Javascript Theory

High-Level Overview of JS

- High-Level, Object-Oriented Multi-Paradigm Language
- Low-Level - Manage resources Manually, High - Level - Manage resources automatically
- ▼ Garbage Collection - what does it do?
  - ▼ Cleans the memory so we don't have to
- ▼ Interpreted vs just-in-time compiler?
  - ▼ todo
- ▼ Multi-paradigm
  - ▼ An approach and overall mindset of structuring code, which will direct your coding style and technique
  - ▼ Procedural programming - linear organization
  - ▼ Object Oriented Programming
  - ▼ Functional programming
- ▼ Prototype-based object-oriented
  - ▼ More on this later =)
- ▼ First Class Functions
  - ▼ Functions are simply treated as variables
- ▼ Dynamic
  - ▼ No data type definitions - types become known at runtime
  - ▼ Data type of variable is automatically changed
- ▼ Single-Threaded / Non-blocking event loop
  - ▼ Concurrency model - how the JS model handles multiple tasks happening at the same time
  - ▼ **JS runs on a Single-Thread.**
  - ▼ Event loops prevent blocking on a single thread - such as with retrieval of data from a server (we need to wait for response back from server)

JS Engine and Runtime

- JS Engine - Program that executes JS Code.
- Heap - where objects are stored (objects in memory)
- Call Stack - where our code is executed (execution context)
- ▼ Compilation vs Interpretation
  - ▼ Compilation: Entire code is converted into machine code at once, and written to a binary file that can be executed by a computer
  - ▼ Interpretation: Interpreter runs through the source code and executes it line by linw
  - ▼ Just In Time Compilation - Entire code is converted into machine code at once, then executed immediately.
    - ▼ Parsing → compilation → execution → optimization → compilation (and it repeats)
- ▼ JS Engine, WEB Apis, and Callback Queue for all callback functions that are ready to be executed

Execution Context

- ▼ Compilation
  - ▼ Creation of global execution context (for top-level code) (not inside a function code)
  - ▼ Execution of top-level code inside the execution context
  - ▼ Execution of functions and waiting for callbacks

- ▼ What's Inside a Execution Context?

  let, const, var declarations

  functions

  arguments objects (not in arrow func)

  scope chain

  this key word (not in arrow func)

- ▼ What is the call stack?

  place where EC gets stacked on top of each in order to keep track of where we are at in the program

  When we call a function, we put it in the call stack. It becomes a new execution context.

  When we hit the return keyword or the end of the function, the execution context is removed from the call stack.

- ▼ Scoping and Scope In Javascript: Concepts
  - Scoping - How our program's variables are organized and accessed.
    - Where can a variable be accessed?
  - Lexical Scoping - Scoping is controlled by placement of functions and blocks in the code
  - ▼ 3 types of scopes
    - ▼ Global Scope

- Outside of any function or block, accessible everywhere

▼ Function Scope

- Only accessible inside function, not outside
- this is var → functions defined inside a block i.e an if statement but not inside a function are accessible at the global level. In other words, it is the function that defines the scope

▼ Block Scope

- VAriables are only accessible inside a block
- let and const only - var not affected

- The scope chain has nothing to do with the order in which functions are called

Environment and the TDZ

Hoisting - Makes some types of variables usable in the code before they are actually declared.

Before execution, code is scanned for variable declarations, and for each variable, a new property is created in the variable environment object.

Scoping

Name	Hoisted?	Initial Value	Scope
function declarations	Yes	Actual Function (in strict mode)	Block
var variables	Yes, and creates a window object - (window.nameOfVar)	undefined	Function
let and const	No	uninitialized	Block
function expressions and arrows	Depends on if using var or let/const	Depends on if using var or let/const	Depends on if using var or let/const
Function using function keyword	yes	User-Defined	Global
Untitled			

Example - Hoisting Bug

```
if(!numProducts) deleteShoppingCart;

var numProducts = 10;

function deleteShoppingCart() {
  console.log('Deleted');
}

//DeleteShoppingCart is called. This is b/c when hoisted,
//our var is set to undefined by default.
```

This keyword

- ▼ This keyword - Special variable that is created for every execution context (every function).
  - ▼ Takes the value of the owner of the function in which the **this** keyword is used.

- **This** is not static. IT depends on how the function is called, and its value is only assigned when the function is actually called.
- ▼ Types of this
  - ▼ Method: this → object that is calling the method
  - ▼ Simple function call →this is undefined
  - ▼ arrow function → this of surrounding function (Lexical this)
  - ▼ Event Listener →Dom element that the handler is attached to
  - ▼ new, call, apply, bind → revise later in course

Arrow Functions and This

```
//Cool arrow function notes - this keyword scope
let me = {
  name: 'Ashutosh Verma',
  thisInArrow: () => {
    console.log('My name is ' + this.name); // no 'this' binding here
  },
  thisInRegular() {
    console.log('My name is ' + this.name); // 'this' binding works here
  },
};
me.thisInArrow();
me.thisInRegular();
```

Primitives vs Objects

- Primitives → Call stack
- Objects → Heap

```
//Primitive values
let age = 30; //memory address of 0001
let oldage = age; //memory address points to 0001
age = 31; //value at memory address cannot be changed, new piece of memory
//allocated, points to 0002
console.log('Age at age);
console.log(oldage);

//Reference values
const me = {
  name: 'jonas',
  age: 30,
};

const friend = me;
friend.age = 27;

console.log(`Friend: ${friend.age}`);
console.log(`Me: ${me.age}`);

//me -> 0003 -> points to memory in heap (D304), where data for object is stored
//friend -> 0003 -> points to memory in heap (D304)
//When you change age, change value in heap. So this changes age for both friend
//and me
```

```
const jessica = {
firstname: 'jess',
lastname: 'williams',
}
const marriedJessica = jessica;
marriedJessica.lastname = 'Davis';
console.log('Before Marriage:', jessica);
console.log('After marriage:', marriedJessica);

//to assign to empty


//copying objects
const jessica2 = {
firstname: 'jess',
lastname: 'williams',
}

//creates shallow copy
//copy contents of jessica2 to empty object, and assign it to jessicaCopy
const jessicaCopy = Object.assign({}, jessica2);
jessicaCopy.lastName = 'Davis'
```

Destructure Arrays

Our Starter code is as follows:

```
'use strict';

const restaurant = {
  name: 'Classico Italiano',
  location: 'Via Angelo Tavanti 23, Firenze, Italy',
  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],
  starterMenu: ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad'],
  mainMenu: ['Pizza', 'Pasta', 'Risotto'],

  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },
};
```

Array Destructuring

```
let [main, , secondary] = restaurant.categories;
console.log(main, secondary);

//swap main and secondary
const temp = main;
main = secondary;
secondary = temp;
console.log(main, secondary);
//or
[main, secondary] = [secondary, main];
console.log(main, secondary);

//destructing
const [starter, mainCourse] = restaurant.order(2, 0);
```

```
console.log(starter, mainCourse);

//nested array destructuring
const nested = [2, 4, [5, 6]];
//EXAMPLE 2
//HANDLING NESTED ARRAYS
const entryPlayers = Object.entries(game.players);
console.log(entryPlayers);
const [firstRoster, [, rosterNames]] = entryPlayers;
console.log(rosterNames);

//const [x, , y] = nested;
//console.log(x, y); //2 [5, 6]
const [i, , [j, k]] = nested;
console.log(i, j, k); //2 5 6

//Default Values
const [x = null, y = null, z = null] = [8, 9]; // -> [8,9, undefined]
console.log(x, y, z);
```

Object Destructuring

```
const {name, openingHours, categories} = restaurant; //Italian, Pizzeria, Vegetarian,
//Organic
console.log(name, openingHours, categories);
const {name : newName, openingHours: hours, categories: tags) = restaurant;

//for handling empties
const {menu = [], starterMenu: starters = []} = restaurants;

//selecting from object
let a = 3;
let b = 2;
const obj = {a: 23, b: 7, c : 45};
({a, b} = obj);

//nested objects
const { sat } = openingHours
```

```
const {name, openingHours, categories} = restaurant; //Italian, Pizzeria, Vegetarian,
//Organic
console.log(name, openingHours, categories);
const {name : newName, openingHours: hours, categories: tags) = restaurant;

//for handling empties
const {menu = [], starterMenu: starters = []} = restaurants;

//selecting from object
let a = 3;
let b = 2;
const obj = {a: 23, b: 7, c : 45};
({a, b} = obj);

//nested objects
const { fri: {open, close} } = openingHours;
//to rename while accessing nested object at same time
const { fri: Friday, fri: {open, close} } = openingHours;

//destructuring in function headers
orderDelivery: function ({starterIndex, mainIndex, time, address}){}
```

JS Spread Operator

```
const arr = [7,8,9]
const arrCopy = ...arr;

onst arr = [7, 8, 9];
const arrCopy = [1, 2, ...arr];
```

```
console.log(arrCopy);
//logs/passes in individual  elements of the array
console.log(...arrCopy);

//new element - create new array
const newMenu = [...restaurant.mainMenu, 'Gnocci'];
console.log(newMenu);

//restaurant.mainMenu = newMenu;
//console.log(restaurant.mainMenu);

//Use Case 1 - copy array
const mainMenuCopy = [...restaurant.mainMenu];

//use case 2 - join 2 arrays
const combinedMenu = [...restaurant.starterMenu, ...restaurant.mainMenu];
console.log(combinedMenu);

//Spread Operator works on all iterables -> arrays, maps, sets, strings, but not objects.
const str = 'Jonas';
const letters = [...str, ' ', 'S.'];
console.log(...letters);

// to escape -> Let\'s make pasta!"

//can also do spread operators for function arguments

// Objects -> cool shit
const newRestaurant = {foundIn: 1998, ...restaurant, founder: "PiccoloDick"};
//object deep copy
const restCopy = {...restauraunt};
```

