# JavaScript Essentials

## 001 Data types

* Number
* String
* Boolean
* Undefined (falsy)
* Null (falsy)
* Symbol
* Bigint

## 002 Operators

### ---Basic operators

* Mathematical operators (+, -, \*, /, %)
* Type (typeof)
* Assignment operators (=, ++, --)
* Comparison operators (>, <, >=, <=, ==, ===, !==)
* Logical operators (||, &&, !)
* Short-circuiting (||, &&)
* Nullish coalescing (??)
* Logical assignments (??=, ||=, &&=)
* Optional chaining (?.) used to check if a property exists

restaurant.openingHours?.mon?.open

also used to check if a method exists and then call it

restaurant.order?.(0, 1)

* Spread operator (Used on the right side of equal sign. Works with iterables: arrays, strings, maps, sets.) – cannot be used in template literals.

Const str = ‘Jonas’;

Const letters = […str];

* Rest operator (Used on the left side of equal sign.)

### ---Ternary operator

<condition> ? <code> : <code>;

## 003 Conditional syntaxes

* If-else

if (<condition>) {

<code block>

} else {

<code block>

}

* Switch

switch(<variable name>) {

case <value>:

<code>;

Break;

case <value>:

<code>;

Break;

Default:

<code>

}

## 003 Functions

### ---Function expression

* Hoisting depends on using var/let/const
* Scoping depends on using var/let/const
* **Remember** | let/const not hoisted – block scoped
* **Remember** | var hoisted – function scoped
* The this keyword (strict mode): undefined if regular function call – object which calls the function as method

function <function name>(<argument>, <argument>) {

<code block>;

return <output variable>;

}

<function name>(<argument>, <argument>);

### ---Function declaration

* Hoisted
* Block scoped
* The this keyword (strict mode): undefined if regular function call – object which calls the function as method

const <function name> = function(<argument>, <argument>) {

<code block>;

return <output variable>;

}

<function name>(<argument>, <argument>);

#### Default parameters:

const <function name> = function(<argument>=200, <argument>=’jonas’) {

<code block>;

return <output variable>;

}

For skipping default parameters in between, you may use *undefined*.

#### Spread operator:

to pass multiple elements of an array as arguments into a function.

addFunction(…arguments);

#### Rest pattern:

makes the function receive any number of arguments with just specifying one argument name. packs all the received arguments into an array.

Const addFunction = function (…arguments) {}

#### Function methods

When calling functions as methods of classes, we may need to manually set the this keyword.

.call (receives first the this keyword as an argument, then the rest of the arguments that the function needs. Calls the function immediately.)

.apply (receives first the this keyword as an argument, then an array of the arguments that the function needs. Calls the function immediately. Consider using *bind* instead.)

.bind (receives the this keyword as an argument. Does not call the function immediately. Partial application possible. Refer to the main documentation p160.)

### ---Arrow function

* Hoisting depends on using var/let/const
* Scoping depends on using var/let/const
* **Remember** | let/const not hoisted – block scoped
* **Remember** | var hoisted – function scoped
* The this keyword: this keyword of the surrounding function

const <function name> = (<arugment>, <argument>) => {

<code block>;

return <output variable>;

}

<function name>(<argument>, <argument>);

Or

const <function name> = (<arugment>, <argument>) => <implicit return><code line>;

### ---Immediately Invoked function Expressions (IIFE)

* Not used anymore in modern JavaScript

(function () {

  <Code block>;

})();

## 004 Data structures

### Arrays

Create an empty array with a certain number of indexes.

const x = new Array(7);

Create an array with a certain number of indexes and fill it with a certain value

const y = Array.from({ length: 7 }, () => 1);

#### Retrieve data:

* Bracket notation:

array[0]

* At method

array.at(-1)

#### Array methods:

* .Length
* .Push (add to the end) – returns length
* .Unshift() (add to the beginning) – returns length
* .Pop (removes from the end) – returns removed element
* .Shift (removes from the beginning) – returns ??
* .indexOf (search for a value) – returns index
* .includes (search for equality) – returns Boolean
* .at (search by index) – returns element at the index
* .entries – returns another array containing key-value pairs
* .slice (extracts an array starting from a certain index to another. Does not change the original array. Also used to make shallow copies. Consider using negative numbers) – returns an array
* .splice (similar to slice, but does change the original array. Consider using negative numbers) – returns an array
* .reverse (reverses the array. changes the original array) – returns an array.
* .concat (to concat two arrays. Spread operator may be used instead.)

arr.concat(arr2)

* .join (joins elements of the array into a whole string with a certain character between them)
* .at (retrieves an element of the array at a certain index)
* .forEach (loops over array. requires callback function. Has access to the current element, the element’s index, and the entire array.)
* .map (loops over array. requires callback function. Has access to the current element, the element’s index, and the entire array.) – returns a new array based on the original array.
* .filter (loops over array. requires callback function. Has access to the current element, the element’s index, and the entire array. filters for elements in the array satisfying a certain condition.) – returns an array of filtered elements.
* .reduce (loops over array. requires callback function and an initial value for an accumulator variable. Has access to an accumulator variable, the current element, the element’s index, and the entire array. boils down all the elements of the array into one single value.) – returns a single value.

const balance = arr.reduce(function (acc, cur, i, arr) {

  <code block>;

}, 0);

