower levels of abstraction
* Functions returning functions
* *const* greet = *function*(*greeting*){
* return *function*(*name*){
* console.log(`${greeting} ${name}`);
* }
* }
* *const* greeterHey = greet('Hey');
* greeterHey('Jonas');

greet('Hello')('Jonas');

Converting to arrow:

*const* greet = *greeting* => *name* => console.log(`${greeting} ${name}`);

*const* greeterHey = greet('Hey');

greeterHey('Jonas')

* This keyword
* *const* lufthansa = {
* airline: 'Lufthansa',
* iataCode: 'LH',
* bookings: [],
* book: *function* (*flightNum*, *name*) {
* console.log(
* `${name} booked a seat on ${*this*.airline} flight ${*this*.iataCode}${flightNum}`
* );
* },
* };
* lufthansa.book(239, 'Jonas Schmedtmann');
* lufthansa.book(635, 'John Smith');
* Explicitly directing JS to know which ‘this’ it should be looking at - .call, .apply
* *const* eurowings = {
* airline: 'Eurowings',
* iataCode: 'EW',
* bookings: [],
* };
* *const* book = lufthansa.book;
* *// book(23, 'Sarah Williams');*
* book.call(eurowings, 23, 'Sarah Williams')

*//Apply method*

*const* flightData = [583, 'George Cooper'];

book.apply(swiss, flightData);

*//Call & spread - best practice*

book.call(swiss, ...flightData);

The bind method – allows permanent assignment of ‘this’ to a specific object

*const* bookEW = book.bind(eurowings);

bookEW(23, 'Steven Williams');

* Partial application
* *const* bookEW23 = book.bind(eurowings, 23);
* bookEW23('Jack Brown');

*/ With Event Listeners*

lufthansa.planes = 300;

lufthansa.buyPlane = *function* () {

  console.log(*this*);

*this*.planes++;

  console.log(*this*.planes);

};

*const* buyPlane = lufthansa.buyPlane.bind(lufthansa);

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

console.log(lufthansa);

*const* buyPlane = lufthansa.buyPlane.bind(lufthansa);

document.querySelector('.buy').addEventListener('click', buyPlane);

console.log(lufthansa);

Partial application – further example

*// Partial application*

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

console.log(addTax(0.1, 200));

*const* addVAT = addTax.bind(null, 0.23);

console.log(addVAT(100));

*//Alternative - callback*

*const* addTax1 = *function*(*rate*){

    return *function*(*value*){

        console.log(`${value + value \* rate}`)

    }

}

*const* totalTax = addTax1(0.23);

totalTax(100);

Coding challenge

*const* poll = {

  question:

    'What is your favourite programming language?\n 0: JavaScript \n 1: Python \n 2: Rust \n 3: C++ \n (Write option number)',

  answers: [0, 0, 0, 0],

  registerNewAnswer: *function* () {

*const* option = Number(prompt(`${*this*.question}`));

*// this.array.push(option);*

    console.log(`Your answer is ${option}`);

    if (option === 0) {

*this*.answers[0] += 1;

    } else if (option === 1) {

*this*.answers[1] += 1;

    } else if (option === 2) {

*this*.answers[2] += 1;

    } else if (option === 3) {

*this*.answers[3] += 1;

    } else {

      console.log('Invalid input');

    }

*// console.log(this.answers);*

*this*.displayResults('string');

  },

  displayResults: *function* (*type*) {

    if (type === 'array') {

      console.log(*this*.answers);

    } else if (type === 'string') {

      console.log(`Poll results are ${*this*.answers.join(', ')}`);

    }

  },

};

document

  .querySelector('.poll')

  .addEventListener('click', poll.registerNewAnswer.bind(poll));

*const* bonus = {

*// answers: [5, 2, 3],*

  answers: [1, 5, 3, 9, 6, 1],

};

*const* bonusDisplay = poll.displayResults.bind(bonus);

bonusDisplay('string');

12 September 2021

Immediately Invoked Function Expressions (IIFE) – a function that disappears after it is called once

(*function*(){

  console.log('This will never run again');

});

(() => console.log('This will never run again'))();

* Reminder of global / local scopes – global cannot access local variables but local variables can access higher scope variables

Closures – bringing together Execution Contexts, Call Stack and Scope Chain

*const* secureBooking = *function* () {

*let* passengerCount = 0;

  return *function*(){

    passengerCount++;

    console.log(`${passengerCount}`);

  }

};

*const* booker = secureBooking();

Call stack

* Global execution context – secureBooking = <f>
  + secureBooking() Execution Context created, a variable environment that contains all its variables
* Global scope – secureBooking = <f>

Strange how the local environment can be kept updated each time the booker function gets called – after all, the local variable gets defined each function call (?)

* Closure – lets the function remember all the variables at their birthplaces e.g passengerCount within the secureBooking function
  + A function has access to the variable environment of the execution context in which it was created, even after the EC is removed from the call stack
* Formal definition - A closure is the closed-over variable environment of the execution context in which a function was created, even after that execution context is gone
  + OR – A closure gives a function access to all its variables of its parent function even after that parent function has returned. The function keeps a reference to its outer scope which preserves the scope chain through time
* Closures happen automatically, are not a tangible JavaScript object
* Console.dir(booker)

Text

Description automatically generated

* Another example
* *let* f;
* *const* g = *function*(){
* *const* a = 23;
* f = *function*(){
* console.log(a \* 2);
* }
* }
* g();
* f();
* What if f gets re-assigned?
* *const* h = *function*(){
* *const* b = 777;
* f = *function*(){
* console.log(b \* 2);
* }
* }
* g();
* f();
* h();
* f();

Working with Arrays

* Array Methods – methods are simply functions we can call on objects, arrays themselves are therefore considered as objects
* *let* arr = ['a', 'b', 'c', 'd', 'e'];
* console.log(arr.slice(2));
* console.log(arr.slice(2, 4));

*//Creating shallow copies*

console.log(arr.slice());

console.log([...arr]);

Splice method – similar function to slice but permanently mutates the original variable

Reverse method – reverses the order of the array, mutates original variable

*const* letters = arr.concat(arr2);

console.log(letters);

console.log([...arr, ...arr2]);

forEach method

for(*const* movement of movements){

  if(movement > 0){

    console.log(`You deposited ${movement}`);

    } else {

      console.log(`You withdrew ${Math.abs(movement)}`);

    }

}

movements.forEach(*function*(*movement*){

  if(movement > 0){

    console.log(`You deposited ${movement}`);

    } else {

      console.log(`You withdrew ${Math.abs(movement)}`);

    }

})

* Note that .forEach automatically accepts index / array as entries
* for(*const* [i, movement] of movements.entries()){
* if(movement > 0){
* console.log(`Movement ${i + 1}: You deposited ${movement}`);
* } else {
* console.log(`Movement ${i + 1}: You withdrew ${Math.abs(movement)}`);
* }
* }
* *//Equivalent process - through forEach (higher order) function*
* movements.forEach(*function*(*movement*, *index*, *array*){
* if(movement > 0){
* console.log(`Movement ${index + 1}: You deposited ${movement}`);
* } else {
* console.log(`Movement ${index + 1}: You withdrew ${Math.abs(movement)}`);
* }
* });

\*\* Refresh - .entries method – breaks down an array into its index & value

forEach with Maps and Sets

*const* currencies = new Map([

  ['USD', 'United States dollar'],

  ['EUR', 'Euro'],

  ['GBP', 'Pound sterling'],

]);

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

  console.log(`${key}: ${value}`);

});

Bankist application

Diagram

Description automatically generated

Creating DOM elements

* .insertAdjacentHTML
* .innerHTML – similar function to textContent

Array methods - data transformations of map, filter, reduce

* Map – similar to for/each, creates a brand new array from the previous data
  + Returns a new array containing the results of applying an operation on all original array elements
* Filter – returns a new array containing the array elements that pass a specified test condition
* Reduce – reduces all array elements down to a single value

The Map method

*const* movements = [200, 450, -400, 3000, -650, -130, 70, 1300];

*const* eurToUSD = 1.1;

*const* movementsUSD = movements.map(*function*(*mov*){

  return mov \* eurToUSD;

});

console.log(movements);

console.log(movementsUSD);

* Using the arrow function for cleaner (but less readable) code

*const* show = movements.map(*function*(*movement*, *i*, *arr*){

  if(movement > 0){

    console.log(`Movement ${i + 1}: You deposited ${movement}`);

    } else {

      console.log(`Movement ${i + 1}: You withdrew ${Math.abs(movement)}`);

    }

});

*const* show1 = movements.map((*movement*, *i*, *arr*) => {

  if(movement > 0){

    console.log(`Movement ${i + 1}: You deposited ${movement}`);

    } else {

      console.log(`Movement ${i + 1}: You withdrew ${Math.abs(movement)}`);

    }

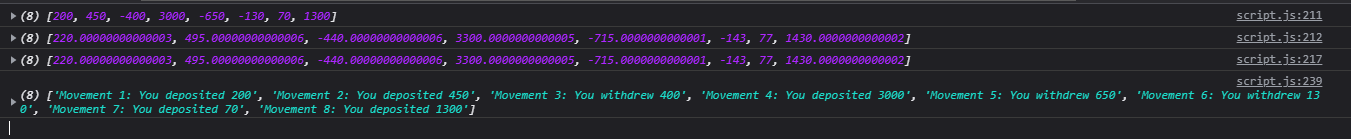
});

*const* movementDescriptions = movements.map(

  (*movement*, *i*) =>

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

  );



Creating usernames

*const* createUsernames = *function*(*accs*){

  accs.forEach(*function*(*account*){

    account.username = account.owner.toLowerCase()

    .split(' ')

    .map(*name* => name[0])

    .join('');

  });

}

createUsernames(accounts);

console.log(accounts);

13 September 2021

The Filter method

*const* deposits = movements.filter(*function*(*mov*){

  return mov > 0;

});

console.log(movements);

console.log(deposits);

* How does this compare to the original for/of loop?

*const* depositsFor = [];

for(*const* mov of movements){

  if (mov > 0) depositsFor.push(mov);

};

console.log(depositsFor);

The Reduce method

*const* sum = movements.reduce(*function*(*acc*, *cur*, *i*, *arr*){

  return acc + cur;

}, 0);

console.log(sum);

* Note how the initiating variables of the function are very different - accumulator
* 0 initial value

A black screen with white text

Description automatically generated with low confidence

* For/each comparison
* *let* balance2 = 0;
* for(*const* mov of movements) balance2 += mov;
* console.log(balance2);

Coding challenge

*//Coding challenge #2*

*const* data1 = [5, 2, 4, 1, 15, 8, 3];

*const* data2 = [16, 6, 10, 5, 6, 1, 4];

*const* calcAverageHumanAge = *function*(*ages*){

*const* convertAge = ages.map(*function*(*dogAge*){

    if(dogAge <= 2){

      return 2 \* dogAge;

    } else {

      return 16 + dogAge \* 4;

    }

  });

  console.log(convertAge);

*const* convertAgeFiltered =  convertAge.filter(*function*(*age*){

    return age > 18;

  });

  console.log(convertAgeFiltered);

*// const convertAgeAve = convertAgeFiltered.reduce(function(acc, curr){*

*//   return acc + curr;*

*// }, 0) / convertAgeFiltered.length;*

*const* convertAgeAve = convertAgeFiltered.reduce(*function*(*acc*, *curr*, *i*, *arr*){

    return acc + curr / arr.length;

  }, 0);

  console.log(convertAgeAve);

};

calcAverageHumanAge(data1);

calcAverageHumanAge(data2);

The magic of chaining methods – filter, map, reduce

*onst* movements = [200, 450, -400, 3000, -650, -130, 70, 1300];

*const* eurToUSD = 1.1;

*const* totalDepositsUSD = movements

  .filter(*mov* => mov > 0)

  .map(*mov* => mov \* eurToUSD)

  .reduce((*acc*, *mov*) => acc + mov, 0);

console.log(totalDepositsUSD);

* Good practice of how to debug a codeblock with stacked methods – using the parameters of the callback functions
* *const* totalDepositsUSD = movements
* *// .filter(mov => mov > 0)*
* *// .map(mov => mov \* eurToUSD)*
* *// .reduce((acc, mov) => acc + mov, 0);*
* .filter((*mov*, *i*, *arr*) => {
* console.log(arr);
* return mov > 0;
* })
* .map((*mov*, *i*, *arr1*) => {
* console.log(arr1);
* return mov \* eurToUSD
* })
* .reduce((*acc*, *mov*) => acc + mov, 0);
* console.log(totalDepositsUSD);
* Bad practice to chain methods that permanently mutates the original data

Coding challenge #3 – chaining practice

*const* calcAverageHumanAge1 = *ages* => {

*const* convertAge = ages

    .map(*function* (*dogAge*) {

      if (dogAge <= 2) {

        return 2 \* dogAge;

      } else {

        return 16 + dogAge \* 4;

      }

    })

    .filter(*function* (*age*) {

      return age > 18;

    })

    .reduce(*function* (*acc*, *curr*, *i*, *arr*) {

      return acc + curr / arr.length;

    }, 0);

    console.log(convertAge);

};

The Find Method – returns the first element in an array that matches the condition, does not return an array like the filter method

*// Find method*

*const* movements = [200, 450, -400, 3000, -650, -130, 70, 1300];

*const* firstWithdrawal = movements.find(*mov* => mov < 0);

* Searching for a user out of an array of accounts

*const* account = accounts.find(*acc* => acc.owner === 'Jessica Davis');

console.log(account);