Rest Pattern and Parameters

```
//Spread -> Because on Right side is the =
const arr = [1,2, ...[3,4]];
const [a,b, ...others] = [1,2,3,4,5]

//rest element must be last element to work
const [pizza, , risotto, ...otherItems] = [
  ...restaurant.mainMenu,
  ...restaurant.starterMenu,
];

console.log(pizza, risotto, otherItems); //-> returns Pizza Risotto (4) ["Focaccia", "Bruschetta", "Garlic Bread", "Caprese Salad"]



// Functions example
//The advantage of doing it this way is you can pass in multiple numbers, or you can
//pass in an array (IMPORTANT - if you pass in array, need to use spread operator first)
const add = function (...numbers) {
  let sum = 0;
  for (let i = 0; i < numbers.length; i++) sum += numbers[i];
  console.log(sum);
};

add(2,3)
add(2,3,4,5,6,6)
add(1,2,3,4,5,6,76,7,8);

//compress
const x=[23,5,7]
add(...x);
///////////////////////
```

Short Circuiting

```
//Use any data type, return any data type,
//short-circuiting
console.log(3 || 'Elijah');
console.log('' || 'Jonas');

console.log(undefined) //false

console.log('Hello' && 23 && null && 'jonas');

//practical example
if(restaurant.orderPizza){
restaurant.orderPizza('mushrooms', 'spinach');
}

//check for existence before running func
if(restauraunt.orderPizza)
  restaurant.orderPizza()

//or

if(restaurant.orderPizza && restaurant.orderPizza())
```

Nullish Coalescing System

```
rest.numGuests = 0;
const guests = restaurant.numGuests || 10;
//this will print 10

//Nullish - null and undefined (NOT 0 or '')
const guessCorrect = restaurant.numGuests ?? 10;
//this will print 0
```

For of loop

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

//for of loop
for (const item of menu) {
  console.log(item);
}

//Getting index using for of
for (const [i, el] of menu.entries()) {
  console.log(i, el);
}
```

Optional Chaining

```
if (restaurant.openingHours && restaurant.openingHours.mon)
  console.log(restaurant.openingHours.mon.open);

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

// WITH optional chaining - tests if the value exists
//if monday exists, then open will be read
console.log(restaurant.openingHours.mon?.open);
//if opening hours exists, and then if monday exists
console.log(restaurant.openingHours?.mon?.open);

// Example
const days = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'];

for (const day of days) {
  const open = restaurant.openingHours[day]?.open ?? 'closed';
  console.log(`On ${day}, we open at ${open}`);
}

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

```
// Arrays
const users = [{ name: 'Jonas', email: 'hello@jonas.io' }];
// const users = [];

console.log(users[0]?.name ?? 'User array empty');

if (users.length > 0) console.log(users[0].name);
else console.log('user array empty');
```

Looping over Objects

```
///////////////////////////////////////
// Looping Objects: Object Keys, Values, and Entries

// Property NAMES
const properties = Object.keys(openingHours);
console.log(properties);

let openStr = `We are open on ${properties.length} days: `;
for (const day of properties) {
  openStr += `${day}, `;
}
console.log(openStr);

// Property VALUES
const values = Object.values(openingHours);
console.log(values);

// Entire object
const entries = Object.entries(openingHours);
// console.log(entries);

// [key, value]
for (const [day, { open, close }] of entries) {
  console.log(`On ${day} we open at ${open} and close at ${close}`);
}
```

Bracket Notation vs Dot Notation

```
const elijah = {
  firstName: 'Elijah',
  lastName: 'Gaytan'
  age: 2021-1995,
}

//Both return the same thing...
console.log(jonas['lastName'])
console.log(jonas.lastName);

Exceptions:
const NameKey = 'Name';
console.log(jonas['first' + NameKey]);
console.log(jonas['last' + NameKey]);

Dot notation can only access the property directly.
```

We can use a variable to access object properties.

```
const chooseProp = prompt('Enter property');
//returns undefined
console.log(elijah.chooseProp);
//returns the object property or the string we specified
console.log(elijah[chooseProp] ?? `That value doesn't exist.`);
```

Coding Challenge 1

```
//1.
const [players1, players2] = game.players;
console.log(players1, players2);

//2
const [gk, ...fieldplayers] = players1;
console.log(gk, fieldplayers);

//3
const allPlayers = [...players1, ...players2];
console.log(allPlayers);

//4
const players1Final = [...players1, 'Thiago', 'Coutinho', 'Perisic'];
console.log(players1Final);

//5
const { team1, x: draw, team2 } = game.odds;
console.log(team1, draw, team2);

//6 ->
const printGoals = function (...players) {
  players.forEach(element => {
    console.log(element);
  });
  console.log(players.length);
};

printGoals(...game.scored);

//7:
team1 > team2 && console.log(team1);
team2 > team1 && console.log(team2);
*/
```

Coding challenge 2

```
//1.
for (const [index, name] of game.scored.entries()) {
  console.log(`Goal ${index + 1}: ${name}`);
}

//2.
const odds = Object.values(game.odds);
let sumOdds = 0;
console.log(odds);
for (const odd of odds) {
  sumOdds += odd;
}
console.log(sumOdds / odds.length);

//3

//mine
/*
const {
  team1,
  team2,
  odds: { team1: team1Odds, x: xOdds, team2: team2Odds },
} = game;

console.log(`Odd of victory of ${team1}: ${team1Odds}`);
console.log(`Odd of draw: ${xOdds}`);
console.log(`Odd of victory of ${team2}: ${team2Odds}`);
OR
*/

//have to use array notation for accesing object value here -> if we use dot notation, we'll be lookign for a property called 'team'
for (const [team, odd] of Object.entries(game.odds)) {
  const teamStr = team === 'x' ? 'draw' : `victory for ${game[team]}`;
  console.log(`Odd of ${teamStr}: ${odd}`);
}
```

```
//4
const scorers = {};
for (const player of game.scored) {
  scorers[player] = scorers[player]
    ? (scorers[player] += 1)
    : (scorers[player] = 1);
}
console.log(scorers);

//HANDLING NESTED ARRAYS
const entryPlayers = Object.entries(game.players);
// console.log(entryPlayers);
const [firstRoster, [, rosterNames]] = entryPlayers;
console.log(rosterNames);
```

Sets

```
///////////////////////////////////////
// Sets
const ordersSet = new Set([
  'Pasta',
  'Pizza',
  'Pizza',
  'Risotto',
  'Pasta',
  'Pizza',
]);
console.log(ordersSet);

console.log(new Set('Jonas'));

console.log(ordersSet.size);
console.log(ordersSet.has('Pizza'));
console.log(ordersSet.has('Bread'));
ordersSet.add('Garlic Bread');
ordersSet.add('Garlic Bread');
ordersSet.delete('Risotto');
// ordersSet.clear();
console.log(ordersSet);

for (const order of ordersSet) console.log(order);

// Example
const staff = ['Waiter', 'Chef', 'Waiter', 'Manager', 'Chef', 'Waiter'];
const staffUnique = [...new Set(staff)]; //makes  a set, and turns it into array
console.log(staffUnique); //Waiter, Chef, Manager

console.log(
  new Set(['Waiter', 'Chef', 'Waiter', 'Manager', 'Chef', 'Waiter']).size
);

console.log(new Set('jonasschmedtmann').size);
*/
```

Maps: Fundamentals

```
//MAPS -
const rest = new Map();
rest.set('name', 'McDonalds');
rest.set(1, 'My Backyard, Texas');
rest.set(2, 'Lesbian, Portugal');
console.log(rest);

// or

rest
  .set('categories', ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'])
  .set('open', 11)
  .set('close', 23)
  .set(true, 'We are open :D')
  .set(false, 'We are closed :(');

console.log(rest.get('home')); //undefined
console.log(rest.get(true));
console.log(rest.get(1));

const time = 21;
console.log(rest.get(rest.get('open') <= time && rest.get('close') > time));

console.log(rest.has('categories'));
rest.delete(2);
console.log(rest);
console.log(rest.size);

rest.set([1, 2], 'Test');
console.log(rest);

//Returns undefined. Even though the arrays contain the same elements, they do not point to the same elements in memory
console.log(rest.get([1, 2]));

//Solution 1:
const arr = [1, 2];

rest.set(arr, 'Test');
console.log(rest.get(arr));

//With dom
rest.set(document.querySelector('h1'), 'Heading');
```

Map Iterations

https://codepen.io/chaleay/pen/dypdZXB?editors=1011

Which data structures should I use?

Overview:

▼ Sources of Data

- From the program itself
- From the UI e.g data written in dom
- from external sources i.e api

▼ Simple list?

- use arrays or sets

▼ Key/value pairs?