* .find (search for condition. loops over array. requires callback function. Has access to the current element, the element’s index, and the entire array. usually used when only one element satisfies a condition.) – returns the first element that satisfies a certain condition.
* .findIndex (search for condition. similar to *find*) – returns the index of the element that satisfies the condition.
* .some (search for condition. loops over array. requires callback function. Has access to the current element, the element’s index, and the entire array.) – returns Boolean. true if any element satisfies the condition.
* .every (search for condition. loops over array. requires callback function. Has access to the current element, the element’s index, and the entire array.) – returns Boolean. True if all elements satisfies the condition.
* .flat (flats a nested array, by default to 1 level, or more if specified)
* .flatMap (used when we need to use *map* and then *flat*.)
* .sort (sorts elements as strings. Changes the original array.) – to sort numbers correctly, a comparing function should be used with *sort*:

movements.sort((*a*, *b*) => a - b); <ascending>

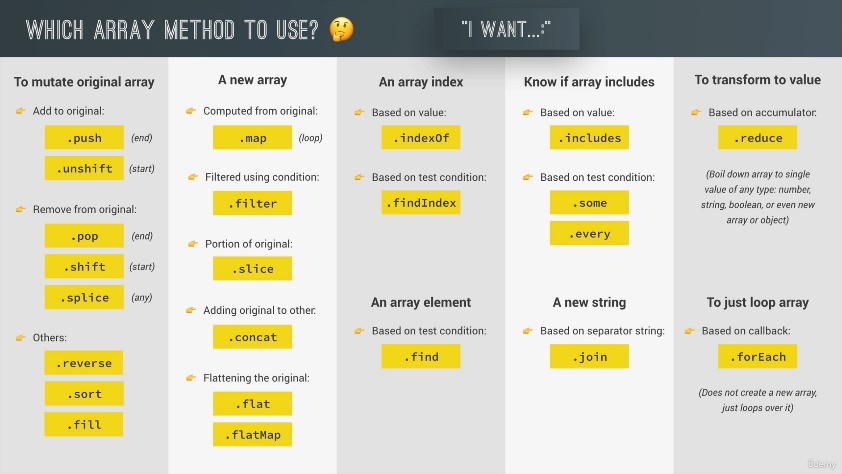
movements.sort((*a*, *b*) => b - a); <descending>

* .fill (used to fill an array with a certain value <first argument>, from a certain index <second argument> to another <third argument>)

x.fill(1, 3, 5);

* Array.from (used to create an array with certain number of indexes and filling it with certain value. Receives two arguments: object defining the length of array, callback function which has access to the current element, the current index, and the entire array)

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



#### De-structureing:

regular

Const arr = [2,3,4];

Const [x, y, z] = arr;

Nested

Const nested = [2, 4, [5, 6]];

Const [i, , [j, k]] = nested;

Ignoring

Const nested = [2, 4, [5, 6]];

Const [i, , j] = nested;

Determining default values to avoid undefined

Const defarr = [8, 9];

Const [p=1, q=1, r=1] = defarr;

#### Spread operator:

Used on the right side of equal sign to expand arrays into individual elements.

Const arr = [7, 8, 9];

Const newArr = [1, 2, …arr];

To create shallow copies

Const newMenuCopy = […mainMenu];

To merge two arrays into one new array

Const arrsum = […arr1, …arr2];

#### Rest pattern:

Used on the left side of equal sign to collect multiple elements and condense them into an array. It is the opposite of spread operator. Remember that the rest element must be the last element in a destructuring syntax. So it is obvious that there can only be one rest element in such syntaxes.

Const [a, b, …others] = [1, 2, 3, 4, 5];

### Objects

#### Retrieve data:

* dot notation:

object.firstName

* bracket notation (computations may be involved):

object[‘firstName’]

#### Object methods:

We basically define object methods ourselves! Consider using enhanced object literals.

* Object.keys(<object name>)
* Object.values(<object name>)
* Object.entries(<object name>)

#### De-structureing:

regular

Const {name, openingHours, categories} = restaurant;

Change variable names

Const {name: restaurantName, openingHours: hours, categories: tags} = restaurant;

Determining default values and changing names

Const {menu = [], starterMenu: starters = [] } = restaurant;

Nested

Const {fri} = openingHours;

Const {fri: {open, close}} = openingHours;

Nested and changing names

Const {fri: {open: o, close: c}} = openingHours;

#### Spread operator:

Since ES2018 the spread operator works on objects even though objects are not iterables.

const newRestaurant = {foundedIn: 1989, …restaurant, founder: ‘Guiseppe’};

Shallow copy

Const restaurantCopy = {…restaurant};

#### Rest pattern:

const { sat, ...weekDays } = restaurant.openingHours;

#### Converting objects to maps

const hoursMap = new Map(Object.entries(openingHours));

### Sets

Expects iterables. Returns a set (similar to an array) with no duplicate values. Sets are iterables themselves. No key-value pairs.

const orderSet = new Set(<iterable>);

#### Set methods:

* .size
* .has (similar to includes in arrays)
* .add
* .delete
* .clear
* .forEach (loops over set. requires callback function. Has access to the current element, the index which makes no sense in sets, and the entire set.)

#### Spread operator:

const staffUnique = […new Set(staff)];

### Maps

Used to map values to keys. data is stored in key-value pairs. Unlike objects (with string keys), in maps keys can have any type, even Booleans. Maps are iterables.

Creating a map

const rest = new Map();

then populating it with the set method. We can also create map and populate it at the same time:

const question = new Map([

  ['question', 'What is the best programming language in the world?'],

  [1, 'C'],

  [2, 'Java'],

  [3, 'JavaScript'],

]);

#### Map methods:

* .size
* .set (to add items) – returns the map > chaining multiple set methods is possible.