Creating user login functionality

btnLogin.addEventListener('click',*function*(*e*){

*//Prevent form from submitting*

  e.preventDefault();

  console.log('LOGIN');

})

Check if user details exist

if (currentAccount?.pin === Number(inputLoginPin.value)){

    console.log('LOGIN');

  }

.blur – remove cursor from login

*//Clear login fields*

  inputLoginUsername.value = inputLoginPin.value = '';

  inputLoginPin.blur();

5 October 2021

Implementing Transfers

findIndex method – account deletion

*const* index = accounts.findIndex(

*acc* => acc.username === currentAccount.username

    );

    accounts.splice(index, 1);

some and every

- .includes (to check for specific value in array)

- Some – Boolean check

*const* anyDeposits = account1.movements.some(*mov* => mov > 150);

- Every - if every element passes the test in the callback function, only then it returns true

\*\* Check if only deposits

*// Every method*

console.log(account1.movements.every(*mov* => mov > 0));

Flat and flatmap methods

- .flat() - Unpacks nested arrays into a single full array, but is limited to surface layer only



*const* arrDeep = [[[1,2],3], [4,[5,6]], 7, 8];

console.log(arrDeep.flat(2));

Calculating total value of all movements across all the accounts

*const* accountMovements = accounts.map(*acc* => acc.movements);

console.log(accountMovements);

*const* allMovements = accountMovements.flat();

console.log(allMovements);

*//JG method lol*

*let* bankTotal = 0;

for(*const* mov of accountMovements.flat()){

  bankTotal = bankTotal + mov;

}

console.log(bankTotal);

*//JS method*

*const* overallBalance = allMovements.reduce((*acc*,*mov*) => acc + mov)

console.log(overallBalance);

- flatMap – DRY principle of dealing with nested arrays

*const* overallBalance = accounts

  .map(*acc* => acc.movements)

  .flat()

  .reduce((*acc*, *mov*) => acc + mov, 0);

console.log(overallBalance);

*//Flat map*

*const* overallBalance2 = accounts

  .flatMap(*acc* => acc.movements)

  .reduce((*acc*, *mov*) => acc + mov, 0);

console.log(overallBalance2);

Sorting Arrays

.sort() – works well for strings but not numbers

*const* owners = ['Jonas', 'Zach', 'Adam', 'Martha'];

console.log(owners.sort());

accounts1.sort((*a*, *b*) => {

  if (a > b) return 1;

  if (b > a) return -1;

});

Even DRYier

console.log(account1.movements.sort((*a*,*b*) => a - b));

More ways of creating and filling arrays

- new Array()



.fill method

- .fill(value, start-index, end-index)





Array.from() method

*const* z = Array.from({length: 7}, (*cur*, *i*) => i + 1);



Which array method to use?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| To mutate original array | A new array | An array index | Know if array includes | To transform to value |
| To add to original:   * .push * .unshift   Remove from original   * .pop * .shift * .splice   Others:   * .reverse * .sort * .fill | Compute from original:   * .map   Filtered using condition   * .filter   Portion of original   * .splice   Add original to other   * .concat   Flattening the original   * .flat * .flatMap | Based on value   * .indexOf   Based on test condition   * .findIndex | Based on value   * .includes   Based on test condition   * .some * .every | Based on accumulator   * .reduce |
| An array element  Based on test condition   * .find | A new string  Based on separator string:   * .join | To just loop array  Based on callback (do some work but not produce anything new)   * .forEach |

6 October 2021

* ++ operator
  + ++count vs count++
* Making a new object from .reduce method
* *//*
* *const* { deposits, withdrawals } = accounts
* .flatMap(*acc* => acc.movements)
* .reduce(
* (*sums*, *curr*) => {
* *// curr > 0 ? (sums.deposits += curr) : (sums.withdrawals += curr);*
* sums[curr > 0 ? 'deposits' : 'withdrawals'] += curr;
* return sums;
* },
* { deposits: 0, withdrawals: 0 }
* );
* console.log(deposits, withdrawals);

*// 4. this is a nice title > This Is a Nice Title*

*const* convertTitleCase = *function* (*title*) {

*const* exceptions = ['a', 'an', 'the', 'but', 'or', 'on', 'in', 'with'];

*const* titleCase = title

    .toLowerCase()

    .split(' ')

    .map(*word* =>

      exceptions.includes(word) ? word : word[0].toUpperCase() + word.slice(1)

    );

  return titleCase;

};

console.log(convertTitleCase('this is a nice title'));

7 October 2021

Converting and checking numbers

* Numbers are stored in JS behind the scenes as integers
* Base 10 vs binary base 2
  + 0 to 9
  + 0s and 1s
  + Avoid really scientific calculations in Javascript
* String to numbers conversion
  + Number()
  + Type coercion
    - Console.log(+’23’);
* Parsing – Number.parseInt (‘ ‘, 10 or 2), Number.parseFloat

Graphical user interface, text, application

Description automatically generated

* ParseFloat – best way to grab numbers from strings
* isNan
* isFinite – best way to check if a value is a number
* console.log(Number.isFinite(23 / 0));

Other mathematical operations

* Squareroot, exponentiation, pi, min/max
* console.log(Math.sqrt(25));
* console.log(25 \*\* (1/2));
* console.log(Math.max(5, 18, '23', 11, 2));
* console.log(Math.max(5, 18, '23px', 11, 2));
* console.log(Math.min(5, 18, '23', 11, 2));
* console.log(Math.PI \* Number.parseFloat('10px') \*\* 2);
* Random number generator within set range
* *//Random dice roll and random number generator*
* console.log(Math.trunc(Math.random() \* 6) + 1);
* *const* randomInt = (*min*, *max*) => Math.trunc(Math.random() \* (max - min) + 1) + min;
* *// 0 ... 1 -> 0 ... (max - min) -> min ... max*
* Rounding integers
  + Math.round
  + Math.trunc
  + Math.floor
  + Math.ceil
* Math.trunc vs Math.floor
  + Floor is applicable to -ve numbers as well
* Rounding decimals – toFixed, can do type coercion
* console.log((2.7).toFixed(3));



Note that the output is a string, to overcome this add the + sign (easy shortcut to Number() )

console.log(+(2.345).toFixed(2));

8 October 2021

The Remainder operator

console.log(5 % 2);

* Useful for checking even or odd numbers
  + 8 % 2 = 0
  + 9 % 2 = 1

labelBalance.addEventListener('click', *function*(){

[...document.querySelectorAll('.movements\_\_row')].forEach(*function*(*row*,*i*){

*// 0, 2, 4, 6*

  if(i % 2 === 0) row.style.backgroundColor = 'orangered';

*// 0, 3, 6, 9*

  if(i % 3 === 0) row.style.backgroundColor = 'blue';

});

});

bigInt

* Maximum JS limitation in representing integers
* console.log(2 \*\* 53 - 1);
* console.log(Number.MAX\_SAFE\_INTEGER);
* What if we have to deal with numbers bigger than this? This is where bigInt comes in
* console.log(100213123123210*n* \* 100000*n*);

*//Convert regular numbers to bigint*

console.log(2312931239812*n* + BigInt(23));

* Logical operators are exceptions – can mix bigint with normal integers
* console.log(20*n* > 15); // True
* console.log(20*n* == '20'); // True, loose equality operator coercion
* String concatenations – converts bigint to string

console.log(123123123123*n* + 'is REALLY BIG!');

Creating dates

* The four ways
  + New Date()
* *const* now = new Date();
* console.log(now);

console.log(new Date(2037, 10, 19, 15, 23, 5));

Thu Nov 19 2037 15:23:05 GMT+1000 (Australian Eastern Standard Time)

\*\*Note that Javascript works the month in zero-based

console.log(new Date(0));

console.log(new Date(3 \* 24 \* 60 \* 60 \* 1000));

Sun Jan 04 1970 10:00:00 GMT+1000 (Australian Eastern Standard Time)

* .getFullYear() method – do not use .getYear \*\*
* .getMonth() method
* .getDate();
* .getDay();
* .getHours();
* .getMinutes();
* .getSeconds();
* .toISOString();
* .getTime() – time stamp, ms since Jan 1 1970
* Date.now(); - shortcut to get current timestamp
* *const* newISO = new Date(future.getTime());
* console.log(newISO.toISOString());
* .setFullYear()… month.. minutes etc

Formatting dates with .padStart

*//Create current date and time*

*const* now = new Date();

*const* day = `${now.getDate()}`.padStart(2, 0);

*const* month = `${now.getMonth() + 1}`.padStart(2, 0);

*const* year = now.getFullYear();

*const* hour = `${now.getHours()}`.padStart(2, 0);

*const* min = `${now.getMinutes()}`.padStart(2, 0);

    labelDate.textContent = `${day}/${month}/${year}, ${hour}:${min}`

Operations with dates

*const* formatMovementDate = *function* (*date*){

*const* calcDaysPassed = (*date1*, *date2*) => Math.round(Math.abs(date2 - date1) / (1000 \* 60 \* 60 \* 24));

*const* daysPassed = calcDaysPassed(date, new Date());

  console.log(daysPassed);

  if(daysPassed === 0) return 'Today';

  if(daysPassed === 1) return 'Yesterday';

  if(daysPassed <= 7) return `${daysPassed} days ago`;

  else{}

*const* day = `${date.getDate()}`.padStart(2, '0');

*const* month = date.getMonth() + 1;

*const* year = date.getFullYear();

  return `${day}/${month}/${year}`;

}

Internationalising dates – formatting dates/times across different languages/countries

*const* now = new Date();

*const* options ={

  day: 'numeric',

  month: 'long',

  year: 'numeric',

  hour: 'numeric',

  minute: 'numeric',

  weekday: 'long',

}

*const* locale = navigator.language;

labelDate.textContent = new Intl.DateTimeFormat(locale, options).format(now);  *//Google 'ISO language code table'*

Dates – ‘today’, ‘yesterday’ etc

*const* formatMovementDate = *function* (*date*, *locale*) {

*const* calcDaysPassed = (*date1*, *date2*) =>

    Math.round(Math.abs(date2 - date1) / (1000 \* 60 \* 60 \* 24));

*const* daysPassed = calcDaysPassed(date, new Date());

*// console.log(daysPassed);*

  if (daysPassed === 0) return 'Today';

  if (daysPassed === 1) return 'Yesterday';

  if (daysPassed <= 7) return `${daysPassed} days ago`;

  else {

  }

*// const day = `${date.getDate()}`.padStart(2, '0');*

*// const month = date.getMonth() + 1;*

*// const year = date.getFullYear();*

*// return `${day}/${month}/${year}`;*

  return new Intl.DateTimeFormat(locale).format(date);

};

Internationalising numbers

*const* num = 23237.23;

*const* options = {

  style: "unit",

  unit: 'mile-per-hour',

}

console.log('US:    ', new Intl.NumberFormat('en-US', options).format(num));

console.log('Germany:    ', new Intl.NumberFormat('de-DE', options).format(num));

Timers: setTimeout and setInterval

setTimeout((*ing1*, *ing2*) => console.log(`Here is your pizza 🍕 with ${ing1} and ${ing2}`), 3000, 'olives', 'spinach');

Here is your pizza 🍕 with olives and spinach

*const* ingredients = ['olives', 'spinach'];

*const* pizzaTimer = setTimeout((*ing1*, *ing2*) => {console.log(`Here is your pizza 🍕 with ${ing1} and ${ing2}`); console.log(...ingredients)} , 3000, ...ingredients);

if(ingredients.includes('chicken')) clearTimeout(pizzaTimer);

Implementing a countdown timer

*const* startLogOutTimer = *function* () {

*const* tick = *function* () {

*const* min = String(Math.trunc(time / 60)).padStart(2, 0);

*const* sec = String(time % 60).padStart(2, 0);

*// In each call, print remaining time to UI*

    labelTimer.textContent = `${min}:${sec}`;

*//When 0 seconds, stop timer and log out user*

    if (time === 0) {

      clearInterval(timer);

      labelWelcome.textContent = 'Log in to get started';

      containerApp.style.opacity = 0;

      currentAccount = {};

    }

*// Decrease 1s*

    time--;

  };

*// Set time to 5 minutes*

*let* time = 120;

*// Call the timer every second*

  tick();

*const* timer = setInterval(tick, 1000);

  return timer;

};

14 October 2021

Bankist landing page

What is the DOM? Review

* Allow JS to interact with browser
  + Create, modify and delete HTML elements
  + Set styles, classes and attributes
  + Listen and respond to events
* DOM tree is generated from HTML document which we can then interact with
* DOM is a very complex API that contains lots of methods and properties to interact with the DOM tree “document.addEventListener()”
* DOM API behind the scenes – all DOM elements are represented by “node” type
  + Node type – that have access to methods (.textContext, .childNodes, .parentNode) represented by JS object
    - Element type - **<p>** Paragraph **</p>**
      * HTMLElement
        + HTMLButtonElement
        + HTMLDivElement
    - Text type - <p> **Paragraph** </p>
    - Comment type - <!—**Comment** --!>
    - Document type
      * .querySelector()
      * .createElement()
      * .getElementbyId()
  + Inheritance – all child types will inherit all the properties of its parent types but the inverse does not work
    - A HTMLElement is also an Element and a Node
  + Why does addEventListener work on all DOM elements?
    - Event Target - Because it sits above the Node type in the inheritance, parent of the Node and Window node type