- Objects or maps

Arrays

▼ When to use

- Use when you need ordered list of values that may contain duplicates
- Use when you need to manipulate data

Sets

▼ When to use

- Use when you need to work with unique values
- use when high-perfomance is really important
- use to remove dups from arrays

Objects

▼ When to use

- More traditional keyvalue store
- **Easier to write and access values with . and []**
- use when you need functions as values
- when working with json

Maps

▼ When to use

- Better performance
- keys can **have any data type**
- Easy to **iterate and compute size**
- when you need keys that are not strings
- use when you need to map keys to values

Strings - Basics

```
//STRINGS
const airline = 'TAP Air Portugal';
const plane = 'A320';

console.log(plane[0]);
console.log(plane[1]);
console.log(plane[2]);

//Get size of string
console.log(airline.length);

//Get index
console.log(airline.indexOf('z'));
console.log(airline.lastIndexOf('r'));
console.log(airline.indexOf('Portugal')); //case sensitive

//position at which extraction starts
console.log(airline.slice(4));
console.log(airline.slice(4, 7));
console.log(airline.slice(0, airline.indexOf(' ')));
console.log(airline.slice(airline.lastIndexOf(' ') + 1));

console.log(airline.slice(-2)); //al
console.log(airline.slice(1, -1)); //prints out AP air Portuga

const checkMiddleSeat = function (seat) {
  const char = seat.slice(seat.length - 1);
  return char === 'B' || char === 'E';
};

console.log(checkMiddleSeat('11B') ? 'Middle Seat' : 'Good Seat!');
console.log(checkMiddleSeat('11E') ? 'Middle Seat' : 'Good Seat!');
console.log(checkMiddleSeat('11C') ? 'Middle Seat' : 'Good Seat!');
```

JS converts string primitives to string objects behind the scenes.

String Methods

```
//Changing the case of a string
console.log(airline.toLowerCase());
console.log(airline.toUpperCase());

//Fix capitilization in name
const passenger = 'Elijah'
const passengerLower = passenger.toLowerCase();
const passengerCorrect = passengerLower[0].toUpperCase() +
passengerLower.slice(1)
//prints out 'Elijah'

//Check Email
const email = 'hello@elijah.io';
const loginEmail = 'Hello@Elijah.IO';

//long way
const lowerEmail = loginEmail.toLowerCase();
const trimmedEmail = lowerEmail.trim();
console.log(trimmedEmail);

const normalizedEmail = loginEmail.toLowerCase().trim();
console.log(normalizedEmail);
console.log(email === normalized);

//Replacement in string
const priceGB = '288,97€'
const priceUS = priceGB.replace('€', '$').replace(',', '.');

const announcement = 'Replace door with gate'
console.log(announcement.replace('door','gate'));

//repalce all with regex
console.log(announcement.replace(/door/g,'gate'));

//Booleans
const plane = 'A320neo';
console.log(plane.include('A320'); //true
console.log(plane.includes('Boeing'); //false
console.log(plane.startsWith('Aib'); //flase


if(plane.startsWith('Airbus') && plane.endsWith('neo')){
console.log('part of the airbus neo family');


//Practice
const checkBaggage = (baggage) => {
  const baggageNormalized = baggage.toLowerCase();
  if(baggageNormalized.includes('knife') || baggageNormalized.includes('gun')){
    return 'Get the fuck out of here';
  }
else return 'Welcome aboard';
}
checkBaggage('Pocket Knife, Food, camera');
checkBaggage('Gun');
```

Strings - splitting and more

```
//Split
console.log('A very nice string'.split(' ')): //An array with 4 elements
//delimited by space

console.log('Elijah-Gaytan'.split('-'));
```

```
//Join
const [firstName, lastName] = 'Elijah Gaytan'.split(' ');
const newName = ['Mr', firstName, lastName.toUpperCase()].join(' ');
// Mr. Elijah GAYTAN

const capitalizeName = function(name) {
  const names = name.split(' ');
  const newName = [];

for(const n of names){
  newName.push(n[0].toUpperCase() + n.slice(1));
  /*
    newName.push(n.replace(n[0], n[0].toUpperCase()));
  */

return newName.join(' ');
}

const passenger = 'Jessica ann smith davis'
const passenger2 = 'Elijah'


//String Padding
const message = 'Go to gate 23';
console.log(message.padStart(25, '+'));
//pad the string to a lenght of 25 with + sign
console.log(message.padStart(25, '+').padEnd(35, '+'));

const maskCreditCard = function(number) {
  //convert number to string
  const str = number + '';
  const last = str.splice(-4);
  return last.padStart(str.length, 'X');
}

console.log(maskCreditCard(430942939045));
console.log(maskCreditCard('4343434343434'));

// Repeat
const message2 = 'Bad Weather, watch the fuck out...';

console.log(message2.repeat(5));
const planesInLine = function(n){
console.log(`There are ${n} planes in line ${'✈️'.repeat(n)}`);
}

planesInLine(5);
```

```
//Split
console.log('A very nice string'.split(' ')): //An array with 4 elements
//delimited by space

console.log('Elijah-Gaytan'.split('-'));

//Join
const [firstName, lastName] = 'Elijah Gaytan'.split(' ');
const newName = ['Mr', firstName, lastName.toUpperCase()].join(' ');
// Mr. Elijah GAYTAN

const capitalizeName = function(name) {
  const names = name.split(' ');
  const newName = [];

for(const n of names){
  newName.push(n[0].toUpperCase() + n.slice(1));
  /*
    newName.push(n.replace(n[0], n[0].toUpperCase()));
  */

return newName.join(' ');
}

const passenger = 'Jessica ann smith davis'
const passenger2 = 'Elijah'


//String Padding
```

```
const message = 'Go to gate 23';
console.log(message.padStart(25, '+'));
//pad the string to a lenght of 25 with + sign
console.log(message.padStart(25, '+').padEnd(35, '+'));

const maskCreditCard = function(number) {
  //convert number to string
  const str = number + '';
  const last = str.splice(-4);
  return last.padStart(str.length, 'X');
}

console.log(maskCreditCard(430942939045));
console.log(maskCreditCard('4343434343434'));

// Repeat
const message2 = 'Bad Weather, watch the fuck out...';

console.log(message2.repeat(5));
const planesInLine = function(n){
console.log(`There are ${n} planes in line ${'✈️'.repeat(n)}`);
}

planesInLine(5);
```

Coding Challenge 4

```
/*
Write a program that receives a list of variable names written in underscore_case and convert them to camelCase.

The input will come from a textarea inserted into the DOM (see code below), and conversion will happen when the button is pressed.

THIS TEST DATA (pasted to textarea)
underscore_case
 first_name
Some_Variable
  calculate_AGE
delayed_departure

SHOULD PRODUCE THIS OUTPUT (5 separate console.log outputs)
underscoreCase      ✅
firstName           ✅✅
someVariable        ✅✅✅
calculateAge        ✅✅✅✅
delayedDeparture    ✅✅✅✅✅

HINT 1: Remember which character defines a new line in the textarea 😉
HINT 2: The solution only needs to work for a variable made out of 2 words, like a_b
HINT 3: Start without worrying about the ✅. Tackle that only after you have the variable name conversion working 😉
HINT 4: This challenge is difficult on purpose, so start watching the solution in case you're stuck. Then pause and continue!

Afterwards, test with your own test data!

*/

//1.
const textarea = document.createElement('textarea');
const button = document.createElement('button');
button.textContent = 'Submit';
document.querySelector('.append').append(textarea);
document.querySelector('.append').append(button);

button.addEventListener('click', () => {
  //split by new line
  const textareaSplit = textarea.value.split('\n');
  console.log(textareaSplit);

  //keep track of array items
  let n = 0;
  for (const text of textareaSplit) {
    n++;
    const textArray = text.trim().toLowerCase().split('_');
    let [firstPart, secondPart] = textArray;
    //capitalize the first letter of the second part
    secondPart = secondPart.replace(secondPart[0], secondPart[0].toUpperCase());
    textArray[1] = secondPart;
    let newText = textArray.join('');
    console.log(`${newText.padEnd(20, ' ')}${'✅'.repeat(n)}`);
  }
});
```

Functions

Default Parameter examples

```
const bookings  = [];
const createBooking = function(flightNum, numPassenger = 1, price = 1) {
  //es5 way
  //numPassengers = numPassenger ?? 1;
  //price = price ?? 1;

  const booking =  {
    flightNum, numPassengers,
    price
  }

booking.push(booking);

}

//without all params defined
createBooking('Blah');
createBooking('Blah', 2);
createBooking('Blah', 4, 5);
//set only the third param
createBooking('Blah', undefined, 1000);
```

Passing by value vs reference

Primitive types are passed via copy.

Objects are passed to functions via value. We pass a reference to the function, but we do not pass by reference. The parameter you passed into the function will still point directly to the same object in the memory heap.

```
const flight = 'ER234'
const Elijah = {
  name: Elijah,
  passport: 23245235
}

const checkIn = function(flightNum, passenger){
//changing the flightNum
  flightNum = 'ASS4';
  passenger.name = 'Mr.' + passenger.name;

  if(passenger.passport === 23245235){
    alert('Check In')
  }
  else{
    alert('Wrong passport');
}

checkIn(flight, Elijah);
console.log(flight); // -> ER234
console.log(passenger; // -> passenager.name = Mr. Elijah

const newPassport = function(person) {
  person.passport = Math.trunc(Math.random()* 9000);

}

//Wrong passport!
newPassport(elijah);
checkIn(flight, elijah);
```

First-Class Functions

▼ What are first-class functions?

- JS treats functions as first-class citizens
- This means that functions are simply values
- Functions are just another **type of object**
  - Store functions in variables or properties
  - Pass functions as arguments to other functions
  - return functions from functions
  - call methods on functions

▼ What are Higher-Order Functions?