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

rest.set(1, 'Firenze, Italy');

* .get (to retrieve value related to a key)

rest.get('name')

if the key is an array, we cannot use the get method with the array literal inside it. Check the main document (p 137) for solution.

* .has (to check if a certain key exists in the map)
* .delete (to delete the value related to a certain key)
* .clear
* .keys
* .values
* .entries
* .forEach (loops over map. requires callback function. Has access to the value of current pair, the key of current pair, and the entire map.)

#### Converting maps to arrays

const questionArr = […question];

## 005 Loops

### For loop

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

<code block>;

}

* Consider using *continue* and *break* keywords under certain conditions.
* Can be used to loop over arrays.

### While loop

Let <counter> = 1;

while(<counter> <= 10) {

console.log(‘Lifting weights repetition ${rep}’);

<counter>++;

}

* Used when there is no need for a counter variable.

### For-of loop

for (const <variable> of <iterable>) {};

* Consider using *continue* and *break* keywords under certain conditions.
* Can be used to loop over arrays.

## 006 DOM manipulation

### Selecting

Selecting the entire HTML document, the head, or the body:

document.documentElement

document.head

document.body

#### .querySelector:

returns the first element that matches the query.

* Selecting elements

document.querySelector(‘img’);

document.querySelector(‘.message’);

document.querySelector(‘#message’);

* Selecting text content of text elements

document.querySelector(‘.message’).textContent;

* Selecting text value of textbox input

document.querySelector(‘.guess’).value

#### .querySelectorAll:

returns a node list containing all the elements matching the query.

#### .getElementByID:

returns the element with the matching ID.

#### .getElementsByTagName:

returns an HTML collection containing all the elements with the matching tag name.

#### .getElementsByClassName:

returns an HTML collection containing all the elements with the matching class name.

### Creating, inserting, deleting

Creating an element:

Const message = document.createElement(‘div’);

#### .innerHTML

possible to read or set html content:

Message.innerHTML = ‘We use cookies for improved functionality and analytics. <button class=”btn btn—close-cookie”>Got it!</button>’;

* May be used to delete any html inside a certain element:

document.querySelector('.movements').innerHTML = '';

#### .insertAdjacentHTML

Use with afterend/before begin to insert as sibling of a certain element. Use with beforeend/afterbegin to insert as child of a certain element.

const html = `<html syntax>`;

document.querySelector('.movements'). insertAdjacentHTML('afterbegin', html);

#### .prepend / .append

adds an element (message) as the first/last child of another element (header).

header.prepend(message);

header.append(message);

* In case both used, the last command will be applied. Node cannot be at two places at the same time. Fix:

header.append(message.cloneNode(true));

copies the messages element with all its children.

#### .before / .after

adds an element (message) as a sibling before/after another element (header).

Header.before(message);

Header.after(message);

#### .remove

Message.remove();

### Handling events

Consider checking MDN for all types of events.

#### click

listening for click events (happen on elements) – callback functions have access to the event

Document.querySelector(‘.check’).addEventListener(‘click’, function (e) {});

#### keypress

Listening for keypress events (happen on document object) – callback functions have access to the event

Document.addEventListener(‘keydown’, function (e) {})

Document.addEventListener(‘keyup’, function (e) {})

Document.addEventListener(‘keypress’, function (e) {})

Consider examining the e.key for keypress events.

#### Lifecycle DOM events

DOMContentLoaded: triggered by the document.

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

load: triggered by the window object.

window.addEventListener('load', function (e) {})

beforeunload: triggered by the window object.

window.addEventListener(beforeunload, function (e) {})

* Used with e.returnValue set to empty string to show a generic message before user leaves the webpage.

### Manipulating CSS style

Values should be written as strings.

document.querySelector(‘.number’).style.width = ‘30rem’;

* CSS properties with names consisting of 2 parts, should be written in camel-case.

document.querySelector(‘body’).style.backgroundColor = ‘#60b347’;

* With *style* we cannot read external CSS properties. It can only read inline styles. Fix below. Returns a CSS Style Declaration containing all the properties and their values.

Console.log(getComputedStyle(message));

#### Custom properties

CSS variables are defined in *root* selection in the CSS file (refer to HTML and CSS Essentials p 10), which is equivalent in JS to the Document. To access these properties, we should access the root section in CSS file first, then using the *style*, we can *setProperty*.

Document.documentElement.style.setProperty(‘—color-primary’, ‘orangered’);

### Reading and Setting Attributes

* When an HTML element is supposed to have certain attributes and if those attributes are defined in the HTML code, then they can be accessed directly (all possible to use for setting attributes):

Const logo = document.querySelector(‘.nav\_\_logo’);

Console.log(logo.src);

* Returns absolute URL. To get the relative URL use *getAttribute*.

Console.log(logo.className);

#### .getAttribute

If some attributes are defined in the HTML code but they are not supposed to be there:

logo.getAttribute(‘designer’)

#### .setAttribute

used to set some non-standard attributes. First the attribute title, then the attribute value.

Logo.setAttribute(‘company’, ‘Bankist’);

#### .dataset

used to create a bridge between the DOM and data of the application.