Diagram

Description automatically generated

Selecting, creating and deleting elements

* Document., querySelector, querySelectorAll, getElementbyId, getElementsByTagName, getElementsbyClassName
  + Outputs – NodeList (static) vs HTMLCollection (live)
* console.log(document.documentElement);
* console.log(document.head);
* console.log(document.body);
* document.querySelector('.head');
* *const* allSections = document.querySelectorAll('.section');
* console.log(allSections); *// This forms a NodeList*
* document.getElementById('section--1');
* *const* allButtons = document.getElementsByTagName('button');
* console.log(allButtons); *//Note that this produces a HTMLCollection - this is a live collection, will reflect any changes that occur in the HTML*
* console.log(document.getElementsByClassName('btn'))
* Using both .prepend and .append simultaneously requires .cloneNode(true)
* header.prepend(message);
* *// header.append(message);*
* header.append(message.cloneNode(true));
* . before and .after
* .remove()
* *// Delete elements*
* document.querySelector('.btn--close-cookie').addEventListener('click', *function*(){
* message.remove();
* *// message.parentElement.removeChild(message);*
* });

Styles, attributes and classes

* Inline styles - .style
* message.style.backgroundColor = '#37383d';
* message.style.width = '120%';
* .getComputedStyle
* console.log(getComputedStyle(message).color);
* Editing directly the height of html element
* message.style.height = Number.parseFloat(getComputedStyle(message).height,10) + 40 + 'px';
* */ Editing custom properties*

document.documentElement.style.setProperty('--color-primary', 'orangered');

Attributes

*// Attributes*

*const* logo = document.querySelector('.nav\_\_logo');

console.log(logo.src);

console.log(logo.alt);

console.log(logo.className);

* Custom attributes
  + Reading
* console.log(logo.getAttribute('designer'));
* logo.setAttribute('company', 'Bankist');
* Note .getAttribute on src
* console.log(logo.src);
* console.log(logo.getAttribute('src')); *// Relative, local location*
* Data attributes – accessing custom names with .dataset
* *//Data attributes*
* console.log(logo.dataset.versionNumber);

<img

*src*="img/logo.png"

*alt*="Bankist logo"

*class*="nav\_\_logo"

*id*="logo"

*designer* = "Jonas"

*data-version-number*="3.0"

        />

* Classes
* *// Classes*
* logo.classList.add('c');
* logo.classList.remove('c');
* logo.classList.toggle('c');
* logo.classList.contains('c');

Smooth scrolling

Determine initial coordinates

.getBoundingClientRect()

console.log('Current scroll (X/Y)', window.pageXOffset, window.pageYOffset);

  console.log('Height/width viewport', document.documentElement.clientHeight, document.documentElement.clientWidth)

Incomplete scroll – does not work if current scroll location is not at top of page

*//Scrolling*

  window.scrollTo(s1coords.left, s1coords.top);

window.scrollTo(

    s1coords.left + window.pageXOffset,

    s1coords.top + window.pageYOffset

  );

Most modern:

window.scrollTo({

    left: s1coords.left + window.pageXOffset,

    top: s1coords.top + window.pageYOffset,

    behaviour: 'smooth',

  });

  section1.scrollIntoView({behavior: 'smooth'});

Events and event handler

* An event is a signal generated by a certain DOM node after an action is taken
* addEventListener vs .onmouseenter
* *const* h1 = document.querySelector('h1');
* h1.addEventListener('mouseenter', *function*(*e*){
* alert('addEventListener: great! You are reading the heading')
* });
* h1.onmouseenter = *function*(*e*){
* alert('addEventListener: great! You are reading the heading')
* };
* .removeEventListener()
  + setTimeout(function)

15 October 2021

Event propagation – bubbling and capturing

Graphical user interface

Description automatically generated

* Capturing phase -> Target phase -> Bubbling phase

Diagram

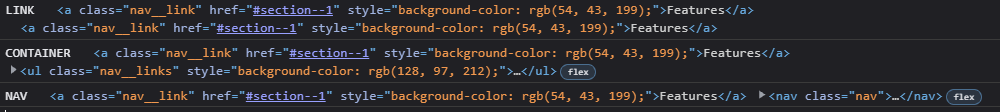
Description automatically generated

* An event travels down through the parent elements to reach its target and then bubbles back up through all the parents again back to the document
* *const* randomInt = (*min*, *max*) => Math.floor(Math.random() \* (max-min+1) + min);
* *const* randomColor = () => `rgb(${randomInt (0,255)}, ${randomInt (0,255)}, ${randomInt (0,255)})`;
* console.log(randomColor());
* document.querySelector('.nav\_\_link').addEventListener('click', *function*(*e*){
* *this*.style.backgroundColor = randomColor();
* })
* document.querySelector('.nav\_\_links').addEventListener('click', *function*(*e*){
* *this*.style.backgroundColor = randomColor();
* })
* document.querySelector('.nav').addEventListener('click', *function*(*e*){
* })

A picture containing text, screenshot, indoor

Description automatically generated

* .nav\_\_link -> the event listener triggers a background colour change for both the child and parent, as the event bubbles up. Cliking on .nav\_\_links parent only does not affect the child
* E.target and e.currentTarget
  + currentTarget – element that the event handler is attached
  + Target – where the event originally happened



* E.stopPropogation() – to stop bubbling, but generally not good practice
* *// Stop propogation*
* e.stopPropagation();
* Listening to event during Capturing Phase vs Bubbling Phase – but Capturing rarely used in modern JS
* document.querySelector('.nav').addEventListener('click', *function*(*e*){
* *this*.style.backgroundColor = randomColor();
* console.log('NAV', e.target, e.currentTarget);
* }, true)

Event delegation – implementing page delegation

Smooth scroll navigation

*// Navigation*

btnScrollTo.addEventListener('click', *function* (*e*) {

  section1.scrollIntoView({ behavior: 'smooth' });

});

document.querySelectorAll('.nav\_\_link').forEach(*el* => el.addEventListener('click', *function*(*e*){

  e.preventDefault();

  console.log('LINK');

*const* id = *this*.getAttribute('href');

  console.log(id);

  document.querySelector(id).scrollIntoView({behavior: 'smooth'});

}))

Implementing event delegation – delegating the eventListener to the parent element as opposed to individually to all 3 child elements

document.querySelector('.nav\_\_links').addEventListener('click', *function*(*e*){

  console.log(e.target);

    e.preventDefault();

  if(e.target.classList.contains('nav\_\_link')){

    console.log('LINK');

*const* id = e.target.getAttribute('href');

    console.log(id);

    document.querySelector(id).scrollIntoView({behavior: 'smooth'});

  }

})

DOM traversing

Downwards

* .childNodes
* .children
* .firstElementChild
* .lastElementChild

Upwards

* .parentNode
* .parentElement
* .closest - looking beyond immediate parent elements

h1.closest('.header').style.background = 'var(--gradient-secondary)';

Siblings / sideways

* .previousElementSibling
* .nextElementSibling
* .previousSibling
* .nextSibling
* Referring up to parent then collecting ALL child elements
* console.log(h1.parentElement.children);
* [...h1.parentElement.children].forEach(*function*(*el*){
* if(el !== h1){
* el.style.transform = 'scale(0.5)';
* }
* });

Building a tabbed component

tabsContainer.addEventListener('click', *function*(*e*){

  e.preventDefault();

*const* clicked = e.target.closest('.operations\_\_tab');

  console.log(clicked);

*//Guard clause - preventing 'null' being returned when clicking on tabsContainer*

  if(!clicked) return;

*//Remove active classes*

  tabs.forEach(*t* => t.classList.remove('operations\_\_tab--active'));

  tabsContent.forEach(*t* => t.classList.remove('operations\_\_content--active'));

*//Activate tab*

  clicked.classList.add('operations\_\_tab--active');

*//Activate content area*

*const* data = clicked.getAttribute('data-tab');

  console.log(data);

*const* data1 = clicked.dataset.tab;

  console.log(data1);

  document.querySelector(`.operations\_\_content--${data}`).classList.add('operations\_\_content--active');

});

Passing arguments to event handler functions

* Mouseover
* Mouseout
* Mouseleave

*//Nav - menu fade animation*

*const* handleHover = *function*(*e*, *opacity*){

  if(e.target.classList.contains('nav\_\_link')){

*const* link = e.target;

*const* siblings = link.closest('.nav').querySelectorAll('.nav\_\_link');

*const* logo = link.closest('.nav').querySelector('img');

    siblings.forEach(*el* => {

      if(el !== link){

        el.style.opacity = opacity;

        logo.style.opacity = opacity;

      }

    })

  }

}

nav.addEventListener('mouseover', *function*(*e*){

  handleHover(e, 0.5);

});

nav.addEventListener('mouseout', *function*(*e*){

  handleHover(e, 1.0);

});

*//Nav - menu fade animation with .bind*

*const* handleHover = *function*(*e*){

  if(e.target.classList.contains('nav\_\_link')){

*const* link = e.target;

*const* siblings = link.closest('.nav').querySelectorAll('.nav\_\_link');

*const* logo = link.closest('.nav').querySelector('img');

    siblings.forEach(*el* => {

      if(el !== link){

        el.style.opacity = *this*;

        logo.style.opacity = *this*;

      }

    })

  }

}

nav.addEventListener('mouseover', handleHover.bind(0.5));

nav.addEventListener('mouseout', handleHover.bind(1.0));

Sticky nav

Old school – not good bc of scroll eventListener constantly firing

*// Sticky navigation*

*const* initialCoords = section1.getBoundingClientRect();

console.log(initialCoords);

window.addEventListener('scroll', *function*(*e*){

  console.log(window.scrollY);

  if(window.scrollY > initialCoords.top){

    nav.classList.add('sticky');

  } else {

    nav.classList.remove('sticky');

  }

});

Intersection Observer API – sticky navigation alternative

*const* header = document.querySelector('.header');

*const* navHeight = nav.getBoundingClientRect().height;

*const* stickyNav = *function* (*entries*) {

*const* [entry] = entries;

  console.log(entry);

  if (!entry.isIntersecting) nav.classList.add('sticky');

  else nav.classList.remove('sticky');

};

*const* headerObserver = new IntersectionObserver(stickyNav, {

  root: null,

  threshold: 0,

  rootMargin: `-${navHeight}px`,

});

headerObserver.observe(header);

Revealing elements on scroll

*//Reveal sections*

*const* allSections = document.querySelectorAll('.section');

*const* revealSection = *function*(*entries*, *observer*){

*const* [entry] = entries;

  console.log(entry);

  if(!entry.isIntersecting) return;

  entry.target.classList.remove('section--hidden');

  observer.unobserve(entry.target); //Note unobserved for performance op.

};

*const* sectionObserver = new IntersectionObserver(revealSection, {

  root: null,

  threshold: 0.15,

});

allSections.forEach(*function*(*section*){

  sectionObserver.observe(section);

  section.classList.add('section--hidden');

})

Lazy loading images

*// Lazy loading images*

*const* imgTargets = document.querySelectorAll('img[data-src]');

console.log(imgTargets);

*const* loadImg = *function* (*entries*, *observer*){

*const* [entry] = entries;

  console.log(entry);

  if(!entry.isIntersecting) return;

*// Replace src with data-src*

  entry.target.src = entry.target.dataset.src;

*// entry.target.classList.remove('lazy-img')    // Bad practice to jump the gun and removing the filter before the image finishes loading*

  entry.target.addEventListener('load',*function*(){

    entry.target.classList.remove('lazy-img')

  });

  observer.unobserve(entry.target);

};

*const* imgObserver = new IntersectionObserver(loadImg,{

  root: null,

  threshold: 0,

  rootMargin: '200px',

});

imgTargets.forEach(*img* => imgObserver.observe(img));

16 October 2021

Building a slider component

*//Slider*

*const* slides = document.querySelectorAll('.slide');

*const* slider = document.querySelector('.slider');

*const* btnLeft = document.querySelector('.slider\_\_btn--left');

*const* btnRight = document.querySelector('.slider\_\_btn--right');

*let* curSlide = 0;

*const* maxSlide = slides.length;

*// slider.style.overflow = 'visible';*

*const* goToSlide = *function*(*slide*){

  slides.forEach((*s*, *i*) => s.style.transform = `translateX(${100 \* (i - slide)}%)`);

};

*// slides.forEach((s, i) => s.style.transform = `translateX(${100 \* i}%)`);*

goToSlide(0);

*const* nextSlide = *function*(){

  if(curSlide === maxSlide - 1){

    curSlide = 0;

  } else {

  curSlide++;

  }

  goToSlide(curSlide);

}

*const* prevSlide = *function*(){

  if(curSlide === 0){

    curSlide = maxSlide - 1;

  } else {

  curSlide--;

  }

  goToSlide(curSlide);

}

btnRight.addEventListener('click', nextSlide);

btnLeft.addEventListener('click', prevSlide)

Dot navigation on slider

*const* init = *function*(){

*// slides.forEach((s, i) => s.style.transform = `translateX(${100 \* i}%)`);*

  goToSlide(0);

  createDots();

  activateDot(curSlide);

}

init();

btnRight.addEventListener('click', nextSlide);

btnLeft.addEventListener('click', prevSlide);

*//Keyboard actions to move slider*

document.addEventListener('keydown', *function* (*e*) {

  if (e.key === 'ArrowLeft') prevSlide();

  else if (e.key === 'ArrowRight') nextSlide();

*//Shortcircuiting*

*// e.key === 'ArrowLeft' && prevSlide();*

});