- a function that receives another function as an argument that returns a new function, or both
- This is only possible because of first-class functions
  - A function that receives another function
  - Functions that return a new function

▼ Callback Functions

```
'use strict';

//cuts out spaces in string
const oneWord = function (str) {
  return str.replace(/ /g, '').toLowerCase();
};

const upperFirstWord = function (str) {
  const [first, ...other] = str.split(' ');
  return [first.toUpperCase(), ...other].join(' ');
};

//Higher order function
const transformer = function (str, fn) {
  console.log(`Original string: ${str}`);
  console.log(`Transform string: ${fn(str)}`);

  console.log(`Transformed by ${fn.name}`);
};

//passing in callback functions (the transformed calls these functions later)
transformer('Javascript is the best!', upperFirstWord);
transformer('Javascript is the best!', oneWord);

const high5 = () => {
  console.log('👋');
};

//JS uses callbacks all the time
document.body.addEventListener('click', high5);

//callbacks here as well
['Blah', 'Yah', 'Fugg'].forEach(high5);
```

Functions returning functions

```
const greet = function(greeting) {
  return function(name) {
    console.log(`${greeting} ${name}`);
  }
}

const greeterHey = greet('hey');
```

Call and Apply

```
// The call and apply Methods
const lufthansa = {
  airline: 'Lufthansa',
  iataCode: 'LH',
  bookings: [],
  // book: function() {}
  book(flightNum, name) {
    console.log(
      `${name} booked a seat on ${this.airline} flight ${this.iataCode}${flightNum}`
    );
    this.bookings.push({ flight: `${this.iataCode}${flightNum}`, name });
  },
};

lufthansa.book(239, 'Jonas Schmedtmann');
lufthansa.book(635, 'John Smith');

const eurowings = {
  airline: 'Eurowings',
  iataCode: 'EW',
  bookings: [],
};

const book = lufthansa.book;

// Does NOT work
// book(23, 'Sarah Williams');

// Call method
book.call(eurowings, 23, 'Sarah Williams');
console.log(eurowings);

book.call(lufthansa, 239, 'Mary Cooper');
console.log(lufthansa);

const swiss = {
  airline: 'Swiss Air Lines',
  iataCode: 'LX',
  bookings: [],
};

book.call(swiss, 583, 'Mary Cooper');

// Apply method
const flightData = [583, 'George Cooper'];
book.apply(swiss, flightData);
console.log(swiss);

book.call(swiss, ...flightData);
```

Bind

```
///////////////////////////////////////
// The bind Method
// book.call(eurowings, 23, 'Sarah Williams');

const bookEW = book.bind(eurowings);
const bookLH = book.bind(lufthansa);
const bookLX = book.bind(swiss);

bookEW(23, 'Steven Williams');

const bookEW23 = book.bind(eurowings, 23);
bookEW23('Jonas Schmedtmann');
bookEW23('Martha Cooper');

// With Event Listeners
lufthansa.planes = 300;
lufthansa.buyPlane = function () {
  console.log(this);

  this.planes++;
  console.log(this.planes);
};
// lufthansa.buyPlane();

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

// Partial application
const addTax = (rate, value) => value + value * rate;
console.log(addTax(0.1, 200));

const addVAT = addTax.bind(null, 0.23);
// addVAT = value => value + value * 0.23;

console.log(addVAT(100));
console.log(addVAT(23));

const addTaxRate = function (rate) {
  return function (value) {
    return value + value * rate;
  };
};
const addVAT2 = addTaxRate(0.23);
console.log(addVAT2(100));
console.log(addVAT2(23));
*/
```

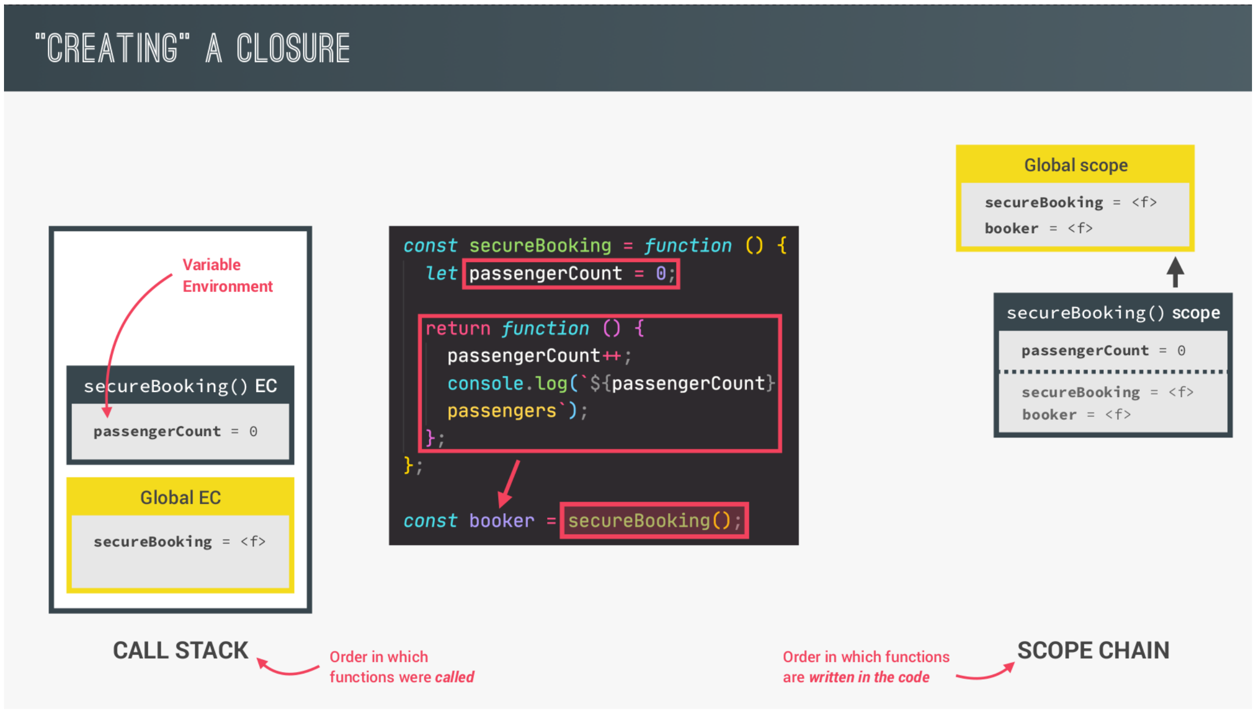
Functions - Challenge 1

Immediately Invoked Functions

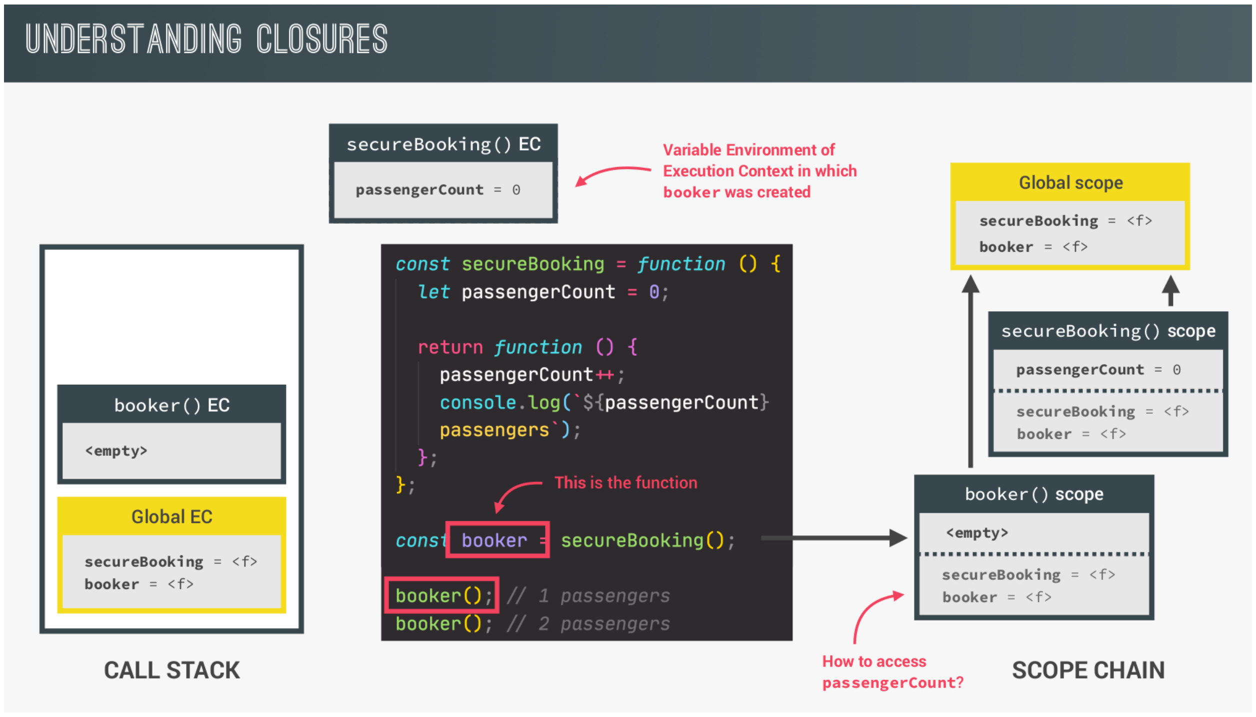
```
*/
//IIFE
/*
const runOnce = function () {
  console.log('This can run again');
};

(function () {
  console.log('This will run once');
})();

console.log(isPrivate);

(() => {
  console.log('This arrow function will only run once');
})();

{
  var c = 12;
}
//works b/c var is not block-scoped
console.log(c);
*/
```