HTML attribute:

data-version-number=”3.0”

JavaScript:

logo.dataset.versionNumber

### Working with classes

Removing a class from element

document.querySelector(‘.modal’).classList.remove(‘hidden’);

Adding a class to element

document.querySelector(‘.modal’).classList.add(‘hidden’);

toggling a class for element

document.querySelector(‘.modal’).classList.toggle(‘hidden’);

Checking if a class exists in the class list:

modalWindow.classList.contains(‘hidden’);

### DOM Traversing

#### Going downwards (child)

* .querySelector (works on both the document and the elements)
* .querySelectorAll (works on both the document and the elements)
* .childNodes (used to get only the direct children nodes) – returns a node list.
* .children (used to get only the direct children elements) – returns an HTML collection
* .firstElementChild / .lastElementChild – returns element

#### Going upwards (parent)

* .parentNode (similar to *childnodes*) – returns ???
* .parentElement (used to get direct parent element)
* .closest (used to get a parent element which is not necessarily a direct parent) – returns a parent element matching the query no matter how far in the DOM tree. If query matches the element that closest is being called on, the element itself is returned.

#### Going sideways (sibling)

* .previousElementSibling / .nextElementSibling (used to access direct sibling elements)
* .previousSibling / .nextSibling (used to access direct sibling nodes)

### Targeted DOM updating

In order to avoid re-rendering all the DOM while only partial changes are about to happen, we can create a virtual DOM representing the new state of the DOM that is going to be rendered, but actually don’t render it. Then we compare this virtual DOM to the current DOM that is currently being displayed on the webpage. This comparison will lead to the replacement of the parts that are different between the two DOMs.

In order to create a new virtual DOM:

Const newDOM = document.createRange().createContextualFragment(<new-markup-string>);

Then we extract all the elements of this new virtual DOM and the current DOM and put them in separate arrays, so that we can compare them in the next step:

Const newElements = Array.from(newDOM.querySelectorAll(‘\*’));

Const curElements = Array.from(this.\_parentElement.querySelectorAll(‘\*’));

Then we loop over both arrays and compare elements of the two arrays. What we should look for are elements that are different and also elements that contain only text. If these two conditions are met, then we replace the text content of the current element with the text content of the virtual element.

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

const curEl = curElements[i];

if (!newEl.isEqualNode(curEl) && newEl.firstChild?.nodeValue.trim() !== ‘’) {

curEl.textContent = newEl.textContent;

}

})

This condition will target only elements that contain text. However, not all elements that need to be updated contain text. Some elements will need to update their attributes according to the new virtual DOM. In order to target these elements, we only need the first condition to be true. Then for these elements, we will get their attributes using the ‘attributes’ method on the element, which will return an object containing the attributes, then convert this object to an array on which we can loop and update attributes.

If (!newEl.isEqualNode(curEl)) {

Array.from(newEl.attributes).forEach(attr => curEl.setAttribute(attr.name, attr.value))

}

**NOTE | This algorithm is probably not a good way to implement targeted DOM updating in huge projects. In small projects it may work fine, but in general, it is not recommended to be followed.**

### Some Special Methods

#### getBoundingClientRect

used on an element to get a ton of information about the element’s position on the page) – returns a DOMRect.

#### window.pageXOffset and window.pageYOffset

used to get the current scroll position

#### window.scrollY and window.scrollX

used to get the current scroll position

#### clientWidth and clientHeight

used to get the viewport’s height and width.

document.documentElement.clientHeight;

document.documentElement.clientWidthl

#### scrollTo

used on the window obejct to make the browser scroll to a certain position using an object of options.

window.scrollTo({

    left: <value>,

    top: <value>,

    behavior: 'smooth',

  });

#### scrollIntoView

used on an element to make the browser scroll it into view using an object of options.

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

#### new IntersectionObserver

creating an observer

const observer = new IntersectionObserver();

call the observer to observe a target. Receives a callback function and an object of options.

const obsOptions = {

  root: null,

  threshold: 0.1,

  rootMargin: ‘-90px’,

};

* Root is the element that the target intersects with. Setting it to *null* will observe intersection with the viewport.
* Threshold: defines the percentage of intersection at which the callback function will actually be called back. Possible to mention multiple thresholds mentioned in an array.
* Rootmargin: defines some kind of an offset for the callback function to be called.

const obsCallback = function (entries, observer) { <code block> };

* Callback function has access to entries (data produced by the observer, containing information about the target that is intersecting), and to the observer itself.

Calling the observer to observe the target:

observer.observe(<target element>)

#### focus / blur

used to activate / deactivate the cursor in a textbox input in certain conditions.

inputDistance.focus();

inputDistance.blur();

#### Geolocation API

Getting current position

navigator.geolocation.getCurrentPosition(<success function> (position) {}, <failure function>() {});

* Returns a GeolocationPosition object containing altitude, longitude, and many other properties.

#### LocalStorage API

Setting data into the localStorage:

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

* JSON.stringify is only necessary if the value that is being stored is an object.

Getting data out of the localStorage:

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

* JSON.parse is always necessary because what we receive directly from the localStorage is in JSON format and is not usable.

Deleting data from the localStorage:

localStorage.removeItem('workouts');

## 007 Strings

Retrieving characters at a certain index

const plane = 'A320';

const firstLetter = plane[0];

const firstLetter2 = “A320”[0];

### String methods:

* .length
* .indexOf (search for the index of the first occurrence of a certain character)
* .lastIndexOf
* .slice (returns an extract of the string starting from a certain index until one less than the second index) – negative numbers will count from the end.

airline.slice(4, 7)

* .toLowerCase
* .toUpperCase
* .trim (deletes any white space)
* .replace

const priceUS = priceGB.replace('&', '$')

* .replaceAll
* .includes
* .startsWith
* .endsWith
* .split (to split a string into multiple parts based on a divider string.