*//Adding the dots*

dotContainer.addEventListener('click', *function* (*e*) {

  if (e.target.classList.contains('dots\_\_dot')) {

*// const slide = e.target.dataset.slide;*

*const* { slide } = e.target.dataset;

    goToSlide(slide);

    activateDot(slide);

  }

});

Lifecycle DOM events

* Loaded DOM content without waiting for resources to finish

document.addEventListener('DOMContentLoaded', *function*(*e*){

  console.log('HTML parsed and DOM tree built!');

});

* Complete page is loaded
* document.addEventListener('load', *function*(*e*){
* console.log('Page fully laoded', e);
* });
* Before users leave the page
* window.addEventListener('beforeunload', *function*(*e*){
* e.preventDefault();
* console.log(e);
* e.returnValue = 'message';
* });

Efficient script loading in HTML

* Traditional - <script src = “script.js”>
* Async and defer additions
* Defer is best practice – parsing the HTML and fetching the script and running the script before DOMContentLoaded is released
  + Scripts are executed in order as opposed to async
* Old browsers don’t support async and defer
* Use async for 3rd party scripts where order doesn’t matter

Graphical user interface, website

Description automatically generated

Graphical user interface, website

Description automatically generated

21 October 2021

Asynchronous Javascript / dealing with APIs

*const* renderCountry = *function*(*data*, *className* =''){

*const* html = `

  <article class="country ${className}">

  <img class="country\_\_img" src="${Object.values(data.flags)[1]}" />

  <div class="country\_\_data">

    <h3 class="country\_\_name">${data.name.common}</h3>

    <h4 class="country\_\_region">${data.region}</h4>

    <p class="country\_\_row"><span>👫</span>${(+data.population/1000000).toFixed(1)}M</p>

    <p class="country\_\_row"><span>🗣️</span>${Object.values(data.languages)}</p>

    <p class="country\_\_row"><span>💰</span>${Object.values(data.currencies)[0].name}</p>

  </div>

  </article>`;

  countriesContainer.insertAdjacentHTML('beforeend', html);

  countriesContainer.style.opacity = 1;

};

*const* getCountryAndNeighbour = *function*(*country*){

*//AJAX call country 1*

*const* request = new XMLHttpRequest();

request.open('GET', `https://restcountries.com/v3.1/name/${country}`);

request.send();

request.addEventListener('load', *function*(){

*// console.log(this.responseText);*

*const* [data] = JSON.parse(*this*.responseText);

    console.log(data);

*//Render country 1*

    renderCountry(data);

*//Get neighbour country*

*const* neighbour = data.borders[0];

    console.log(neighbour);

    if(!neighbour) return;

*//AJAX call country 2*

*const* request2 = new XMLHttpRequest();

    request2.open('GET', `https://restcountries.com/v3.1/alpha/${neighbour}`);

    request2.send();

    request2.addEventListener('load', *function*(){

*const* [data2] = JSON.parse(*this*.responseText);

      console.log(data2);

*//Render country 1*

    renderCountry(data2, 'neighbour');

    });

});

};

getCountryAndNeighbour('Albania');

Callback hell

* It works but is bad practice – difficult to read / understand

Promises

* An object used as a placeholder for the future result of an asynchronous operation
  + OR – a container for an asynchronously delivered value
    - OR – a container for a future value
* Why use promises?
  + No longer rely on events/call backs passed into asynchronous functions to handle asynchronous results
  + We can instead chain promises for a sequence of asynchronous operations
* Promise lifecycle
  + Pending – before future value is available
  + Settled (fulfilled / rejected) – asynchronous task finished
  + Fetch API – builds the promise that then gets consumed
* Using promises
* *const* getCountryData = *function* (*country*) {
* fetch(`https://restcountries.com/v3.1/name/${country}`)
* .then(*response* => response.json())
* .then(*data* => renderCountry(data[0]));
* };
* getCountryData('Australia');

22 October 2021

* Chaining promises with fetch
* *const* getCountryData = *function* (*country*) {
* fetch(`https://restcountries.com/v3.1/name/${country}`)
* .then(*response* => response.json())
* .then(*data* => {
* console.log(data);
* renderCountry(data[0]);
* *const* neighbour = data[0].borders[0];
* if (!neighbour) return;
* return fetch(`https://restcountries.com/v3.1/alpha/${neighbour}`);
* })
* .then(*response* => response.json())
* .then(*data* => renderCountry(data[0], 'neighbour'));
* };
* getCountryData('Albania');
* Handling rejected promises - .catch
  + Handling an error (console feedback) by catching the error
* then(*data* => renderCountry(data[0], 'neighbour'))
* .catch(*err* => {
* console.error(`${err}`);
* renderError(`Something went wrong ${err.message}. Try again!`);
* }).finally(() => {
* });
* Displaying fetch errors to user interface
* *const* renderError = *function* (*msg*) {
* countriesContainer.insertAdjacentText('beforeend', msg);
* countriesContainer.style.opacity = 1;
* };
* Throwing errors manually

A screenshot of a computer

Description automatically generated

.then(

*response* => {

        console.log(response);

        if(!response.ok)

          throw new Error(`Country not found (${response.status})`);

        return response.json()

      })

* Manually catching errors - The throw new Error message will reject the promise and propagate all the way through to the .catch – good practice to inform users of more specific details of errors

*const* getCountryData = *function* (*country*) {

  getJSON(`https://restcountries.com/v3.1/name/${country}`, 'Country not found')

    .then(*data* => {

      console.log(data);

      renderCountry(data[0]);

*const* neighbour = data[0].borders;

      console.log(neighbour);

      if (!neighbour) throw new Error(`No neighbour found!`);

      return getJSON(`https://restcountries.com/v3.1/alpha/${neighbour}`, 'Country not found');

    })

    .then(*data* => renderCountry(data[0], 'neighbour'))

    .catch(*err* => {

      console.error(`${err}`);

      renderError(`Something went wrong(${err.message}). Try again!`);

    }).finally(() => {

      countriesContainer.style.opacity = 1;

    });

};

btn.addEventListener('click', *function* () {

  getCountryData('Australia');

});

Javascript runtime refresh

* Runtime in browser
* JS engine – heap / call stack
* WEB APIs
* Callback queue – ‘to do list’
  + The Event loop tick – the looks into call stack if empty or not, if it is, then it takes the first callback from the callback queue and puts it into the call stack

Graphical user interface, website

Description automatically generated

* Everything related to the DOM are not within JS itself but instead the web APIs – which allows asynchronous tasks to be possible
  + E.g .querySelector(‘.img’)
* How can asynchronous code be executed in a non blocking way if there is only one thread of execution in the engine?
  + WEB APIs | callback queue / event loop | call stack
  + Promises -> What happens after the results are ready and a promise is fulfilled?
    - Microtasks queue – has priority over callback queue
    - The downside – too many microtasks can starve the callback queue

Graphical user interface, text, website

Description automatically generated

The Event loop in practice

console.log('Test start');

setTimeout(() => console.log('0 sec timer'),0);

Promise.resolve('Resolved promise 1').then(*res* => console.log(res));

console.log('Test end');

Q – in what order will the following messages be generated?

1. Test start
2. Test end
3. Resolved promise
4. Set timeout

Building a simple promise

*const* lotteryPromise = new Promise(*function* (*resolve*, *reject*) {

  console.log('Lottery draw is happening');

  setTimeout(*function*(){

    if (Math.random() >= 0.5) {

      resolve('You WIN!');

    } else {

      reject(new Error('You have lost.'));

    }

  }, 2000);

});

lotteryPromise.then(*res* => console.log(res)).catch(*err* => console.error(err));

Promisifying – convert callback based async behaviour into Promise based

*const* lotteryPromise = new Promise(*function* (*resolve*, *reject*) {

  console.log('Lottery draw is happening');

  setTimeout(*function* () {

    if (Math.random() >= 0.5) {

      resolve('You WIN!');

    } else {

      reject(new Error('You have lost.'));

    }

  }, 2000);

});

lotteryPromise.then(*res* => console.log(res)).catch(*err* => console.error(err));

*const* wait = *function* (*seconds*) {

  return new Promise(*function* (*resolve*) {

    setTimeout(resolve, seconds \* 1000);

  });

};

wait(2)

  .then(() => {

    console.log('I waited for 1 seconds');

    return wait(1);

  }).then(() => {

    console.log('I waited for 2 seconds');

    return wait(1);

  }).then(() => {

    console.log('I waited for 3 seconds');

    return wait(1);

  }).then(() => {

    console.log('I waited for 4 seconds');

    return wait(1);

  })

  .then(() => console.log('I waited for 5 second'));

  Promise.resolve('abc').then(*x* => console.log(x));

  Promise.reject(new Error('rej')).catch(*err* => console.log(err));

Text

Description automatically generated

Promisifying the geolocation API

A callback based API:

navigator.geolocation.getCurrentPosition(

*position* => console.log(position),

*err* => console.error(err)

);

console.log('Geting position');

A promise based API:

*console.log('Geting position');*

*const getPosition = function(){*

*return new Promise(function(resolve, reject){*

*// navigator.geolocation.getCurrentPosition(*

*//   position => resolve(position),*

*//   err => reject(err),*

*// );*

*navigator.geolocation.getCurrentPosition(resolve, reject);*

*})*

*}*

*getPosition().then(res => console.log(res));*

*const whereAmI = function () {*

*getPosition().then(pos => {*

*const lat = pos.coords.latitude;*

*const lng = pos.coords.longitude;*

*// const {latitude: lat, longitude: lng } = pos.coords;   // Much prettier code*

*return fetch(`https://geocode.xyz/${lat},${lng}?geoit=json`)*

*})*

*.then(response => {*

*console.log(response);*

*if (!response.ok)*

*throw new Error(`Too many requests (${response.status})`);*

*return response.json();*

*})*

*.then(data => {*

*console.log(`You are in ${data.city}, ${data.country}`);*

*getCountryData(`${data.country}`);*

*})*

*.catch(err => {*

*console.error(`${err}`);*

*renderError(`${err.message}`);*

*});*

*};*

*whereAmI();*

Coding challenge #2

*// Coding challenge 2*

*const* wait = *function* (*seconds*) {

  return new Promise(*function* (*resolve*) {

    setTimeout(resolve, seconds \* 1000);

  });

};

*const* image = document.querySelector('.images');

*const* createImage = *function* (*imgPath*) {

  return new Promise(*function* (*resolve*, *reject*) {

*const* img = document.createElement('img');

    img.src = imgPath;

    img.addEventListener('load', *function* () {

      image.append(img);

      resolve(img);

    });

    wait(2);

    img.addEventListener('error', *function* () {

      reject(new Error('Image not found'));

    });

  });

};

*let* currentImg;

createImage('img/img-1.jpg')

  .then(*img* => {

    currentImg = img;

    console.log(img);

    return wait(2);

  })

  .then(() => {

    currentImg.style.display = 'none';

    return createImage('img/img-2.jpg');

  })

  .then(*img2* => {

      currentImg = img2;

      console.log(img2);

      return wait(2);

  }).then(() => {

    currentImg.style.display = 'none';

    return createImage('img/img-3.jpg');

  })

  .then(*img3* => {

      currentImg = img3;

      console.log(img3);

      return wait(2);

  })

  .catch(*err* => {

    console.error(`${err}`);

  });

23/11/2021

Async/await function – syntactic sugar over the then method in Promises

* Important to understand the underlying mechanism before jumping ahead and using the algorithm
* *const* whereAmI = *async* *function* () {
* *//Geolocation*
* *const* pos = await getPosition();
* console.log(pos);
* *const* { latitude: lat, longitude: lng } = pos.coords;
* *//Reverse Geocoding*
* *const* resGeo = await fetch(`https://geocode.xyz/${lat},${lng}?geoit=json`);
* console.log(resGeo);
* *const* dataGeo = await resGeo.json();
* console.log(dataGeo);
* *//Country data*
* *const* res = await fetch(
* `https://restcountries.com/v3.1/name/${dataGeo.country}`
* );
* console.log(res);
* *const* data = await res.json();
* console.log(data);
* renderCountry(data[0]);
* };
* whereAmI();
* console.log('first');

Error handling with try.. catch

*const* whereAmI = *async* *function* () {

  try{

*//Geolocation*

*const* pos = await getPosition();

  console.log(pos);

*const* { latitude: lat, longitude: lng } = pos.coords;

*//Reverse Geocoding*

*const* resGeo = await fetch(`https://geocode.xyz/${lat},${lng}?geoit=json`);

  if(!resGeo.ok) throw new Error('Problem getting location data');

  console.log(resGeo);

*const* dataGeo = await resGeo.json();

  console.log(dataGeo);

*//Country data*

*const* res = await fetch(

    `https://restcountries.com/v3.1/name/${dataGeo.country}`

  );

  if(!res.ok) throw new Error('Problem getting country');

  console.log(res);

*const* data = await res.json();

  console.log(data);

  renderCountry(data[0]);

  countriesContainer.style.opacity = 1;

}catch (err){

  console.log(`${err}`);

  countriesContainer.insertAdjacentText('beforeend', `${err.message}`);

  countriesContainer.style.opacity = 1;

}

};

whereAmI();

console.log('first');

Returning values from async functions

* Note – an async function always returns a Promise
* *//Mix between new async/await and the old Promises/Then*
* *// whereAmI()*
* *//   .then(city => console.log(city))*
* *//   .catch(err => console.log(`{err.message}`))*
* *//   .finally(() => console.log('3: Finished getting location'));*
* *// Pure async/await with try/catch*
* (*async* *function*(){
* try{
* *const* city = await whereAmI();
* console.log(city);
* } catch (err){
* console.log(`${err.message}`)
* }
* console.log('3: Finished getting location');
* })();
* Note the use of IIFE – Immediately Invoked Function Expression

Running promises in parallel vs sequence

*const* get3Countries = *async* *function*(*c1*, *c2*, *c3*){

  try{

*const* [data1] = await getJSON(`https://restcountries.com/v3.1/name/${c1}`);

    console.log(data1);

*const* [data2] = await getJSON(`https://restcountries.com/v3.1/name/${c2}`);

*const* [data3] = await getJSON(`https://restcountries.com/v3.1/name/${c3}`);

    console.log([...data1.capital, ...data2.capital, ...data3.capital]);

  }catch(err){

    console.error(err);

  }

}

get3Countries('brazil', 'canada', 'tanzania');

* In this current structure, data 2 would wait for data 1 to load. Data 3 for data 2 to load.