Closures

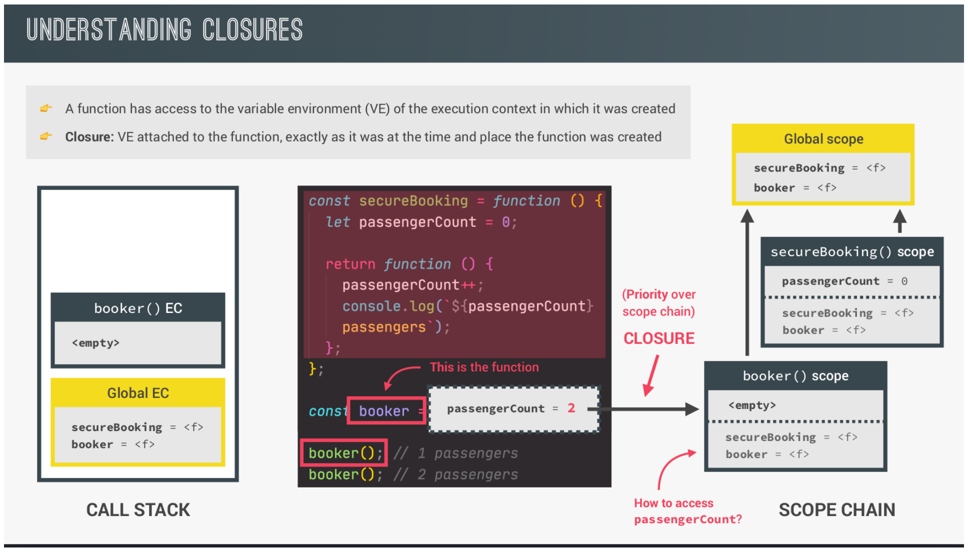
```
//Example 1
const secureBooking = function () {
  let passengerCount = 0;
  return function () {
    passengerCount++;
    console.log(`${passengerCount} passengers`);
  };
};

const booker = secureBooking();

//What allows booker able to update passengerCount despite it not being in the execution context is Closures
booker(); //1 passengers
booker(); //2 passengers
booker(); //3 passengers

//Any function always has access to the variable environment of the execution context in which the function was created, even after
```



```
const secureBooking = function () {
  let passengerCount = 0;

  return function () {
    passengerCount++;
    console.log(`${passengerCount}
    passengers`);
  };
};

const booker = secureBooking();
```



```
const secureBooking = function () {
  let passengerCount = 0;

  return function () {
    passengerCount++;
    console.log(`${passengerCount}
    passengers`);
  };
};

const booker = secureBooking();

booker(); // 1 passengers
booker(); // 2 passengers
```



A function has access to the variable environment (VE) of the execution context in which it was created

Closure: VE attached to the function, exactly as it was at the time and place the function was created

```
const secureBooking = function () {
  let passengerCount = 0;

  return function () {
    passengerCount++;
    console.log(`${passengerCount}
    passengers`);
  };
};

const booker = secureBooking();

booker(); // 1 passengers
booker(); // 2 passengers
```

Closure Summary

- 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.*
- A closure gives a function access to all the 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 throughout the time.
- **A closure makes sure that a function doesn't lose connection to variables that existed at the functions birth place.**

Closures - Examples

```
let f;
```

```
const g = function () {
  const a = 23;
  f = function () {
    console.log(a * 2);
  };
};

const h = function () {
  const b = 777;
  f = function () {
    console.log(b * 2);
  };
};

g();
//still has access to g's variables even though g's execution context is removed
f();
//reassign f
console.dir(f);

h();
f();
console.dir(f); //no longer has the value of a after being reassigned

//Example 2
const boardPassengers = function (n, wait) {
  const perGroup = n / 3;

  //this function creates a closure to access boardPasengers variables after boardPassengers finishes its execution.
  setTimeout(function () {
    console.log(`We are now boarding all ${n} passengers`);
    console.log(`There are 3 groups, each with ${perGroup} passengers`);
  }, wait * 1000);

  console.log(`Will start boarding in ${wait} seconds.`);
};

//closure has priority over scope chain
const perGroup = 1000;
boardPassengers(180, 3);
```

CC #4

```
//1. Why does this work?
/*
  Even though the main function has finished its execution, you are still able to access the header variable that is declared.
  This is due to closure. A closure makes sure that the function does not lose access
  to variables that existed at its birth place.
*/

(function () {
  const header = document.querySelector('h1');
  header.style.color = 'red';

  document.body.addEventListener('click', () => {
    header.style.color = 'blue';
  });
})();
```

Arrays

Simple Array Methods

```
let arr = ['a,', 'b', 'c', 'd', 'e'];
//starting index included
console.log(arr.slice(2));
//length of array is end paramater - begin parameter
//end is exclusive
```

```
console.log(arr.slice(2, 4)); //c,d
console.log(arr.slice(-2)); //d, e
console.log(arr.slice(-1)); // e
console.log(arr.slice(1, -1)); //b c d
//The best way I've found about thinking about this one is that a negative element in the
//second parameter refers to how many elements you ignore going from the end of the array to the start
console.log(arr.slice(1, -2)); //b c

//shallow copies
const copy1 = [...arr];
const copy2 = arr.slice();

//Splice - mutates the actual array to get the elements you didn't slice
//console.log(arr.splice(2)); //2 3 4
//console.log(arr); // 1 2

//deleting elements of array - last element
//arr.splice(-1);
//console.log(arr);
//at index 1, delete 2 elements
console.log(arr.splice(1, 2)); //b c removed a d e
console.log(arr); // a d e

//REVERSE
arr = ['a', 'b', 'c', 'd', 'e'];
const arr2 = ['j', 'i', 'h', 'g', 'f'];
console.log(arr2.reverse());

//Concat
const letters = arr.concat(arr2);
console.log(letters);
//can just do spread ops instead
console.log([...arr, ...arr2]); //"a", "b", "c", "d", "e", "f", "g", "h", "i", "j"

//Join
console.log(letters.join(' - ')); //a - b - c - d - e - f - g - h - i - j
```

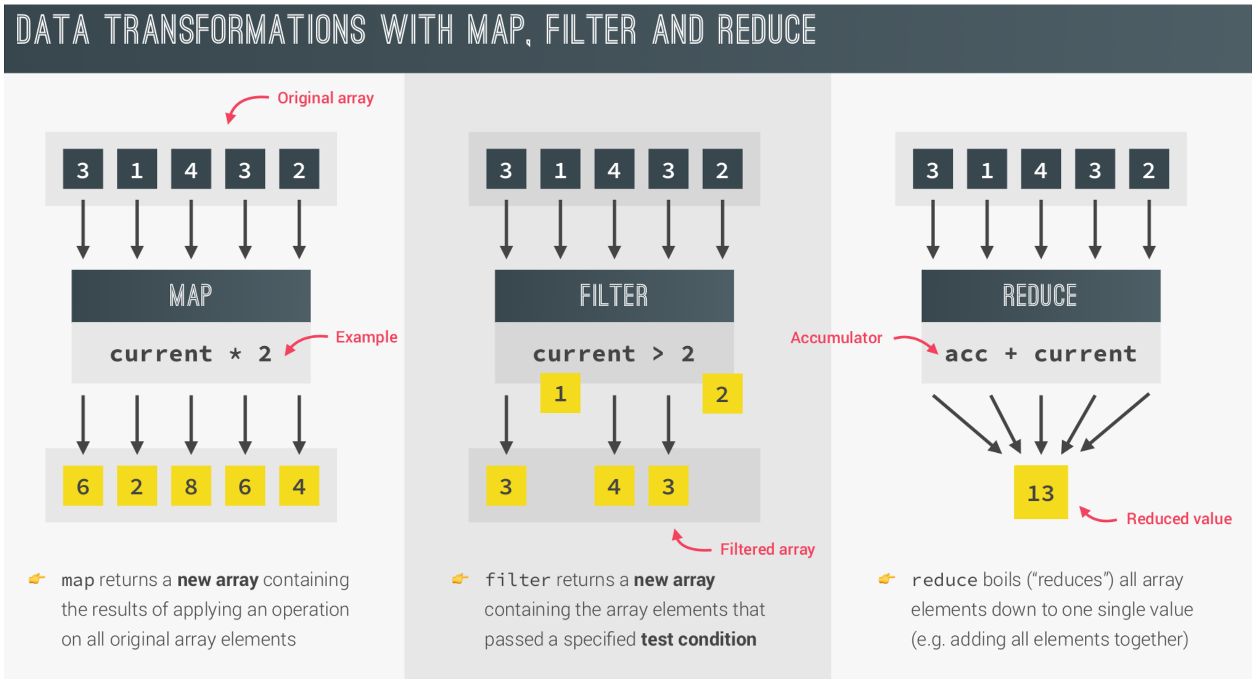
Array forEach

```
/////////////////////////////////////////////////////////////
//Looping

//for of
// for (const movement of movements) {
// if (movement > 0) {
// console.log(`You deposited ${movement}`);
// } else {
// console.log(`You widthdrew ${Math.abs(movement)}`);
// }
// }

//for each - passes in current element, index, and the entire array
//a fundamental difference is that you cannot break out of a for each loop using continue or break
//the break statement can be used to jump out of loop
//the continue keyword can be used to break one iteration of the loop and continues with the next iteration

movements.forEach((movement, index, array) => {
  if (movement > 0) {
    console.log(`Movement ${index + 1}: You deposited ${movement}`);
  } else {
    console.log(`Movement ${index + 1}: You widthdrew ${Math.abs(movement)}`);
  }
});
```

forEach on Maps and Sets

```
///////////////////////////////////////
// forEach With Maps and Sets
// Map
const currencies = new Map([
  ['USD', 'United States dollar'],
  ['EUR', 'Euro'],
  ['GBP', 'Pound sterling'],
]);

currencies.forEach(function (value, key, map) {
  console.log(`${key}: ${value}`);
});

// Set
const currenciesUnique = new Set(['USD', 'GBP', 'USD', 'EUR', 'EUR']);
console.log(currenciesUnique);
currenciesUnique.forEach(function (value, _, map) {
  console.log(`${value}: ${value}`);
});
```

Filter, Map, and Reduce


DATA TRANSFORMATIONS WITH MAP, FILTER AND REDUCE
Original array
3 1 4 3 2
MAP
current * 2
Example
6 2 8 6 4
3 1 4 3 2
FILTER
current > 2
1 2
3 4 3
Filtered array
Accumulator
3 1 4 3 2
REDUCE
acc + current
13
Reduced value
map 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 passed a specified test condition
reduce boils ("reduces") all array elements down to one single value (e.g. adding all elements together)