'a+very+nice+string'.split('+')

* Join (joins multiple strings in an array into a whole string with a certain character between them)

['Mr.', firstName, lastName.toUpperCase()].join(' ')

* .padStart (adds a certain character to the beginning of a string to make its size equal to a certain amount)

message.padStart(25, '+')

* .padEnd
* .repeat (repeats a string for a certain amount)

message2.repeat(5)

## 008 Numbers and Dates

### Numbers

Converting string to number

Number('23')

#### Methods:

The Number object provides a namespace containing some methods:

* Number.parseInt (used to separate a number from a string containing a number and after that, other characters probably as units.) – returns integer number with no decimals

Number.parseInt('30px', <radix>)

Radix is the base of numeral system which, by default, is 10.

* Number.parseFloat (similar to *parseInt*. Usually used to read a value out of a CSS string) – returns floating number
* Number.isNaN (used to check if a value is Not-a-Number. It may have confusing results. Consider using *isFinite*) – returns Boolean.
* Number.isFinite (used to check if a value is a number) – returns Boolean.
* Number.isInteger (used to check if a value is integer and not a float.

The Math object provides a namespace for some methods:

* Math.sqrt (used to calculate the square root of a number)
* Math.max (used to get the maximum value among multiple numbers. involves type coercion, but no parsing.)

Math.max(5, 18, 23, 11, 2)

* Math.PI (the PI constant related to the calculation of area of a circle)
* Math.random – returns a random value between 0 and 1. To create a random number between two certain numbers:

const randomInt = (min, max) => Math.trunc(Math.random() \* (max - min) + 1) + min;

* Math.trunc (removes any decimals, involves type coercion)
* Math.round (rounds a floating number to its nearest integer, involves type coercion)
* Math.ceil (rounds a floating number to its higher integer, involves type coercion)
* Math.floor (rounds a floating number to its lower integer, involves type coercion)

Methods to be used on a number:

* .toFixed (receives a number which defines the number of decimals. If set to 0, works like *round*. If set to a certain number, will *round* or add 0s to match the number of decimals with that certain number)

#### Internationalizing Numbers (Intl)

const num = 3885843.23;

displaying the number with internationalization formatting according to an object of options:

new Intl.NumberFormat('en-US', options).format(num)

const options = {

  style: 'unit',

  unit: 'mile-per-hour',

  useGrouping: false,

};

* specify first the style inside this object of options. There are three different values that we can put here: **unit**, **percent**, **currency**. When we set the style to unit, we then have to define the unit. When we set the style to percent, the unit property is completely ignored. When we set the style to currency, the unit property is ignored, and we will have to define the currency. Consider checking MDN for all styles and units available.

### Date and Time

Creating a date – returns current date and time:

const now = new Date();

Parsing a date from a date string (the string can either be written by ourselves or fetched from somewhere else):

new Date('December 24, 2015')

new Date('Aug 09 2022 22:23:14')

Parsing a date from a series of numbers defining year, month (zero-based), day, weekday, hours, minutes, and seconds:

new Date(2037, 10, 19, 15, 23, 5)

Beginning of Unix time:

console.log(new Date(0));

the Date object provides a namespace for some methods:

const future = new Date(2037, 10, 19, 15, 23);

* .getFullYear (also have set)
* .getMonth (also have set)
* .getDate (day of the month, also have set)
* .getDay (day of the week, also have set)
* .toISOString (converts the date to ISO string)
* .getTime (acquire amount of milliseconds passed since the beginning of Unix time)
* Date.now (timestamp for now)

#### Calculations with dates:

when we attempt to convert a date to a number, then the result is going to be the timestamp in milliseconds. With this milliseconds we can then perform calculations (e.g. subtract to measure the distance between two dates).

#### Internationalizing dates (Intl)

const now = new Date();

displaying the current date with internationalization formatting according to an object of options:

new Intl.DateTimeFormat(<locale>, options).format(now);

const options = {

  hour: 'numeric',

  minute: 'numeric',

  day: 'numeric',

  month: 'long', | ‘2-digit’

  year: 'numeric', ‘2-digit’

  weekday: 'short', | ‘long’

};

Getting the locale information from the client, instead of hard-coding:

const locale = navigator.language;

#### Timers:

Timeout (used to execute a callback function after a certain amount of time in milliseconds.)

setTimeout(() => console.log('Here is your pizza'), 3000);

* If the callback function requires arguments, we can determine them after defining the delay.
* Possible to cancel the timeout:

const pizzaTimer = setTimeout(<fn>, <delay>, <arguments>);

clearTimeout(pizzaTimer);

Interval (used to execute a callback function every certain amount of time passed)

setInterval((<fn>, <interval>);

## 009 OOP

#### ES6 classes

Creating a parent class

class CarCl {

  constructor(*make*, *speed*) {

    this.make = make;

    this.speed = speed;

  }

  accelerate() {

    this.speed += 10;

  }

  break() {

    this.speed -= 5;

  }

  get speedUS() {

    return this.speed / 1.6;

  }

  set speedUS(*speed*) {

    this.speed = speed \* 1.6;

  }

}

Creating a child class:

class EVCl extends CarCl {

  #charge;

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

    super(make, speed);

    this.#charge = charge;

  }

  chargeBattery(*chargeTo*) {

    this.#charge += chargeTo;

    return this;

  }

  accelerate() {

    this.speed += 20;

    this.#charge -= 1;

    return this;

  }

}

Creaing an instance:

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

#### Constructor function

Creating a parent class.

const Car = function (*make*, *speed*) {

  this.make = make;

  this.speed = speed;

};

Methods are then inserted in the prototype of this class.