Graphical user interface, text

Description automatically generated

Running in parallel with Promise.all (combinator function)

*const* get3Countries = *async* *function* (*c1*, *c2*, *c3*) {

  try {

*// const [data1] = await getJSON(`https://restcountries.com/v3.1/name/${c1}`);*

*// console.log(data1);*

*// const [data2] = await getJSON(`https://restcountries.com/v3.1/name/${c2}`);*

*// const [data3] = await getJSON(`https://restcountries.com/v3.1/name/${c3}`);*

*const* data = await Promise.all([

      getJSON(`https://restcountries.com/v3.1/name/${c1}`),

      getJSON(`https://restcountries.com/v3.1/name/${c2}`),

      getJSON(`https://restcountries.com/v3.1/name/${c3}`),

    ]);

*// console.log([...data1.capital, ...data2.capital, ...data3.capital]);*

    console.log(data.map(*d* => d[0].capital[0]));

  } catch (err) {

    console.error(err);

  }

};

get3Countries('brazil', 'canada', 'tanzania');

Other Promise Combinators – Promise.race, allSettled and any

* Promise.race is important for slow internet connections
* *const* timeout = *function* (*sec*) {
* return new Promise(*function* (*\_*, *reject*) {
* setTimeout(*function* () {
* reject(new Error('Request took too long!'));
* }, sec \* 1000);
* });
* };
* Promise.race([getJSON(`https://restcountries.com/v3.1/name/italy`), timeout(1)])
* .then(*res* => console.log(res[0]))
* .catch(*err* => console.error(err));

Promise.allSettled vs Promise.all

Promise.allSettled([

  Promise.resolve('Success'),

  Promise.reject('Error'),

  Promise.resolve('Another success'),

]).then(*res* => console.log(res));

Promise.all([

  Promise.resolve('Success'),

  Promise.reject('Error'),

  Promise.resolve('Another success'),

])

  .then(*res* => console.log(res))

  .catch(*err* => console.error(err));

Promise.any

  Promise.any([

    Promise.resolve('Success'),

    Promise.reject('Error'),

    Promise.resolve('Another success'),

  ])

    .then(*res* => console.log(res))

    .catch(*err* => console.error(err));

Coding challenge #3

*const* image = document.querySelector('.images');

*const* createImage = *function* (*imgPath*) {

  return new Promise(*function* (*resolve*, *reject*) {

*const* img = document.createElement('img');

    img.src = imgPath;

    img.addEventListener('load', *function* () {

      image.append(img);

      resolve(img);

    });

    img.addEventListener('error', *function* () {

      reject(new Error('Image not found'));

    });

  });

};

*const* wait = *function* (*seconds*) {

  return new Promise(*function* (*resolve*) {

    setTimeout(resolve, seconds \* 1000);

  });

};

*let* currentImg;

*const* loadNPause = *async* *function* () {

  try {

*let* img = await createImage('img/img-1.jpg');

    await wait(2);

    img.style.display = 'none';

    img = await createImage('img/img-2.jpg');

    await wait(2);

    img.style.display = 'none';

    img = await createImage('img/img-3.jpg');

    await wait(2);

    img.style.display = 'none';

  } catch (err) {

    console.error(`${err}`);

  }

};

*// loadNPause();*

*const* imgArr = ['img/img-1.jpg', 'img/img-2.jpg', 'img/img-3.jpg'];

*const* loadAll = *async* *function*(*arr*){

  try{

*const* imgs = arr.map(*imgSrc* => createImage(imgSrc));

  console.log(imgs);

*const* data = await Promise.all(imgs);

  console.log(data);

  data.forEach(*img* => img.classList.add('parallel'));

  } catch (err){

    console.error(err);

  }

};

loadAll(imgArr);

2 December 2021

Object Oriented Programming

* What is OOP?
  + A programming paradigm based on the concept of objects
  + We use objects to model real world or abstract features
  + Object may contain data (properties) and code (methods)
  + Objects are self-contained pieces/blocks of code, which are then the building blocks of applications
* Classic OOP
  + Class – can be understood as a ‘blueprint’
  + Objects (instances) are instantiated from a class
* 4 pillars of OOP
  + Abstraction – ignoring or hiding details that don’t matter, in order to get the overview perspective of the thing being implemented
  + Encapsulation – keeping methods and properties private within the class so they are not accessible from outside the class
  + Inheritance – making all properties/methods of a certain class available to a child class, allowing for reuse of common logic
  + Polymorphism – a child class can overwrite a method inherited from the parent class
* OOP in Javascript – prototypes
  + All objects are linked to a ‘prototype’
  + The prototypes contains the methods
    - Prototypal inheritance – the prototype contains methods that are accessible to all objects linked to that prototype
  + Behaviour is delegated to the linked prototype object
* Implementing prototypal inheritance
  + Constructor functions – create objects from a function (arrays, maps, sets)
  + ES6 classes –
  + Object.create()
* Constructor function example
* *//Building constructor functions*
* *const* Person = *function* (*firstName*, *birthYear*) {
* *this*.firstName = firstName;
* *this*.birthYear = birthYear;
* };
* *const* jonas = new Person('Jonas', 1991);
* console.log(jonas);
* *//What does the new operator do?*
* *// 1. New object {} is created*
* *// 2. Function is called, this = {}*
* *// 3. {} linked to prototype*
* *// 4. function automatically return {}*
* *const* matilda = new Person('Matilda', 2017);
* *const* jack = new Person('Jack', 1975);
* console.log(matilda, jack);
* instanceof – to check whether an instant belongs to a class
* console.log(jonas instanceof Person);
* Prototypes / prototypal inheritance – each function has a prototype, incl. constructor functions
  + Person.prototype.calcAge = function(){};
* \_\_proto\_\_ / isPrototypeOf
* console.log(jonas.\_\_proto\_\_);
* 
* console.log(Person.prototype.isPrototypeOf(jonas));
* The prototype chain
  + Person.prototype is an object in itself
    - \_\_proto\_\_ -> Object.prototype
* 
* hasOwnProperty – can navigate up the prototype chain

ES6 classes

* An improved syntax but still uses Prototype mechanics underneath
* *class* PersonCl {
* *constructor*(*firstName*, *birthYear*){
* *this*.firstName = firstName;
* *this*.birthYear = birthYear;
* }
* *//Methods added directly to .prototype property*
* calcAge(){
* console.log(2037 - *this*.birthYear);
* }
* }
* *const* jessica = new PersonCl('Jessica', 1996);
* console.log(jessica.\_\_proto\_\_ === PersonCl.prototype);
* PersonCl.prototype.greet = *function*(){
* console.log(`Hey ${*this*.firstName}`);
* };
* jessica.greet();
* Classes are NOT hoisted
* Classes are first-class citizens
* Classes are executed in strict mode

Setters and getters

* A
* *const account = {*
* *owner:'jonas',*
* *movements: [200, 530, 120, 300],*
* *get latest(){*
* *return this.movements.slice(-1).pop();*
* *},*
* *set latest(mov){*
* *this.movements.push(mov);*
* *},*
* *}*
* *console.log(account.latest);*
* *// Notice how the 'set' method is utilised*
* *account.latest = 50;*
* *console.log(account.movements);*

Static methods – static only available within class

* Array.from method – attached to the array constructor
* *const* Person = *function* (*firstName*, *birthYear*) {
* *//Instance properties*
* *this*.firstName = firstName;
* *this*.birthYear = birthYear;
* };
* Person.hey = *function*(){console.log('Hey there!')};
* *const* jonas = new Person('Jonas', 1991);
* Person.hey();
* *//The below doesn't work bc the method exists attached to the constructor itself*
* jonas.hey();
* Object.create – to create custom objects to assign as prototypes

*const* PersonProto = {

    calcAge(){

        console.log(2037 - *this*.birthYear);

    },

    initialise(*firstName*, *birthYear*){

*this*.firstName = firstName;

*this*.birthYear = birthYear;

    },

}

*const* steven = Object.create(PersonProto);

console.log(steven);

steven.name = 'Steven';

steven.birthYear = 2002;

steven.calcAge();

console.log(steven.\_\_proto\_\_);

*const* sarah = Object.create(PersonProto);

sarah.initialise('Sarah', 1979);

console.log(sarah);

Coding challenge #2

*//Coding challenge #2*

*class* CarCl{

*constructor*(*make*, *speed*){

*this*.make = make;

*this*.speed = speed;

    }

    accelerate(){

        console.log(*this*.speed += 10);

    }

    brake(){

        console.log(*this*.speed -= 10);

    }

*get* speedUS(){

        return *this*.speed / 1.6;

    }

*set* speedUS(*speed*){

*this*.speed = speed \* 1.6;

    }

}

*const* BMW = new CarCl('BMW', 120);

BMW.accelerate();

BMW.accelerate();

BMW.accelerate();

BMW.accelerate();

console.dir(CarCl);

*//Utilising the getter method*

console.log(BMW.speedUS);

*//Utilising the setter method*

BMW.speedUS = 50;

4 December 2021

Inheritance between classes

* Developing a child ‘student’ underneath a generic ‘person’ class

Diagram

Description automatically generated

*const* Person = *function* (*firstName*, *birthYear*) {

*//Instance properties*

*this*.firstName = firstName;

*this*.birthYear = birthYear;

};

Person.prototype.calcAge = *function*(*birthYear*){

    console.log(2037 - birthYear);

};

*const* Student = *function*(*firstName*, *birthYear*, *course*){

*this*.firstName = firstName;

*this*.birthYear = birthYear;

*this*.course = course;

};

Student.prototype.introduce = *function*(){

    console.log(`My name is ${*this*.firstName} and I study ${*this*.course}.`);

}

*const* mike = new Student('Mike', 2020, 'Computer Science');

console.log(mike);

mike.introduce();

* How to remove duplicate code of Person/Student overlap

Note the use of the .call method

*const* Student = *function*(*firstName*, *birthYear*, *course*){

    Person.call(*this*, firstName, birthYear)

*this*.course = course;

};

Text

Description automatically generated

* Linking the child Student to Parent prototype
* *//Linking prototypes*
* Student.prototype = Object.create(Person.prototype);
* Instantof
* mike.calcAge();
* console.log(mike instanceof Student);
* console.log(mike instanceof Person);
* Student.prototype.constructor = Student;
* console.dir(Student.prototype.constructor);
* When child / parent prototype are of the same name – child takes priority in the chain, which makes sense in terms of ‘searching up’ the chain for what it needs

A screenshot of a computer

Description automatically generated with medium confidence

Inheritance between ‘classes’ – ES6 classes

* Extends keyword – creates the link from child to parent, like Object.create
* Super keyword – allows parent to be used within the child, like XX.call(this, Z, A)
* Necessary to use with extends
* *class* StudentCl *extends* *PersonCl* {}

*class* StudentCl *extends* *PersonCl* {

*constructor*(*fullName*, *birthYear*, *course*){

*//Link to parent class needs to happen first*

*super*(fullName, birthYear);

*this*.course = course;

    };

    introduce(){

        console.log(`My name is ${*this*.fullName} and I study ${*this*.course}.`);

    }

    calcAge(){

        console.log(`I'm ${2037 - *this*.birthYear} years old, but as a student I feel more like ${2037 - *this*.birthYear + 10}`);

    }

}

*const* martha = new StudentCl('Martha Jones', 2012, 'Computer Science');

console.log(martha);

martha.introduce();

martha.calcAge();

* Creating inheritance through Object.create()
* *const* PersonProto = {
* calcAge(){
* console.log(2037 - *this*.birthYear);
* },
* initialise(*firstName*, *birthYear*){
* *this*.firstName = firstName;
* *this*.birthYear = birthYear;
* },
* }
* *const* steven = Object.create(PersonProto);
* *const* StudentProto = Object.create(PersonProto);
* StudentProto.init = *function*(*firstName*, *birthYear*, *course*){
* PersonProto.initialise.call(*this*, firstName, birthYear);
* *this*.course = course;
* }
* StudentProto.introduce = *function*(){
* console.log(`My name is ${*this*.firstName} and I study ${*this*.course}.`);
* }
* *const* jay = Object.create(StudentProto);
* jay.init('Jay', 2010, 'Computer science');
* jay.introduce();
* jay.calcAge();

Encapsulation protected properties and methods

* Why? To save methods away from public access
  + Protection adding ‘\_’ to the front of fields
* *class* Account {
* *constructor*(*owner*, *currency*, *pin*) {
* *this*.owner = owner;
* *this*.currency = currency;
* *//Protected property*
* *this*.\_pin = pin;
* *this*.\_movements = [];
* *this*.locale = navigator.language;
* console.log(`Thanks for opening an account, ${owner}.`);
* }
* Protected vs private
  + Private class fields and methods
    - Public fields
    - Private fields
* *//Private fields*
* #movements = [];