Map

```
//The map method
const euroToUd = 1.1;

//Map method
//First param: value
//Second param: index
//Third param: array
const movementsUSD = movements.map(mov => {
  return mov * euroToUd;
});

console.log(movementsUSD);

const moveDescriptions = movements.map((movement, index) => {
  if (movement > 0) {
    return `Movement ${index + 1}: You deposited ${movement}`;
  } else {
    return `Movement ${index + 1}: You withdrew ${Math.abs(movement)}`;
  }
});

console.log(moveDescriptions);
```

Filter

```
/Filter method

const deposits = movements.filter(mov => {
  return mov > 0;
});
const withdrawals = movements.filter(mov => {
  return mov < 0;
});

console.log(deposits);
console.log(withdrawals);
```

Reduce

```
//Reduce
console.log(movements);

//func
const balance = movements.reduce(
  function (acc, cur, i, arr) {
    console.log(`Iteration ${i}: ${cur} added to ${acc}`);
    return acc + cur;
  },
  0 //inital value of the accumulator
);

//arrow
const balanceArrow = movements.reduce((acc, curr) => {
  return acc + curr;
}, 0);

//Find max
const max = movements.reduce((acc, curr) => {
  return curr > acc ? curr : acc;
}, movements[0]);

console.log(max);
console.log(balanceArrow);
```

CC W/ Map, Reduce, Filter #2

```
///////////////////////////////////////
// Coding Challenge #2

/*
Let's go back to Julia and Kate's study about dogs. This time, they want to convert dog ages to human ages and calculate the average

Create a function 'calcAverageHumanAge', which accepts an arrays of dog's ages ('ages'), and does the following things in order:

1. Calculate the dog age in human years using the following formula: if the dog is <= 2 years old, humanAge = 2 * dogAge. If the dog
2. Exclude all dogs that are less than 18 human years old (which is the same as keeping dogs that are at least 18 years old)
3. Calculate the average human age of all adult dogs (you should already know from other challenges how we calculate averages 😉)
4. Run the function for both test datasets

TEST DATA 1: [5, 2, 4, 1, 15, 8, 3]
TEST DATA 2: [16, 6, 10, 5, 6, 1, 4]

GOOD LUCK 😀
*/

const calcAverageHumanAge = function (ages) {
  const humanAges = ages.map((val, index) => {
    return val <= 2 ? 2 * val : 16 + val * 4;
  });
  const humanAgesFiltered = humanAges.filter(age => {
    return age >= 18;
  });
  const avg =
    humanAgesFiltered.reduce((acc, cur) => {
      return acc + cur;
    }) / humanAgesFiltered.length;

  return avg;
};

console.log(calcAverageHumanAge([5, 2, 4, 1, 15, 8, 3]));
console.log(calcAverageHumanAge([16, 6, 10, 5, 6, 1, 4]));
```

It's good practice to not chain multiple functions that mutate the array. It is also important to be wary of the impact

Chaining

```
//Chaining
const euroToUsd = 1.1;

console.log(movements);

const totalDepositsUSD = movements
  .filter(mov => mov > 0)
  .map(mov => mov * euroToUsd)
  .reduce((acc, mov) => acc + mov, 0);

const totalDepositsUSD2 = movements
  .filter(mov => mov > 0)
  .map((mov, i, arr) => {
    //the arr returned above
    console.log(arr);
    return mov * euroToUsd;
  })
  .reduce((acc, mov) => acc + mov, 0);

console.log(totalDepositsUSD2);
```

CC#3

```
const calcAverageHumanAge = ages => {
  return ages
    .map(val => (val <= 2 ? 2 * val : 16 + val * 4))
    .filter(age => age >= 18)
    .reduce((acc, age, i, arr) => acc + age / arr.length, 0);
};

console.log(calcAverageHumanAge([5, 2, 4, 1, 15, 8, 3]));
```

The Find Method

```
//The find method - Returns the first element that satisfies this condition
const firstWidthdrawal = movements.find(mov => mov < 0);

console.log(movements);
console.log(firstWidthdrawal);

//using find w/ objects
const account = accounts.find(acc => acc.owner === 'Jessica Davis');
console.log(account); //returns the jessica davis object
```

Some, Includes, and Every

```
//includes, some, and every

//returns true if any value in the array matches -130 EXACTLY
console.log(movements.includes(-130));

//TESTS FOR EQUALITY
const anyDeposits = console.log(movements.some(mov => mov > 0));

//Every method - every item must pass this test
console.log(movements.every(mov => mov > 0));

//seperate callback
const deposit = mov => mov > 0;
console.log(movements.some(deposit));
```

Flat and FlatMap

```
////////
//FLAT//
const arr = [1, 2, 3, [4, 5, 6], 7, 8];
console.log(arr.flat());
const arrDeep = [1, 2, 3, [4, 5, [6, 7, 8], 9], [10, [2]]];
console.log(arrDeep.flat(2)); //[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 2]

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

//denest our arrays
console.log(accountMovements.flat());

//chainign way
let overallBalance = accounts
  .map(acc => acc.movements)
  .flat()
  .reduce((acc, mov) => acc + mov);

//flatmap way (only goes 1 level deep)
overallBalance = accounts
  .flatMap(acc => acc.movements)
  .reduce((acc, mov) => acc + mov);

console.log(overallBalance);

//////////////////////////////////////
```

Sorting

```
//Strings
const owners = ['Jonas', 'Zach', 'Adam', 'Martha'];
console.log(owners.sort());

//Numbers -> return < 0, A, B -> keep order
//return > 0, B, A -> switch order

//ascending order
console.log(
  movements.sort((a, b) => {
    return a > b ? 1 : -1;
  })
);

//ascending order
console.log(
  movements.sort((a, b) => {
    return a - b;
  })
);

//Descending
console.log(
  movements.sort((a, b) => {
    return a > b ? -1 : 1;
  })
);

//Descending
console.log(
  movements.sort((a, b) => {
    return b - a;
  })
);
```

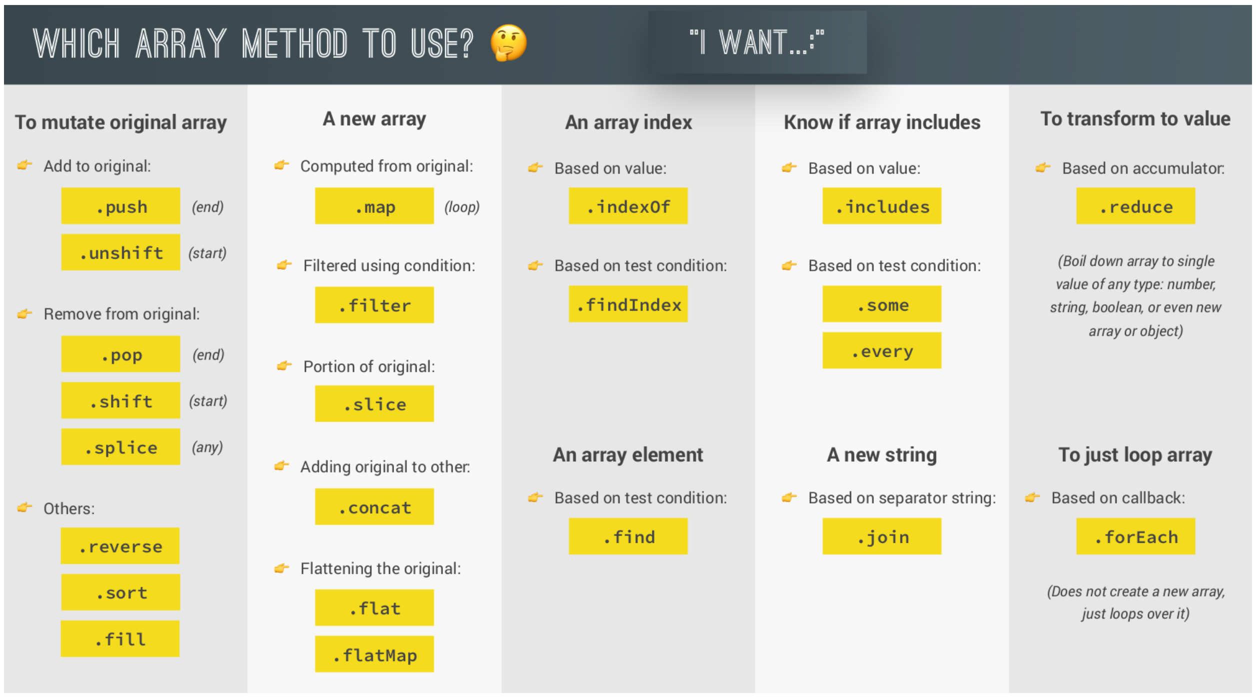
HTML CHEAT SHEET

HTML CHEAT SHEET

ARRAY - FILL AND FROM

```
//Array Creation and Filling
const sucker = new Array(4); //empty * 4
console.log(sucker);

//Fill
//sucker.fill(1);
//@PARAMS
//1ST - object we want to insert
//2ND - index to start at
//3RD - index to end at (exlusive)
sucker.fill(1, 3);
console.log(sucker); // 1 1 1 1

//
const arr = [1, 2, 3, 4, 5, 6, 7];
arr.fill(23, 4, 6);
console.log(arr); // 1 1 1 1

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

//NOTE: USE _ for optional params
//in the length part optionally you can put an array
const z = Array.from({ length: 7 }, (cur, i) => i);
console.log(z);

//100 random rice roll
//math.random - begin is inclusive, end is exclusive
const randomDiceRoll = Array.from({ length: 100 }, (_, i) => {
  return 1 + Math.floor(Math.random() * 6);
});
console.log(randomDiceRoll);

labelBalance.addEventListener('click', function () {
  //instead of having to map through with a second function, we can pass in a callback function instead
  const movementsUI = Array.from(
    document.querySelectorAll('.movements__value'),
    el => Number(el.innerText.replace('$', ''))
  );
  console.log(movementsUI);
});
```

Which Array Method should I use?