Car.prototype.accelerate = function () {

  this.speed = this.speed + 10;

};

Car.prototype.break = function () {

  this.speed = this.speed - 5;

};

Now we have a child class which is called EV.

const EV = function (*make*, *speed*, *charge*) {

  Car.call(this, make, speed);

  this.charge = charge;

};

But until now the prototype of this child class is not connected to its parent class. So we should connect them:

EV.prototype = Object.create(Car.prototype);

Methods for the child class should be implemented as before:

EV.prototype.chargeTo = function (*chargeTo*) {

  this.charge = chargeTo;

};

EV.prototype.accelerate = function () {

  this.charge--;

  this.speed += 20;

  console.log(

    `${this.make} going at ${this.speed} km/h, with a charge of ${this.charge}%`

  );

};

And now we want to create an instance of the EV.

const tesla = new EV('Tesla', 120, 23);

#### Object.create

We start by manually creating prototype objects. These will act as classes.

const PersonProto = {

  clacAge() {

    console.log(2037 - this.birthYear);

  },

  init(*firstName*, *birthYear*) {

    this.firstName = firstName;

    this.birthYear = birthYear;

  },

};

createing another prototype object which would act as the child of the PersonProto.

const StudentProto = Object.create(PersonProto);

We now have created an empty object called StudentProto and connected its prototype to the PersonProto object. Now we to fill this child class with methods:

StudentProto.init = function (*firstName*, *birthYear*, *course*) {

  PersonProto.init.call(this, firstName, birthYear);

this.course = course;

};

StudentProto.introduce = function () {

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

};

finally, we want to create an instance of the StudentProto.

const jay = Object.create(StudentProto);

So now the prototype chain starts from StudentProto, continues to PersonProto, and eventually reaches the Object prototype.

## 010 Asynchronous JavaScript

### XMLHttpRequest method

Calling the XMLHttpRequest function. Then call the open method on that variable. In the open method we need to specify what should be done and we also need to specify the URL to which we make the AJAX call. Then we should send the request.

const request = new XMLHttpRequest();

request.open('GET', 'https://restcountries.com/v3.1/name/iran');

request.send();

now the request is sent and is being processed asynchronously. To get the result we should listen for the load event on the request. After the result has arrived, we can access the *responseText* property on the request. However, the *responseText* is in JSON format and we need to convert it into a JS object with *JSON.parse*.

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

const data = JSON.parse(this.responseText);

});

### Fetch API & Promises

Starting a request by fetching some data.

const request = fetch('https://restcountries.com/v3.1/name/portugal');

fetch will immediately return a pending promise, and the request will be processed asynchronously. To consume the promise, we use the then method. This Then method receives the response to the request. This response should be returned after being converted using *.json()* method. The then method will return a promise, and this promise should again be consumed with another then method.

* Then methods can receive a second argument after the callback function. The second argument is a callback function that will be executed if the promise is rejected.

request.then(response => response.json()).then(data => { <code block> });

Now we have access to the data we want. On this then, we can call the *catch* method to catch any error happened during the request processing. All the errors during the process will propagate down and arrive at the *catch* method.

.catch(err => alert(err.message));

* The *err* is actually an object which we can also access its *message* property.
* The only way that the fetch promise is rejected, is when the user loses internet connection. But invalid queries will simply return 404 error and does not reject the promise. They actually fulfill the promise. So to display errors for invalid queries we should *throw* errors manually.

const getCountryData = function (country) {

  fetch(`https://restcountries.com/v3.1/name/${country}`)

    .then(response => {

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

      return response.json();

    })

### Creating a promise manually

Creating a new promise and set it to a variable:

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

  if (Math.random() >= 0.5) {

    resolve('you WIN!!!');

  } else {

    reject('You LOST your money!!!');

  }

});

Consuming the promise with then and catch:

lotteryPromise

  .then(response => console.log(response))

  .catch(err => console.error(err));

* In practice, most of the time we only consume promises. We usually only build promises manually to wrap old callback-based functions into promises. This is called promisifying.

### Consuming promises with Async/Await

const whereAmI = async function (country) {

  const response = await fetch(

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

  );

  const data = await response.json();

return data;

};

* This async function which acts as a promise, can be consumed using the then and catch methods again. But it can also be further consumed with async/await again, and this is the better way to go.

### Try/Catch

This is not exclusive to asynchronous JavaScript. It can be used anywhere, but one of the most important use cases is asynchronous JavaScript.

const whereAmI = async function () {

  try {

    //  All the code

  } catch (err) {

    // Error handling code

  }

};

* Consider re-throwing errors if necessary. Refer to the main document p388.

### Promise combinators

* Promise.all (used to run multiple promises simultaneously. Receives an array of promises. If one promise rejects, the whole promise rejects or short circuits) – returns an array containing the data received in response for each promise.

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

  try {

    const data = await Promise.all([

      getJSON('url1'),

      getJSON('url2'),

      getJSON('url3'),

    ]);

    console.log(data);

  } catch (err) {

    console.log(err);

  }

};

* Promise.race (used to run multiple promises simultaneously. Receives an array of promises.) – returns an array containing only the first promise that was settled, either rejected or fulfilled. Use when you need to implement request timeout mechanisms.
* Proimse.allSettled (used to run multiple promises simultaneously. Receives an array of promises.) – returns an array containing all the settled promises promise, either rejected or fulfilled. A bit like *Promise.all*, but this method never short circuits.
* Promise.any (used to run multiple promises simultaneously. Receives an array of promises.) – returns an array containing only the first fulfilled promise. A bit similar to *race*.