* + - Public methods
    - Private methods
* *//   Private methods*
* #approveLoan(*val*) {
* return true;
* }
* }
  + What is a field – property on all instances

13 December 2021

Chaining methods

*//Chaining methods*

acc1.deposit(300).deposit(500).withdraw(35).requestLoan(25000).withdraw(4000);

console.log(acc1.getMovements());

ES6 classes summary

* Parent/child classes
  + Extends keyword – establishing inheritance between classes
  + Public / private / static public fields
* Super keyword – call to parent
* Referencing private field and method
* Getter method – get value out of object by writing a property rather than method

Text

Description automatically generated

Coding challenge #4

*//Coding challenge #4*

*class* CarCl{

*constructor*(*make*, *speed*){

*this*.make = make;

*this*.speed = speed;

    }

    accelerate(){

        console.log(*this*.speed += 10);

        return *this*;

    }

    brake(){

        console.log(*this*.speed -= 10);

        return *this*;

    }

*get* speedUS(){

        return *this*.speed / 1.6;

    }

*set* speedUS(*speed*){

*this*.speed = speed \* 1.6;

    }

}

*class* EVCl *extends* *CarCl*{

    #charge;

*constructor*(*make*, *speed*, *charge*){

*super*(make, speed);

*this*.#charge = charge;

    };

    chargeBattery(*chargeTo*){

*this*.#charge = chargeTo;

        return *this*;

    }

}

*const* car1 = new EVCl('Rivian', 120, 23);

console.log(car1);

car1.accelerate().chargeBattery(42).brake().brake();

console.log(car1);

console.log(car1.speedUS);

car1.speedUS = 68.75;

console.log(car1)

Mapty Application – OOP, geolocation, external libraries

* Project planning
  + User stories – a description of functionality from a user’s perspective
    - “As a X, I want Y so that Z.
      * Who | what | why
    - As a user, I want to log my running workouts with location, distance, time, pace and steps/minute, so I can keep a log of all my running
    - Log cycling workouts with location, distance, time, speed and elevation gain
    - As a user, I want to see all my workouts at a glance so I can easily track my progress over time
    - I want to see my workouts on a map, so I can easily check where I work out the most
    - I want to see all my workouts when I leave the app and come back later, so that I can keep using the app over time
  + Features
  + Flowchart – the what
  + Architecture – the how
  + Development

Diagram

Description automatically generated

Destructuring

* *// const latitude = position.coords.latitude;*
* *// const longitude = position.coords.longitude;*
* *const* {latitude} = position.coords;
* *const* {longitude} = position.coords;

Geolocation API

navigator.geolocation.getCurrentPosition(

*function* (*position*) {

    console.log(position);

*// const latitude = position.coords.latitude;*

*// const longitude = position.coords.longitude;*

*const* {latitude} = position.coords;

*const* {longitude} = position.coords;

    console.log(`https://www.google.com.au/maps/@${latitude},${longitude},15z`);

  },

*function* () {

    alert('Could not get your position');

  }

);

Leaflet API

<script

*defer*

*src*="https://unpkg.com/leaflet@1.7.1/dist/leaflet.js"

*integrity*="sha512-XQoYMqMTK8LvdxXYG3nZ448hOEQiglfqkJs1NOQV44cWnUrBc8PkAOcXy20w0vlaXaVUearIOBhiXZ5V3ynxwA=="

*crossorigin*=""

    ></script>

* Note that all the global variables inside leaflet.js is globally accessible to rest of code

14 December 2021

* .focus() method – useful for better user experience, sends mouse to desired location
* Toggling on / off input fields

Text

Description automatically generated

Architecture

Key data

* Running – location / distance / time / pace / cadence
* Cycling – location / distance / time / speed / elevation gain
* Include methods in constructor function to automatically load method upon object creation

Managing workout data

* Data entry validation

Graphical user interface, text

Description automatically generated

Mapty

'use strict';

*// prettier-ignore*

*class* Workout {

  date = new Date();

  id = (Date.now() + '').slice(-10);

  clicks = 0;

*constructor*(*coords*, *distance*, *duration*) {

*// this.date = ...*

*// this.id = ...*

*this*.coords = coords; *// [lat, lng]*

*this*.distance = distance;

*this*.duration = duration;

  }

  \_setDescription(){

*const* months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December'];

*this*.description = `${*this*.type[0].toUpperCase()}${*this*.type.slice(1)} on ${months[*this*.date.getMonth()]} ${*this*.date.getDate()}`;

  }

  click(){

*this*.clicks++;

  }

}

*class* Running *extends* *Workout* {

  type = 'running';

*constructor*(*coords*, *distance*, *duration*, *cadence*) {

*super*(coords, distance, duration);

*this*.cadence = cadence;

*this*.calcPace();

*this*.\_setDescription();

  }

  calcPace() {

*//min/km output*

*this*.pace = *this*.duration / *this*.distance;

*// return this.pace;  //This can be subbed by including in constructor function*

  }

}

*class* Cycling *extends* *Workout* {

  type = 'cycling';

*constructor*(*coords*, *distance*, *duration*, *elevationGain*) {

*super*(coords, distance, duration);

*this*.elevationGain = elevationGain;

*this*.calcSpeed();

*this*.\_setDescription();

  }

  calcSpeed() {

*//km/h*

*this*.speed = Math.trunc(*this*.distance / (*this*.duration / 60));

  }

}

*const* run1 = new Running([39, -12], 5.2, 24, 178);

*const* cycling1 = new Cycling([39, -12], 27, 95, 523);

console.log(run1, cycling1);

*/////////////////////////////////////////////////////////////*

*// APPLICATION ARCHITECTURE*

*const* form = document.querySelector('.form');

*const* containerWorkouts = document.querySelector('.workouts');

*const* inputType = document.querySelector('.form\_\_input--type');

*const* inputDistance = document.querySelector('.form\_\_input--distance');

*const* inputDuration = document.querySelector('.form\_\_input--duration');

*const* inputCadence = document.querySelector('.form\_\_input--cadence');

*const* inputElevation = document.querySelector('.form\_\_input--elevation');

*class* App {

  #map;

  #mapEvent;

  #workouts = [];

  #mapZoomLevel = 13;

*constructor*() {

*// Get user's position*

*this*.\_getPosition();

*//Get data from local storage*

*this*.\_getLocalStorage();

*//Attach event handlers*

    form.addEventListener('submit', *this*.\_newWorkout.bind(*this*));

    inputType.addEventListener('change', *this*.\_toggleElevationField);

    containerWorkouts.addEventListener('click', *this*.\_moveToPopup.bind(*this*));

  }

  \_getPosition() {

    if (navigator.geolocation)

      navigator.geolocation.getCurrentPosition(

*this*.\_loadMap.bind(*this*),

*function* () {

          alert('Could not get your position');

        }

      );

  }

  \_loadMap(*position*) {

    console.log(position);

*// const latitude = position.coords.latitude;*

*// const longitude = position.coords.longitude;*

*const* { latitude } = position.coords;

*const* { longitude } = position.coords;

    console.log(`https://www.google.com.au/maps/@${latitude},${longitude}`);

*const* coords = [latitude, longitude];

*this*.#map = L.map('map').setView(coords, *this*.#mapZoomLevel);

    L.tileLayer('https://{s}.tile.openstreetmap.fr/hot/{z}/{x}/{y}.png', {

      attribution:

        '&copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> contributors',

    }).addTo(*this*.#map);

*//Handling clicks on map*

*this*.#map.on('click', *this*.\_showForm.bind(*this*));

*//Render markers after map fully loaded*

*this*.#workouts.forEach(*work* => {

*this*.\_renderWorkout(work);

*this*.\_renderWorkoutMarker(work);

    });

  }

  \_showForm(*mapE*) {

*this*.#mapEvent = mapE;

    form.classList.remove('hidden');

    inputDistance.focus();

  }

  \_hideForm() {

*//Empty inputs*

    inputDistance.value =

      inputDuration.value =

      inputCadence.value =

      inputElevation.value =

        '';

    form.style.display = 'none';

    form.classList.add('hidden');

    setTimeout(() => (form.style.display = 'grid'), 1000);

  }

  \_toggleElevationField() {

    inputElevation.closest('.form\_\_row').classList.toggle('form\_\_row--hidden');

    inputCadence.closest('.form\_\_row').classList.toggle('form\_\_row--hidden');

  }

  \_newWorkout(*e*) {

*//Helper functions*

*const* validInputs = (...*inputs*) =>

      inputs.every(*inp* => Number.isFinite(inp));

*const* allPositive = (...*inputs*) => inputs.every(*inp* => inp > 0);

    e.preventDefault();

*//Get data from form*

*const* type = inputType.value;

*const* distance = +inputDistance.value;

*const* duration = +inputDuration.value;

*const* { lat, lng } = *this*.#mapEvent.latlng;

*let* workout;

*//Check if data is valid*

*//If workout is running, create running object*

    if (type === 'running') {

*const* cadence = +inputCadence.value;

      if (

        !validInputs(distance, duration, cadence) ||

        !allPositive(distance, duration, cadence)

      )

        return alert('Inputs have to be positive numbers!');

      workout = new Running([lat, lng], distance, duration, cadence);

    }

*//If cycling, create cycling object*

    if (type === 'cycling') {

*const* elevation = +inputElevation.value;

      if (

        !validInputs(distance, duration, elevation) ||

        !allPositive(distance, duration)

      )

        return alert('Inputs have to be positive numbers!');

      workout = new Cycling([lat, lng], distance, duration, elevation);

    }

*//Add new object to workout object*

*this*.#workouts.push(workout);

    console.log(workout);

*//Render workout on map as marker*

*this*.\_renderWorkoutMarker(workout);

*//Render workout on list*

*this*.\_renderWorkout(workout);

*//Hide form and clear input fields*

*this*.\_hideForm();

*//Set local storage to all workouts*

*this*.\_setLocalStorage();

  }

  \_renderWorkoutMarker(*workout*) {

    L.marker(workout.coords)

      .addTo(*this*.#map)

      .bindPopup(

        L.popup({

          maxWidth: 250,

          minWidth: 100,

          autoClose: false,

          closeOnClick: false,

          className: `${workout.type}-popup`,

        })

      )

      .setPopupContent(

        `${workout.type === 'running' ? ' 🏃‍♂️' : ' 🚴‍♀️'} ${workout.description}`

      )

      .openPopup();

  }

  \_renderWorkout(*workout*) {

*let* html = `

        <li class="workout workout--${workout.type}" data-id=${workout.id}>

          <h2 class="workout\_\_title">${workout.description}</h2>

          <div class="workout\_\_details">

            <span class="workout\_\_icon">${

              workout.type === 'running' ? ' 🏃‍♂️' : ' 🚴‍♀️'

            }</span>

            <span class="workout\_\_value">${workout.distance}</span>

            <span class="workout\_\_unit">km</span>

          </div>

          <div class="workout\_\_details">

            <span class="workout\_\_icon">⏱</span>

            <span class="workout\_\_value">${workout.duration}</span>

            <span class="workout\_\_unit">min</span>

          </div>

          `;

    if (workout.type === 'running')

      html += `

      <div class="workout\_\_details">

        <span class="workout\_\_icon">⚡️</span>

        <span class="workout\_\_value">${workout.pace}</span>

        <span class="workout\_\_unit">min/km</span>

      </div>

      <div class="workout\_\_details">

        <span class="workout\_\_icon">🦶🏼</span>

        <span class="workout\_\_value">${workout.cadence}</span>

        <span class="workout\_\_unit">spm</span>

      </div>

      `;

    if (workout.type === 'cycling')

      html += `

      <div class="workout\_\_details">

        <span class="workout\_\_icon">⚡️</span>

        <span class="workout\_\_value">${workout.speed}</span>

        <span class="workout\_\_unit">km/h</span>

      </div>

      <div class="workout\_\_details">

        <span class="workout\_\_icon">⛰</span>

        <span class="workout\_\_value">${workout.elevationGain}</span>

        <span class="workout\_\_unit">m</span>

      </div>

    `;

    form.insertAdjacentHTML('afterend', html);

  }

  \_moveToPopup(*e*) {

*const* workoutEl = e.target.closest('.workout');

    console.log(workoutEl);

    if (!workoutEl) return;

*const* workout = *this*.#workouts.find(

*work* => work.id === workoutEl.dataset.id

    );

    console.log(workout);

*this*.#map.setView(workout.coords, *this*.#mapZoomLevel, {

      animate: true,

      pan: {

        duration: 1

      }

    });

*//Using the public interface*

    workout.click(); *//Not compatible with local storage*

  }

  \_setLocalStorage(){

    localStorage.setItem('workouts', JSON.stringify(*this*.#workouts));

  }

  \_getLocalStorage(){

*const* data = JSON.parse(localStorage.getItem('workouts'));

    console.log(data);

    if(!data) return;

*this*.#workouts = data;

  }

}

*const* app = new App();