WHICH ARRAY METHOD TO USE? 🤔 "I WANT...:"

To mutate original array

- Add to original:
  - `.push` (end)
  - `.unshift` (start)
- Remove from original:
  - `.pop` (end)
  - `.shift` (start)
  - `.splice` (any)
- Others:
  - `.reverse`
  - `.sort`
  - `.fill`

A new array

- Computed from original:
  - `.map` (loop)
- Filtered using condition:
  - `.filter`
- Portion of original:
  - `.slice`
- Adding original to other:
  - `.concat`
- Flattening the original:
  - `.flat`
  - `.flatMap`

An array index

- Based on value:
  - `.indexOf`
- Based on test condition:
  - `.findIndex`

An array element

- Based on test condition:
  - `.find`

Know if array includes

- Based on value:
  - `.includes`
- Based on test condition:
  - `.some`
  - `.every`

A new string

- Based on separator string:
  - `.join`

To transform to value

- Based on accumulator:
  - `.reduce`

(Boil down array to single value of any type: number, string, boolean, or even new array or object)

To just loop array

- Based on callback:
  - `.forEach`

(Does not create a new array, just loops over it)

```
///////////////////////////////////////
// Coding Challenge #4

/*
Julia and Kate are still studying dogs, and this time they are studying if dogs are eating too much or too little.
Eating too much means the dog's current food portion is larger than the recommended portion, and eating too little is the opposite.
Eating an okay amount means the dog's current food portion is within a range 10% above and 10% below the recommended portion (see hi

1. Loop over the array containing dog objects, and for each dog, calculate the recommended food portion and add it to the object as
2. Find Sarah's dog and log to the console whether it's eating too much or too little. HINT: Some dogs have multiple owners, so you
3. Create an array containing all owners of dogs who eat too much ('ownersEatTooMuch') and an array with all owners of dogs who eat
4. Log a string to the console for each array created in 3., like this: "Matilda and Alice and Bob's dogs eat too much!" and "Sarah
5. Log to the console whether there is any dog eating EXACTLY the amount of food that is recommended (just true or false)
6. Log to the console whether there is any dog eating an OKAY amount of food (just true or false)
7. Create an array containing the dogs that are eating an OKAY amount of food (try to reuse the condition used in 6.)
8. Create a shallow copy of the dogs array and sort it by recommended food portion in an ascending order (keep in mind that the port

HINT 1: Use many different tools to solve these challenges, you can use the summary lecture to choose between them 😉
HINT 2: Being within a range 10% above and below the recommended portion means: current > (recommended * 0.90) && current < (recomme

TEST DATA:
const dogs = [
  { weight: 22, curFood: 250, owners: ['Alice', 'Bob'] },
  { weight: 8, curFood: 200, owners: ['Matilda'] },
  { weight: 13, curFood: 275, owners: ['Sarah', 'John'] },
  { weight: 32, curFood: 340, owners: ['Michael'] }
];

GOOD LUCK 😀
*/

const dogs = [
  { weight: 22, curFood: 250, owners: ['Alice', 'Bob'] },
  { weight: 8, curFood: 200, owners: ['Matilda'] },
  { weight: 13, curFood: 275, owners: ['Sarah', 'John'] },
  { weight: 32, curFood: 340, owners: ['Michael'] },
];

//1
dogs.forEach(dog => {
  dog.portion = Math.trunc(dog.weight ** 0.75 * 28);
});
console.log(dogs);

//2
const sarahDog = dogs.find(dog => {
  return dog.owners.includes('Sarah');
  //return dog.owners.find(owner => owner === 'Sarah');
});
console.log(
  `
  Sarah's Dog ${
    sarahDog.curFood > sarahDog.portion
      ? `is eating too much`
      : sarahDog.portion === sarahDog.curFood
      ? `is eating the perfect amount`
      : `is not eating enough`
  } `
);

///3.
const ownersEatTooMuch = dogs.filter(dog => dog.curFood > dog.portion);
const ownersEatTooLittle = dogs.filter(dog => dog.curFood < dog.portion);

console.log(ownersEatTooMuch);
console.log(ownersEatTooLittle);

//4:

//grab the owners, flat them
// const allOwnersTooMuch = ownersEatTooMuch.map(dog => dog.owners).flat();
// const allOwnersTooLittle = ownersEatTooLittle.map(dog => dog.owners).flat();
const allOwnersTooMuch = ownersEatTooMuch.flatMap(dog => dog.owners);
const allOwnersTooLittle = ownersEatTooLittle.flatMap(dog => dog.owners);
console.log(`${allOwnersTooMuch.join(' and ')}\'s dog eat too much!`);
console.log(`${allOwnersTooLittle.join(' and ')}\'s dog eat too little!`);

//5

const justRight = dogs.some(dog => dog.curFood === dog.portion);
console.log(justRight);

//6
let justOK = dogs.some(
  dog => dog.curFood > 0.9 * dog.portion && dog.curFood < 1.1 * dog.portion
);
console.log(justOK > 0);

//7
justOK = dogs.filter(
  dog => dog.curFood > 0.9 * dog.portion && dog.curFood < 1.1 * dog.portion
);
console.log(justOK);

//8
const sortedDogs = dogs.slice().sort((a, b) => a.portion - b.portion);
console.log(sortedDogs);
```

Converting and Checking Numbers

```
//int 64 bit
console.log(0.1 + 0.2); //.300000004

//comparing - need to scale
console.log(Math.round((0.1 + 0.2) * 10) === 0.3 * 10);

//conversion
console.log(Number('23'));
console.log(+'23');

//parsing
//accepts number to parse and the base
console.log(Number.parseInt('30px', 10)); //30
console.log(Number.parseInt('e30px', 10)); //nan

//convention to call w/ number (use namespace)
console.log(parseFloat('2.5rem')); //2.5
console.log(parseFloat('    2.5rem   ')); //2.5

//check if value is NaN
console.log(Number.isNaN(20)); //false
console.log(Number.isNaN('19')); //false
console.log(Number.isNaN(+'19')); //true
console.log(Number.isNaN(23 / 0)); //infinity -> false

//Check if a value is a number
console.log(Number.isFinite(20)); //true
console.log(Number.isFinite('20')); //false
console.log(Number.isFinite(+'20')); //false -> nan
console.log(Number.isNaN(23 / 0)); //infinity -> false
```

Math and Rounding

```
///////////////////////////////////////
// Math and Rounding
console.log(Math.sqrt(25));
console.log(25 ** (1 / 2));
console.log(8 ** (1 / 3));

console.log(Math.max(5, 18, 23, 11, 2));
console.log(Math.max(5, 18, '23', 11, 2));
console.log(Math.max(5, 28, '23px', 11, 2)); //nan as long as incompatible type is in there

console.log(Math.min(5, 18, 23, 11, 2));

console.log(Math.PI * Number.parseFloat('10px') ** 2);

console.log(Math.trunc(Math.random() * 6) + 1);

const randomInt = (min, max) =>
  //my version - actually generates values between 10...20
  Math.floor(Math.random() * (max - (min - 1)) + 1) + (min - 1);
//his version - but is min exclusive (i.e doesn't include the first number of min in calculation)
//Math.floor(Math.random() * (max - min) + 1) + min;
// 0...1 -> 0...(max - min) -> 0 + min...(max - min + min) -> min...max
console.log(randomInt(10, 20));

// Rounding integers
console.log(Math.round(23.3));
console.log(Math.round(-23.9)); //-24

//up
console.log(Math.ceil(23.3));
console.log(Math.ceil(23.9));

//down
console.log(Math.floor(23.3));
console.log(Math.floor('23.9')); //23

//down if positive, up if negative (Just cuts off decimal part, whole num stays the same)
console.log(Math.trunc(23.3));

console.log(Math.trunc(-23.3));
console.log(Math.floor(-23.3));

// Rounding decimals
console.log((2.7).toFixed(0)); //returns a string
console.log((2.7).toFixed(3));
console.log((2.345).toFixed(2));
console.log(+(2.345).toFixed(2));
```

```
///////////////////////////////////////

// The Remainder Operator

console.log(5 % 2);
console.log(5 / 2); // 5 = 2 * 2 + 1



console.log(8 % 3);
console.log(8 / 3); // 8 = 2 * 3 + 2
console.log(6 % 2);
console.log(6 / 2);



console.log(7 % 2);
console.log(7 / 2);


const isEven = n => n % 2 === 0;
console.log(isEven(8));
console.log(isEven(23));
console.log(isEven(514));


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

```
// Working with BigInt
console.log(2 ** 53 - 1);
console.log(Number.MAX_SAFE_INTEGER);
console.log(2 ** 53 + 1);
console.log(2 ** 53 + 2);
console.log(2 ** 53 + 3);
console.log(2 ** 53 + 4);

//Two ways to make big ints
console.log(4838430248342043823408394839483204n);
console.log(BigInt(48384302));