## 011 Modules in JavaScript

To make the script file work as a module, we have to set the *type* attribute of the script tag to *module* in HTML code.

### Exporting and Importing in ES6 Modules

Importing a module without importing any value

import './shoppingCart.js';

Now let’s define some variables in the shoppingCart.js module.

const shippingCost = 10;

const cart = [];

* top-level variables are scoped to the module in which they are defined. They are not scoped globally. To make these variables accessible in the main module, we have to export these variables from this module.

#### Named exports

We export a variable or function or anything from this module.

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

Then we import it in the main module:

import { addToCart } from './shoppingCart.js';

We can export multiple things from a module using named exports.

const totalPrice = 237;

const totalQuantity = 23;

export { totalPrice, totalQuantity };

Then we import them in the main module:

import { addToCart, totalPrice, totalQuantity } from './shoppingCart.js';

we can change variable names while importing or exporting:

import {totalPrice as price,  totalQuantity,} from './shoppingCart.js';

export { totalPrice, totalQuantity as qt };

we can import all the exports of a module at the same time:

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

then we can access anything inside the exporting module from the importing module:

ShoppingCart.addToCart('bread', 5);

* consider similarities with classes.

#### Default exports

Used when we want to export one thing per module. These exports don’t have any name.

export default function (product, quantity) {};

We give it a name when importing it:

import add from './shoppingCart.js'

* It is not a good practice to mix default and named imports.

### Publisher-Subscriber pattern

The publisher-subscriber pattern is used to handle events in the context of a MVC-structured code base. This involves implementing a publisher in the View code, which is normally called by an Initializer function. The Initializer function passes a Subscriber function into the Publisher function. The Subscriber function should be defined inside the Controller file.

**View file:**

addHandler(handler) {

window.addEventListener(‘<event>’, <callback-function>

}

This publisher function is usually exported as a method of an object’s publish API.

**Controller file:**

Const init = function() {

<view-file-import>.addHandler(<subscriber-function>)

}

init();

## 012 Introduction to the Command Line

To see the directory of the current folder:

dir

To go up one step in the directory:

cd ..

to go up two steps in the directory:

cd ../..

to go down one step in the directory (use tab to auto-complete file name:

cd filename

to clear the terminal console:

clear

to create a folder:

mkdir foldername

to create a file:

ni filename

to create multiple files in one line of command:

ni filename1, filename2, filename3

to delete a file:

del filename1

to delete multiple files:

del filename1, filename2

to move a file to the parent folder of the current folder:

mv filename ../

to remove a folder:

rmdir foldername

## 013 Working with NPM

In order to use NPM we should follow these steps:

1. Install NPM from URL: nodejs.org/en/ (you can check if any version of NPM is already installed using this command in terminal:

npm –v

1. In each project where we need to use NPM we need to start by initializing it using this command in terminal:

npm init

this will prompt the coder with multiple question in order to create and populate the **package.json** file.

1. Install any library you need to use in your application. You can search for all available libraries (e.g. leaflet, lodash, parcel, etc.). you can find the installation command on the library’s website. For example, for installing leaflet we can use this command in terminal:

npm install leaflet

or

npm install lodash-es

after installing the library, some changes happen. First, in the package.json file a **dependency** will be added with the library version inside it. Second, in our project folder, a new folder called **node\_modules** will be created. This folder contains the folder for the installed library. Note that some libraries use CommonJS. We cannot directly import these libraries into our code. We can only do that if we use a module bundler later.

It is important to note that if you want to move your project to somewhere else or share it with other developers, you don’t need to include the node\_modules folder. You can re-install all your libraries again using this command in the terminal:

npm install

this will enable NPM to reach into the package.json file, read the dependencies and install them back.

## 014 Working with Parcel

To use Parcel with NPM we should follow these steps:

1. Install Parcel using NPM with this command in the terminal:

npm install parcel --save-dev

a dev dependency is like a tool that we need to **build** our application, but it is not a dependency that we include in our code. So this will appear in a devDependencey field in package.json file.

1. Run Parcel using 2 different ways: run Parcel using this command for local installation:

npx parcel index.html

index.html is the entry point. The entry point is the file in which we include script.js file. This will establish a **development server** usually on localhost:1234. In addition, 2 new folders will be created in our project folder: **dist** and **.cache**. The dist folder contains the new index.html and all css and script files. At this stage, these are not compressed files and also dead codes are not eliminated yet. This will be carried out in the **build** step. However, it is important to note that having the parcel started at this stage, we no longer need to mention the long source path for imports on other libraries. For example, an import like this:

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

will turn into this:

import cloneDeep from ‘lodash-es’;

now parcel will automatically find the path to this module and import it without us having to manually type the entire path.

Another way to run parcel is to insert a “start” field inside the “script” field of the package.json file:

“script”: {

“start”: “parcel index.html”

}

Then in the terminal we can use this command:

npm run start

we run parcel whenever we want to start developing. But when we are done developing, we finally need to build the final bundle which is compressed and has dead code elimination.

1. Build the final bundle by inserting a “build” filed inside the “scripts” field of package.json file.

“script”: {

“start”: “parcel index.html”,

“build”: “parcel build index.html”

}

Then in the terminal we can use this command:

npm run build

now the files inside the dist folder are fully compressed.

**IMPORTANT:** you need to tell parcel to ignore the “main” field of the package.json file by inserting a “targets” field:

“targets”: {

“main”: false

}

This solution is suggested here: [link](https://github.com/parcel-bundler/parcel/issues/3500)

The final result of this command is the dist folder and we can send this folder for production.