15 December 2021

Modern Javascript development

* Going from ‘chronological’ development to modular development w/ 3rd-party packages
  + Node Package Manager (NPM)
* Process
  + Development (modules / 3rd party packages)
  + Build (webpack, PARCEL, Babel)
    - Bundling (join all modules into one file)
    - Transpiling / polyfiling (convert modern JS back to ES5)
  + Production – final JS bundle
* Modules in Javascript
  + What is a module? Reusable piece of code that encapsulates implementation details
  + Usually a standalone file
  + Why?
    - Composability - Modules allow for small building blocks that can be put together to build greater and greater applications
    - Modules can be developed in isolation – task delegation
    - Abstraction – implement low-level code in modules then import these abstractions into other modules
    - Re-usable code – modules can be transferred across projects
  + Native Javascript (ES6) modules
    - One module per file. What’s the difference with scripts?

|  |  |  |
| --- | --- | --- |
|  | **ES6 module** | **Script** |
| Top level variables | Scoped to module | Global |
| Default mode | Strict mode | Sloppy mode |
| Top-level this | Undefined | Window |
| Import and exports | Yes | No |
| HTML linking | <script type = “module”> | <script> |
| File downloading | Asynchronous | Synchronous |

* Imports should be at top-level – ‘hoisted’
  + Cannot be nested

Diagram

Description automatically generated

* Import / export
* As keyword – change name of imported variable
* *import {*
* *//   addToCart,*
* *//   totalPrice as price,*
* *//   totalQuantity,*
* *// } from './shoppingCart.js';*
* *// addToCart('bread', 5);*

*//Exporting module*

console.log('Exporting module');

*const* shippingCost = 10;

*const* cart = [];

export *const* addToCart = *function*(*product*, *quantity*){

    cart.push({product, quantity});

    console.log(`${quantity} ${product} added to cart`);

};

*const* totalPrice = 237;

*const* totalQuantity = 23;

export { totalPrice, totalQuantity};

* Import entire module

import \* as ShoppingCart from './shoppingCart.js';

ShoppingCart.addToCart('bread', 5);

* Default exports – export without a name
* export default *function*(*product*, *quantity*){
* cart.push({product, quantity});
* console.log(`${quantity} ${product} added to cart`);
* };

import add from './shoppingCart.js';

add('pizza', 2);

* Mixing default exports and named exports – not good practice
* Imports are not copies, but instead links to the original modules
* Note that the <script> tag in the HTML file will need a “type=module”
* Top-level await – using await outside of the async function
  + Note that now the code becomes synchronous – ‘something’ will only run after the fetch has been completed

Text

Description automatically generated with low confidence

The Module Pattern

* To encapsulate functionality with private data while exposing any public API
* Using Immediately-Invoked-Function-Expressions (IIFE)
* *const* shoppingCart2 = (*function*(){
* *const* cart = [];
* *const* shippingCost = 10;
* *const* totalPrice = 237;
* *const* totalQuantity = 23;
* *const* addToCart = *function*(*product*, *quantity*){
* cart.push({product, quantity});
* console.log(`${quantity} ${product} added to cart`);
* };
* *const* orderStock = *function*(*product*, *quantity*){
* console.log(`${quantity} ${product} ordered from supplier`);
* };
* return{
* addToCart,
* cart,
* totalPrice,
* totalQuantity
* }
* })();
* shoppingCart2.addToCart('apple', 4)
* Closures – allow function to have access to all variables in its birthplace

CommonJS modules

*//nodeJS environment*

*//Export*

export.addToCart = *function*(*product*, *quantity*){

    cart.push({product, quantity});

    console.log(`${quantity} ${product} added to cart`);

};

*//Import*

*const* { addToCart } = require('./shoppingCart.js');

Command line

* Cd – change directory
  + Move up – cd..
  + Move up two levels cd ../..
* Dir – check what is in current directory
* Clear – clear console
* Mkdir – Make new folder directory
* Type nul > index.html – make new files
* Del – removing files
* Mv\_fileName\_Directory – move folder
* Rm\_directoryName – delete empty directories

NPM

* Why NPM?
  + Keeping libraries updated
  + Having a single repository for APIs

import cloneDeep from './node\_modules/lodash-es/cloneDeep.js';

*const* state = {

    cart: [

        {product: 'bread', quantity: 5},

        {product: 'pizza', quantity: 5},

    ],

    user: { loggedIn: true },

};

*const* stateClone = Object.assign({}, state);

*const* stateDeepClone = cloneDeep(state);

state.user.loggedIn  = false;

console.log(stateClone);

console.log(stateDeepClone);

* Restoring all npm files – npm i

Bundling with parcel and npm scripts

* Bundling – compressing modules into a single output file
* Dev dependencies
  + Npm I parcel –save-dev

Configuring babel and polyfilling

* Back-compatibility of code via babel
* Polyfilling – import ‘core-js/stable’
  + Re-create ES6 functions
* Polyfilling async functions
  + Npm I regenerator-runtime

Clean and modern javascript code – review

* Create readable code – so others can read and the future-you can read
  + Keep things simple
  + Descriptive variable names (what they contain) / function names (what they do)
* General
  + DRY principle
  + Don’t pollute global namespace
  + Don’t use var
  + Use strong type checks === and !==
* Functions
  + Generally should be single purposed
  + Do not use more than 3 function parameters
  + Use default parameters whenever possible
  + Generally, return same data type as received
  + Use arrow functions when they make code more readable
* OOP
  + Use ES6 classes
  + Encapsulate data and don’t mutate it from outside the class
  + Implement method chaining (this keyword is critical as well as .bind)
  + Do not use arrow functions as methods – cannot use this keyword
* Avoid nested code
  + Use early return (guard clauses)
  + Use ternary or logical operators instead of if
  + Use multiple if instead of if/else-if
  + Avoid for loops, use array methods instead
  + Avoid callback-based async APIs
* Asynchronous code
  + Use async/await for best readability instead of then..catch – more nested code via more callback functions
  + Whenever possible, run promises in parallel (Promise.all)
  + Handle errors and promise rejections

Fixing some bad code

* Optional chaining / nullish coalescing operator
* *const* limit = spendingLimits[user] ? spendingLimits[user] : 0;
* *// Optional chaining solution*
* *const* limit1 = spendingLimits?.[user] ?? 0;

16 December 2021

Declarative and functional javascript principles

* Two paradigms – imperative vs declarative
* Imperative
  + Programmer explains “how to do things”
* Declarative – best practice
  + Programmers tells WHAT to do
  + The how gets abstracted away
* Readability vs succinctness (?)
* Functional programming principles
  + Declarative programming paradigm – write software by combining many *pure functions*, avoiding side effects and mutating data
  + Side effect – mutation of any data outside of the function
  + Pure function – does not mutate or depend on external variables, given the same input it always returns the same outputs
  + Immutability – state (data) is never modified, state is copied and it is the copy that is mutated then returned
    - Makes it much easier to keep track of data flow
* Frameworks use functional programming – React, Redux
* Functional programming techniques
  + Avoid data mutations
  + Use built-in methods that don’t produce side effects
  + Do data transformations with methods - .map(), .filter(), .reduce()
  + Avoid side effects in functions
* Functional programming – declarative syntax
  + Use array/object destructuring
  + Use spread operator
  + Use ternary operator
  + Use template literals
* Developing functional programs takes time and experience (and probably iterations!)

Fixing bad code - #2

* Immutability
  + Object.freeze() – only first level freeze, not a deep freeze. Objects within the object can still be mutated
* *const* spendingLimits = Object.freeze({
* jonas: 1500,
* matilda: 100,
* });

Forkify App

Project overview

* User stories
  + As a user, I want to search for recipes so that I can find new ideas for meals
    - Search functionality – input field to send request to API
    - Display results with pagination – hide search results behind pages to not overwhelm with information
    - Display recipe with cooking time, servings and ingredients
  + As a user, I want to be able to update the number of servings so that I can cook a meal for different number of people
    - Change servings functionality
  + As a user, I want to bookmark recipes so that I can review them later
    - Bookmarking functionality
  + As a user, I want to be able to create my own recipes, so that I can have all of them organised in the same app
    - User can upload own recipe
    - Automatically bookmark user recipes
    - Users can only see their own recipes
  + As a user, I want to see my bookmarks and recipes when I leave the app and come back later
    - Store bookmark data using localstorage
    - On page load, read save bookmarks from localstorage and display

Loading a recipe from API

*// import icons from '../img/icons.svg';*

import icons from 'url:../img/icons.svg';

import 'core-js/stable'; *// For polyfilling everything else*

import 'regenerator-runtime/runtime'; *// For polyfilling async/await*

*const* recipeContainer = document.querySelector('.recipe');

*const* timeout = *function* (*s*) {

  return new Promise(*function* (*\_*, *reject*) {

    setTimeout(*function* () {

      reject(new Error(`Request took too long! Timeout after ${s} second`));

    }, s \* 1000);

  });

};

*// https://forkify-api.herokuapp.com/v2*

*///////////////////////////////////////*

*const* renderSpinner = *function*(*parentEl*) {

*const* markup = `

  <div class="spinner">

    <svg>

      <use href="${icons}#icon-loader"></use>

    </svg>

  </div>`

  parentEl.innerHTML = '';

  parentEl.insertAdjacentHTML('afterbegin', markup);

}

*const* showRecipe = *async* *function* () {

  try {

*//Load data from API*

    renderSpinner(recipeContainer);

*const* res = await fetch(

      'https://forkify-api.herokuapp.com/api/v2/recipes/5ed6604591c37cdc054bc886'

    );

*const* data = await res.json();

    if (!res.ok) throw new Error(`${data.message} (${res.status})`);

    console.log(res, data);

*// Deconstruct API data into object*

*let* { recipe } = data.data;

    recipe = {

      id: recipe.id,

      title: recipe.title,

      publisher: recipe.publisher,

      sourceUrl: recipe.source\_url,

      image: recipe.image\_url,

      servings: recipe.servings,

      cookingTime: recipe.cooking\_time,

      ingredients: recipe.ingredients,

    };

    console.log(recipe);

*//Rendering recipe*

*const* markup = `

        <figure class="recipe\_\_fig">

          <img src="${recipe.image}" alt="${recipe.title}" class="recipe\_\_img" />

          <h1 class="recipe\_\_title">

            <span>${recipe.title}</span>

          </h1>

        </figure>

        <div class="recipe\_\_details">

          <div class="recipe\_\_info">

            <svg class="recipe\_\_info-icon">

              <use href="${icons}#icon-clock"></use>

            </svg>

            <span class="recipe\_\_info-data recipe\_\_info-data--minutes">${recipe.cookingTime}</span>

            <span class="recipe\_\_info-text">minutes</span>

          </div>

          <div class="recipe\_\_info">

            <svg class="recipe\_\_info-icon">

              <use href="${icons}#icon-users"></use>

            </svg>

            <span class="recipe\_\_info-data recipe\_\_info-data--people">${recipe.servings}</span>

            <span class="recipe\_\_info-text">servings</span>

            <div class="recipe\_\_info-buttons">

              <button class="btn--tiny btn--increase-servings">

                <svg>

                  <use href="${icons}#icon-minus-circle"></use>

                </svg>

              </button>

              <button class="btn--tiny btn--increase-servings">

                <svg>

                  <use href="${icons}#icon-plus-circle"></use>

                </svg>

              </button>

            </div>

          </div>

          <div class="recipe\_\_user-generated">

            <svg>

              <use href="${icons}#icon-user"></use>

            </svg>

          </div>

          <button class="btn--round">

            <svg class="">

              <use href="${icons}#icon-bookmark-fill"></use>

            </svg>

          </button>

        </div>

        <div class="recipe\_\_ingredients">

          <h2 class="heading--2">Recipe ingredients</h2>

          <ul class="recipe\_\_ingredient-list">

          ${recipe.ingredients.map(*ing* => {

            return `

            <li class="recipe\_\_ingredient">

              <svg class="recipe\_\_icon">

                <use href="${icons}#icon-check"></use>

              </svg>

              <div class="recipe\_\_quantity">${ing.quantity}</div>

              <div class="recipe\_\_description">

                <span class="recipe\_\_unit">${ing.unit}</span>

                ${ing.description}

              </div>

            </li>`

          }).join('')};

          </ul>

        </div>

        <div class="recipe\_\_directions">

          <h2 class="heading--2">How to cook it</h2>

          <p class="recipe\_\_directions-text">

            This recipe was carefully designed and tested by

            <span class="recipe\_\_publisher">${recipe.publisher}</span>. Please check out

            directions at their website.