//Operations

console.log(1000000000n * 10000n);
const num = 23;
console.log(huge * BigInt(num));


console.log(20n > 15)
```

```
//Create a date
const now = new Date();
console.log(now);

console.log(new Date('Wed Jan 06 2021 01:08:43'));
//not best practice to do this
console.log(new Date('December 24, 2015'));
console.log(new Date(account1.movementsDates[0]));

console.log(new Date(2037, 10, 19, 15, 23, 5));
//auto corrects the date since november does NOT have 31 days (sets to dec 1st)
console.log(new Date(2037, 10, 31));

console.log(new Date(0));
//create a date 3 days after the above
console.log(new Date(3 * 24 * 60 * 60 * 1000));

const future = new Date(2037, 10, 19, 15, 23, 5);
console.log(future.getFullYear());
console.log(future.getMonth());
console.log(future.getDate());
console.log(future.getDay());
console.log(future.getHours());
console.log(future.getMinutes());
console.log(future.getSeconds());
console.log(future.toISOString());
console.log(future.getTime());

console.log(new Date(2142278585000));
console.log(now.getTime()); //1609918645093 time in ms
console.log(now.toLocaleDateString()); //prints out date 1/06/21

//get current timestamp
console.log(Date.now());

//setting
future.setFullYear(2040);
console.log(future);



const future = new Date(2037, 10, 19, 15, 23);
console.log(+future);

//DATE OPERATIONS
//convert to days
const daysPassed = (date1, date2) =>
  Math.abs((date2 - date1) / (1000 * 60 * 1440)); //this converts ms to days

console.log(daysPassed(new Date(2037, 3, 14), new Date(2037, 3, 24)));
```

Internationalization

```
//Dates
const newDate = new Intl.DateTimeFormat(locale).format(new Date(2019-11-01
T13:15:33.035Z));
const localDate = new Intl.DateTimeFormat(
      currentAccount.locale,
      {
      hour: 'numeric',
      minute: 'numeric',
      day: 'numeric',
      month: 'numeric', //long displays in letters, 2-digits
      year: 'numeric',
      //weekday: 'numeric',
    }
    ).format(now);

//Internationalizing numbers
const num = 3882325;

const options = {
  style: 'unit', //percent, or currency
  unit: 'mile-per-hour', //loook these up
};

console.log(new Intl.NumberFormat('en-US').format(num));
console.log(new Intl.NumberFormat('de-DE').format(num));
console.log(new Intl.NumberFormat('ar-SY').format(num));
console.log(new Intl.NumberFormat(navigator.language).format(num));
```

FOR MORE INFORMATION ON INTL API, PLEASE SEE:

chaleay/Mini-Javascript-Projects

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or window. Reload to refresh your session. Reload to refresh your session.

https://github.com/chaleay/Mini-Javascript-Projects

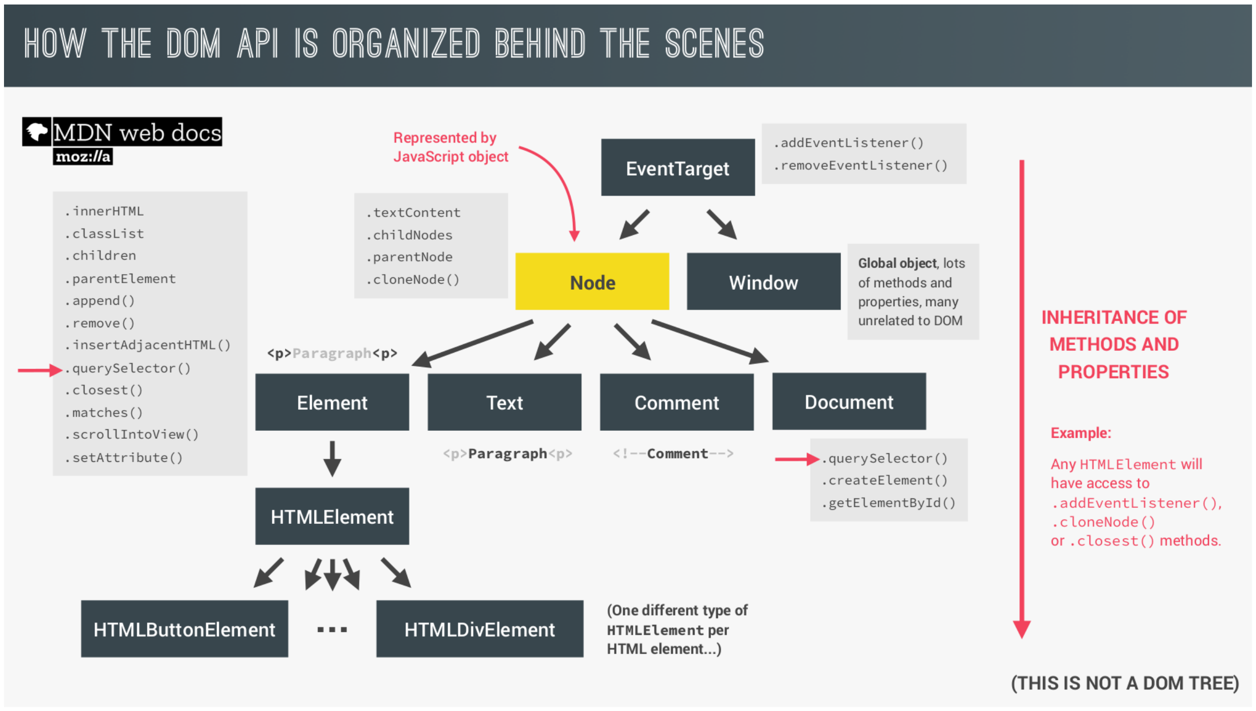
setTimer and SetInterval

```
//tiemout function
setTimeout(
  (ing1, ing2) => console.log(`Here is your pizza with ${ing1} and ${ing2}`),
  3000,
  'olives',
  'spinach'
);
//this code will execute asynchronously
console.log('Wait 3 seconds...');


setInterval(() => {
  const now = new Date();
  const newDate = new Intl.DateTimeFormat(navigator.locale, {
    month: 'long',
    day: 'numeric',
    hour: 'numeric',
    minute: 'numeric',
    second: 'numeric',
  }).format(now);

  console.log(newDate);
}, 2000);
```

What is the Dom?

- Allows us to make javascript interact with the browser
- We can write JS to create, modify, delete HTML elements. We can set styles, classes, and attributes, and listen and respond to events.
- DOM tree is generated from an HTML document, which we can then interact with.
- DOM is a very complex api that contains a lot of methods and properties to interact with the dom tree

How the DOM API is organized behind the scene



Selecting, Creating, and Deleting Elements

```
//creating and inserting Elements
// //element.insertAdjacentHTML('afterend', <p> Hello There </p>)

const message = document.createElement('div');
message.classList.add('cookie-message');
//message.textContent
message.innerHTML = `We use cookies for improved functionality and analytics <button class = "btn btn--close-cookie">
Got it</button>`;
//document.querySelector('.header').prepend(message);\
//this allows you to clone
//header.append(message.cloneNode(true));

//Inserts message as child of header
header.append(message);

//insert message before header
//header.before(message);
// header.after(message);

//delete Element
document.querySelector('.btn--close-cookie').addEventListener('click', () => {
  message.remove();
  //dom traversal method
  message.parentElement.removeChild(message);
});
```

Styles, attributes, and classes

```
///////////////////////////////
//Styles - inline
message.style.backgroundColor = '#37383d';
message.style.width = '120%';
console.log(message.style.width); //would work
console.log(message.style.height); //wouldnt work

console.log(getComputedStyle(message).color);
console.log(getComputedStyle(message).height);

//get the css height and then add 10px to it
message.style.height =
  Number.parseFloat(getComputedStyle(message).height) + 30 + 'px';
console.log(getComputedStyle(message).height);

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

//atributes
const logo = document.querySelector('.nav__logo');
console.log(logo.alt);
console.log(logo.src);
console.log(logo.className);

// logo.alt = 'test';

//Non-standard
logo.setAttribute('designer', 'Elijah');
console.log(logo.getAttribute('designer'));

//Accessing attributes - the difference
console.log(logo.src); //absolute
console.log(logo.getAttribute('src')); //relative

//Data attributes
console.log(logo.dataset.versionNumber);

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

//overrides ALL other classes
```

```
logo.className = '';
/////////////////////////////
```

Smooth Scrolling

```
const btnScrollTo = document.querySelector('.btn--scroll-to');
const section1 = document.querySelector('#section--1');

btnScrollTo.addEventListener('click', function (e) {
  const s1coords = section1.getBoundingClientRect();
  console.log(s1coords);
  //x / left- left distance from viewport
  //y / top - top distance from viewport
  console.log(e.target.getBoundingClientRect());

  //prints out how far you've scrolled relative to the top of the browser (same applies for horizontal)
  console.log('Current Scroll (x/y):', window.pageXOffset, window.pageYOffset);
  console.log(
    'height and width of viewport:',
    document.documentElement.clientHeight,
    document.documentElement.clientWidth
  );

  // Scrolling left 0, and scroll down s1coords.top px
  //however, this solution only really works when at very top of page,
  //if you scroll down distance from top of viewport to section is a lot less
  //window.scrollTo(s1coords.left, s1coords.top + window.pageYOffset);
  // window.scrollTo({
  // left: s1coords.left + window.pageXOffset, //should be 0
  // top: s1coords.top + window.pageYOffset,
  // behavior: 'smooth',
  // });

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

Event Listeners

```
//OLD SCHOOL - DONT DO
// h1.onmouseenter = function (e) {
// alert('mouse enter h1 w/o function');
// };

//Types of events and event handlers
const h1 = document.querySelector('h1');

//listen once
const alertH1 = e => {
  alert('mouse enter h1');

  //remove
  // h1.removeEventListener('mouseenter', alertH1);
};

//DO THIS - better
h1.addEventListener('mouseenter', alertH1);

//can also do this
setTimeout(() => h1.removeEventListener('mouseenter', alertH1), 3000);
```