## 015 Working with Babel and Polyfilling

Parcel uses babel automatically to transpile your code. Babel can convert new codes into codes that can be executed on older browsers. To decide which codes should be converted, babel uses presets. By default, it uses preset-env. This will automatically select which JavaScript features should be compiled based on browser support. However, we should note that transpiling is only done for syntaxes, not features. for example, ES6 features like Promise or array methods like find, cannot be transpiled. But arrow functions, as a syntax, can be transpiled into regular function syntax. But for features, we can use polyfilling.

When we run the npm start command, the babel will automatically be used for transpiling the codes with some default settings. These settings are editable.

To enable polyfilling we should first install it using this command in the terminal:

npm install core-js

then in the script file we should import it:

import ‘core-js/stable’;

now if we look at the output code, we don’t see any visible conversion, because polyfilling is not supposed to work that way. Instead, it recreates the features and enable them to be used in the code. Polyfilling will polyfill everything even if we don’t need it. we could also selectively import features for polyfilling like this:

import ‘core-js/stable/array/find’;

this will reduce the bundle size a lot and it can be done if it is a real concern.

There is one more feature that is not polyfilled by core-js, and that is async functions. So we usually need to install another package in the terminal:

npm install regenerator-runtime

and then we need to import it in the script file:

import ‘regenerator-runtime/runtime’;

**IMPORTANT:** all of these procedures might change by time. So you always need to update your technical knowledge about using these packages.

## 016 Working with Git

To work with git you should follow these steps:

1. Download and install Git from its website: [link](https://git-scm.com/downloads)
2. Make sure that in your terminal you are in your project folder, and not in its child folders.
3. Initialize git using this command in the terminal:

git init

this will initialize an empty git repository in the project folder. Repository is the fundamental concept of git. This will mark all of our code files with “U” which stands for Untracked.

1. Create a .gitignore file in your project folder. In this folder you can set some files and folders in your project folder to be ignored in commitments. Any folder name listed in this file will be un-highlighted in the folders pane on the left. List file names just like this:

node\_modules

dist

.parcel-cache

1. If you have an account on github you are good to go, But if not, create one. Github enables you to store your local repository on the cloud so that you will be able to switch between computers while making sure you don’t lose your codes.
2. Connect your local git installation with your github account. To do this you should first determine your username on github:

git config --global user.name omidarmat

and then you should determine your user email on github:

git config --global user.email [omidarmat@gmail.com](mailto:omidarmat@gmail.com)

remember that these configurations will always be used in all the repositories that you create on your computer.

Now let’s learn a bit about git fundamentals:

* Check the status of commitments:

git status

this will show us the untracked files and check if there are any commits. We can also see the branch that we are currently in.

* Track files for any changes. It means to add them to the staging area:

git add –A

-A means all the files. We can add files manually, but usually that is not necessary. Now all files will be marked with ‘A’ which stands for Active. From this point, any change implemented in the code will be tracked. Added codes will be marked in green, deleted codes will be marked in red, changed codes will be marked in orange. Additionally, files that their code is changed will be marked with ‘M’ standing for ‘Modified’.

* Commit files: commit is like a snapshot of your code at a certain point in time. So before making some significant changes to your code, you should always commit. Then if necessary, you can go back to past commits and undo your changes. To commit:

git commit –m ‘<commit message>’

after commit, if we check the status, we will see that the working tree is clean, meaning that there is nothing to commit.

* To go back to the previous commit before committing newly modified files:

git reset --hard HEAD

* To go back to the previous commit after committing newly modified files: we first need to find the id of the commit to which we want to return. So we should take a look at the previous commits:

git log

this will give all the commits with their id. We can copy this id and use it in this command:

git reset --hard <commit id>

**IMPORTANT:** moving between commits like this is a bit dangerous. We usually create new branches when we want to implement a lot of changes.

* To see the list of branches that we have:

git branch

* To create a new branch

git branch <branch name>

we now have to switch to this new branch. Otherwise, we will remain in the previous branch.

git checkout <branch name>

this new branch is a copy of the current master branch in which we can develop new code and add new features without affecting the code in the master branch.

Now if we go back to the master branch using the checkout command, we see the previous version of code.

* We can now merge the new branch with the master branch. So inside the master branch, we use this command:

git merge <branch name>

**IMPORTANT:** there is a nice cheat sheet for git here: [link](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjrksv7uJX9AhVMhv0HHW4dBWsQFnoECA4QAQ&url=https%3A%2F%2Feducation.github.com%2Fgit-cheat-sheet-education.pdf&usg=AOvVaw2D3W2R0fwoOBi8YrhZYLFJ)

#### Push to Github

We first need to create a repository in the github. Once the repository is created, a URL address to the repository will be shown along with different scenarios and their related codes. We can copy and paste these codes in our terminal.

This is the first line of code that should be used in terminal for importing commits into github. This basically connects Git to your GitHub repo:

git remote add origin https://github.com/omidarmat/390calculationspanel.git

then to push a branch into the origin branch of github we can use this command in the terminal:

git push origin <branch name>

we can create a readme file with this command in the terminal:

ni README.md

then by committing and pushing it to github, we will see all the explanations in this file in the github repository.

We usually carry out these steps the other way round. That means that we usually create an empty repository in github with a README file and a .gitignore file. Then we pull this repository onto our computer. This makes things a bit easier.

#### continuous integration with Netlify

On Netlify, you can link an existing website to git, or you can link a website right from the beginning. We can simply select the “New site from Git” option on Netlify. But if there is an existing website, we can go to its settings on netlify, then in the “build and deployment” tab, we can link the site to Git.