          </p>

          <a

            class="btn--small recipe\_\_btn"

            href="${recipe.sourceUrl}"

            target="\_blank"

          >

            <span>Directions</span>

            <svg class="search\_\_icon">

              <use href="${icons}#icon-arrow-right"></use>

            </svg>

          </a>

        </div>`;

      recipeContainer.innerHTML ='';

      recipeContainer.insertAdjacentHTML('afterbegin', markup);

  } catch (err) {

    alert(err);

  }

};

showRecipe();

Listening for load and hashchange events

* The hash being the recipe ID
* Listening for multiple addEventListener events –
* ['hashchange', 'load'].forEach(*ev* => window.addEventListener(ev, showRecipe));

The Model View Controller (MVC) architecture

* Why bother?
  + Structure - Like a house needs structure, code needs a structure
  + Maintainability – allow for easy changes
  + Expandability – easily add new features
* Components of any architecture
  + Business logic – code that solves the actual business problem, directly related to what the business needs
  + State – stores all the data about the application
    - Treat as ‘the single source of truth’
    - UI kept in sync with the state
  + HTTP Library – responsible for making/receiving AJAX requests
  + Application logic (router) – the implementation of the application itself, handles navigation and UI events
  + Presentation logic (UI layer) – managing the visible part of the application
* The Model – View – Controller architecture
  + Model – business logic / state / HTTP library
  + Controller (bridges the model and the view, which don’t know about each other)– application logic
  + The view – presentation logic

Chart

Description automatically generated with medium confidence

Diagram

Description automatically generated

Helpers and configuration files

* Storing common, persisting variables in one location



Text

Description automatically generated

Event handlers in MVC – publisher-subscriber pattern

* Events should be **handled** in the Controller (otherwise there is application logic in the view)
* Events should be **listened for** in the View
* But this is difficult bc the view does not know anything about the controller
* Subscriber – code that wants to react
* Publisher – code that knows when to react

Graphical user interface, text, application

Description automatically generated

* The publisher-subscriber pattern keeps the listener in the View and the handler in the Controller

Graphical user interface

Description automatically generated

Graphical user interface, text, application

Description automatically generated

22 December 2021

Implementing Search results – 1/3

* First step is to work with the data – within the Model

Text

Description automatically generated

Text

Description automatically generated

Implementing search results – 2/3

Grabbing id from url

*const* id = window.location.hash.slice(1);

    if(!id) return;

Preventing page refreshes in between edits via Parcel

if(*module*.hot){

*module*.hot.accept();

}

Guard clause good practice – checking if any data provided || array is empty

if(!data || (Array.isArray(data) && data.length === 0)) return *this*.renderError();

Implementing pagination – ½

import View from './view.js';

import icons from 'url:../../img/icons.svg';

*class* PaginationView *extends* *View* {

  \_parentElement = document.querySelector('.pagination');

  \_generateMarkup() {

*const* curPage = *this*.\_data.page;

*const* numPages = Math.ceil(

*this*.\_data.results.length / *this*.\_data.resultsPerPage

    );

    console.log(numPages);

    console.log(curPage);

*//Page 1 and there are other pages*

    if (curPage === 1 && numPages > 1) {

      return `

        <button class="btn--inline pagination\_\_btn--next">

            <span>${curPage + 1}</span>

            <svg class="search\_\_icon">

              <use href="${icons}}#icon-arrow-right"></use>

            </svg>

        </button>

          `;

    }

*//Last page*

    if (curPage === numPages && numPages > 1) {

      return `

        <button class="btn--inline pagination\_\_btn--prev">

            <svg class="search\_\_icon">

                <use href="${icons}#icon-arrow-left"></use>

            </svg>

            <span>${curPage - 1}</span>

        </button>`;

    }

*//Other page*

    if (curPage < numPages) {

      return 'Other';

    }

*//Page 1 and there are no other pages*

    return '1 page and no others';

  }

}

export default new PaginationView();

23 November 2021

Implementing pagination 2/2

Publisher-subscriber addEventListener

addHandlerClick(*handler*){

*this*.\_parentElement.addEventListener('click', *function*(*e*){

      e.preventDefault();

*const* btn = e.target.closest('.btn--inline');

      console.log(btn);

      if(!btn) return;

*const* goToPage = +btn.dataset.goto;

      handler(goToPage);

    })

  }

*const* init = *function* () {

  recipeView.addHandlerRender(controlRecipes);

  searchView.addHandlerSearch(controlSearchResults);

  paginationView.addHandlerClick(controlPagination);

};

*const* controlPagination = *function* (*goToPage*) {

*//Render new results*

*// resultsView.render(model.state.search.results);*

  resultsView.render(model.getSearchResultsPage(goToPage));

*//Render NEW pagination buttons*

  paginationView.render(model.state.search);

};

Project Planning II

Updating servings

addHandlerUpdateServings(*handler*){

*this*.\_parentElement.addEventListener('click', *function*(*e*){

*const* btn = e.target.closest('.btn--update-servings');

      if (!btn) return;

*// const {updateTo} = btn.dataset;*

*const* updateTo = +btn.dataset.updateTo;

      handler(updateTo);

    })

  };

*const* controlServings = *function*(*newServings*){

*//Update the recipe servings (in state)*

  model.updateServings(newServings);

*// Update the recipe view*

  recipeView.render(model.state.recipe);

}

*const* init = *function* () {

  recipeView.addHandlerRender(controlRecipes);

  recipeView.addHandlerUpdateServings(controlServings);

  searchView.addHandlerSearch(controlSearchResults);

  paginationView.addHandlerClick(controlPagination);

};

Developing a DOM updating algorithm

nodeValue – checking for text only

  update(*data*) {

    if (!data || (Array.isArray(data) && data.length === 0))

      return *this*.renderError();

*this*.\_data = data;

*const* newMarkup = *this*.\_generateMarkup();

*//Comparing new markup with existing markup in memory*

*const* newDOM = document.createRange().createContextualFragment(newMarkup);

*const* newElements = Array.from(newDOM.querySelectorAll('\*'));

*const* curElements = Array.from(*this*.\_parentElement.querySelectorAll('\*'));

    newElements.forEach((*newEl*, *i*) => {

*const* curEl = curElements[i];

      console.log(curEl, newEl.isEqualNode(curEl));

*//Update changed texts*

      if (

        !newEl.isEqualNode(curEl) &&

        newEl.firstChild?.nodeValue.trim() !== ''

      ) {

        curEl.textContent = newEl.textContent;

      }

*//Update changed attributes*

      if (!newEl.isEqualNode(curEl)) {

        Array.from(newEl.attributes).forEach(*attr* =>

          curEl.setAttribute(attr.name, attr.value)

        );

      }

    });

  }

Implementing bookmarks – ½

Controller

*const* controlAddBookmark = *function* () {

  if (!model.state.recipe.bookmarked) model.addBookmark(model.state.recipe);

  else model.deleteBookmark(model.state.recipe.id);

  console.log(model.state.recipe);

  recipeView.update(model.state.recipe);

};

*const* init = *function* () {

  recipeView.addHandlerRender(controlRecipes);

  recipeView.addHandlerUpdateServings(controlServings);

  recipeView.addHandlerAddBookmark(controlAddBookmark);

  searchView.addHandlerSearch(controlSearchResults);

  paginationView.addHandlerClick(controlPagination);

};

init();

Model

export *const* addBookmark = *function*(*recipe*){

*//Add bookmark*

  state.bookmarks.push(recipe);

  console.log(state.bookmarks);

*//Mark current recipe as bookmark*

  if(recipe.id === state.recipe.id) state.recipe.bookmarked = true;

};

export *const* deleteBookmark = *function*(*id*){

*const* index = state.bookmarks.findIndex(*el* => el.id === id);

  state.bookmarks.splice(index, 1);

*//Mark current recipe as NOT bookmarked*

  if(id === state.recipe.id) state.recipe.bookmarked = false;

};

recipeView

  addHandlerAddBookmark(*handler*){

*this*.\_parentElement.addEventListener('click', *function*(*e*){

*const* btn = e.target.closest('.btn--bookmark');

      if(!btn) return;

      handler();

    });

  };

DRY – previewView.js

Text

Description automatically generated

After DRY

Text

Description automatically generated

Text

Description automatically generated

29 December 2021

Storing bookmarks with LocalStorage

*const* persistBookmarks = *function*(){

  localstorage.setItem('bookmarks', JSON.stringify(state.bookmarks));

}

export *const* addBookmark = *function*(*recipe*){

*//Add bookmark*

  state.bookmarks.push(recipe);

*//Mark current recipe as bookmark*

  if(recipe.id === state.recipe.id) state.recipe.bookmarked = true;

  persistBookmarks();

};

export *const* deleteBookmark = *function*(*id*){

*const* index = state.bookmarks.findIndex(*el* => el.id === id);

  state.bookmarks.splice(index, 1);

*//Mark current recipe as NOT bookmarked*

  if(id === state.recipe.id) state.recipe.bookmarked = false;

  persistBookmarks();

};

*const* init = *function*(){

*const* storage = localStorage.getItem('bookmarks');

  if (storage) state.bookmarks = JSON.parse(storage);

};

init();

*const* clearBookmarks = *function*(){

  localStorage.clear('bookmarks');

}

Uploading a new recipe

* When no interaction required with controller, self-declare function within constructor
* Handling forms – […new FormData(this)], Object.fromEntries([ ])
* addHandlerUpload(*handler*){
* *this*.\_parentElement.addEventListener('submit', *function*(*e*){
* e.preventDefault();
* *const* dataArr = [...new FormData(*this*)]
* *const* data = Object.fromEntries(dataArr);
* console.log(data);
* handler(data);
* });
* };
* Conditionally adding properties to objects via spread operator
* *const* createRecipeObject = *function*(*data*){
* *const* { recipe } = data.data;
* return {
* id: recipe.id,
* title: recipe.title,
* publisher: recipe.publisher,
* sourceUrl: recipe.source\_url,
* image: recipe.image\_url,
* servings: recipe.servings,
* cookingTime: recipe.cooking\_time,
* ingredients: recipe.ingredients,
* ...(recipe.key && {key: recipe.key}),
* };
* }
* Updating new ID upon recipe upload
* *// Change ID in URL*
* window.history.pushState(null, '', `#${model.state.recipe.id}`);

Recipe upload

import View from './view.js';

import icons from 'url:../../img/icons.svg';

*class* AddRecipeView *extends* *View* {

  \_parentElement = document.querySelector('.upload');

  \_message = 'Recipe was successfully uploaded!'

  \_window = document.querySelector('.add-recipe-window');

  \_overlay = document.querySelector('.overlay');

  \_btnOpen = document.querySelector('.nav\_\_btn--add-recipe');

  \_btnClose = document.querySelector('.btn--close-modal');

*constructor*(){

*super*();

*this*.\_addHandlerShowWindow();

*this*.\_addHandlerHideWindow();

  };

  toggleWindow(){

*this*.\_overlay.classList.toggle('hidden');

*this*.\_window.classList.toggle('hidden');

  }

  \_addHandlerShowWindow(){

*this*.\_btnOpen.addEventListener('click', *this*.toggleWindow.bind(*this*));

  };

  \_addHandlerHideWindow(){

*this*.\_btnClose.addEventListener('click', *this*.toggleWindow.bind(*this*));

*this*.\_overlay.addEventListener('click', *this*.toggleWindow.bind(*this*));

  }

  addHandlerUpload(*handler*){

*this*.\_parentElement.addEventListener('submit', *function*(*e*){

          e.preventDefault();

*const* dataArr = [...new FormData(*this*)]

*const* data = Object.fromEntries(dataArr);

        handler(data);

      });

  };

  \_generateMarkup() {

}

}

export default new AddRecipeView();

*const* controlAddRecipe = *async* *function*(*newRecipe*){

  try{

*//Show loading spinner*

  addRecipeView.renderSpinner();

*// Upload new recipe data*

  await model.uploadRecipe(newRecipe);

*//Render recipe*

    recipeView.render(model.state.recipe);

*//Success message*

  addRecipeView.renderMessage();

*//Render bookmark view*

  bookmarksView.render(model.state.bookmarks);

*// Change ID in URL*

  window.history.pushState(null, '', `#${model.state.recipe.id}`);

*// window.history.back()*

*//Close form window*

  setTimeout(*function*(){

    addRecipeView.toggleWindow();

  }, MODAL\_CLOSE\_SEC \* 1000);

  } catch(err){

    console.error(err);

    addRecipeView.renderError(err.message);

  }

}

*const* init = *function* () {

  bookmarksView.addHandlerRender(controlBookmarks);

  recipeView.addHandlerRender(controlRecipes);

  recipeView.addHandlerUpdateServings(controlServings);

  recipeView.addHandlerAddBookmark(controlAddBookmark);

  searchView.addHandlerSearch(controlSearchResults);

  paginationView.addHandlerClick(controlPagination);

  addRecipeView.addHandlerUpload(controlAddRecipe)

};

init();

export *const* uploadRecipe = *async* *function* (*newRecipe*) {

  try {

    console.log(newRecipe);

*//Formatting the ingredients from inputs*

*const* ingredients = Object.entries(newRecipe)

      .filter(*entry* => entry[0].startsWith('ingredient') && entry[1] !== '')

      .map(*ing* => {

*const* ingArr = ing[1].split(',').map(*el* => el.trim());

        console.log(ingArr);

        if (ingArr.length !== 3)

          throw new Error(

            'Wrong ingredient format! Please use the correct format.'

          );

*const* [quantity, unit, description] = ingArr;

        return { quantity: quantity ? +quantity : null, unit, description };

      });

*const* recipe = {

      title: newRecipe.title,

      source\_url: newRecipe.sourceUrl,

      image\_url: newRecipe.image,

      publisher: newRecipe.publisher,

      cooking\_time: +newRecipe.cookingTime,

      servings: +newRecipe.servings,

      ingredients,

    };

*const* data = await AJAX(`${API\_URL}?key=${KEY}`, recipe);

    state.recipe = createRecipeObject(data);

    addBookmark(state.recipe);

  } catch (err) {

    throw err;

  }

};

Javascript documentation good practice

*/\*\**

*\**

*\* @param {\*} data*

*\* @param {\*} render*

*\* @returns*

*\*/*

*/\*\**

*\* Render the received object to the DOM*

*\* @param {Object | Object[]} data  The data to be rendered (e.g recipe)*

*\* @param {boolean} [render = true] if false, create makrup string instead of rendering to the DOM*

*\* @returns {undefined | string} A markup string is returned if render=false*

*\* @this {Object} View instance*

*\* @author Julian Goh*

*\* @todo Finish implementation*

*\*/*

Fin: 29/12/21