lesson 5

code quality tooling

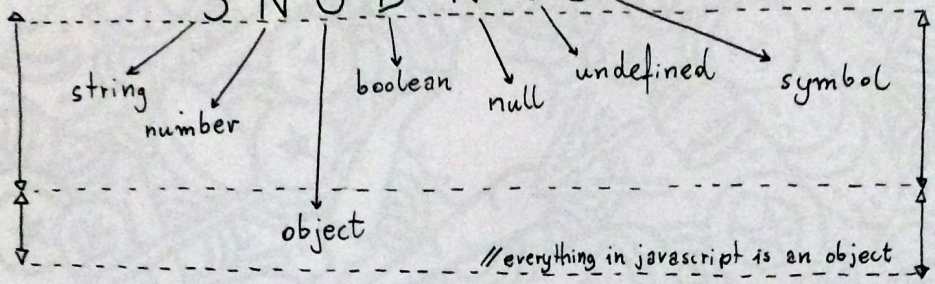
eslint
javascript linter tool for identifying and reporting potential issues in your code.
- pluggable (vue, react, angular)

prettier
an opinionated code formatter

lesson 6

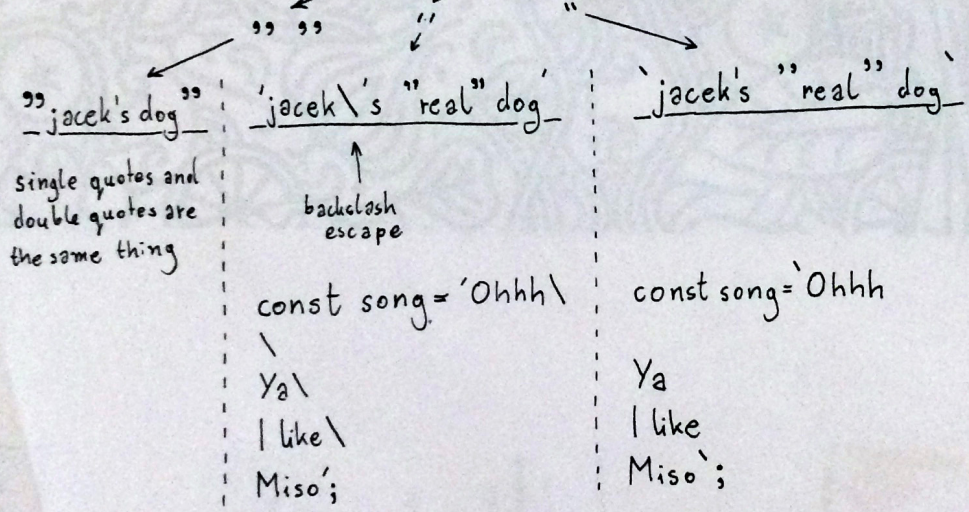
7 types in javascript

SNOBUSN



lesson 7

ways to create a string



concatenation
connecting strings

```
const hello = `hello my name is ` + name + `. Nice to meet you.`
```

interpolation
putting variable into middle of the string

```
const hello = `hello my name is ${name}. Nice to meet you.`
```

Lesson 8

number types in javascript

there is only one type of number in javascript which is... "number"

example of number types in another language

- integer — whole number
- float — number with decimal

```
> typeof NaN
> typeof Infinity
> typeof -Infinity
```
→ 'number'

```
const x = 2.5;
Math.round(x);   → 3
Math.floor(x);   → 2
Math.ceil(x);    → 3
Math.random();   → 0.174156...   between 0 (inclusive) and 1
8 % 3;           → 2             reminder of division of one number by another
```

```
>> 0.1 + 0.2
← 0.30000000000000004
```

Floating Point Math. Your language isn't broken, it's doing floating point math. Computers can only natively store integers, so they need some way of representing decimal numbers. This representation comes with some degree of inaccuracy. That's why, more often than not, .1 + .2 != .3.

```
cost price = 1034; // it's better to deal with money in whole numbers and divide it in the very end so that you don't run into decimal problems
```

Lesson 9

random keywords:
- nesting objects
- object vs referencing
- copying objects

Object

biggest building block

objects are basically used for collections of data and functionality. lorem ipsum

```
const person = {
  first: 'jacek',
  last: 'super',
  age: 1337
};
```

// person.first
// person.last
// person.age

Lesson 10

Undefined

A variable that has been created but hasn't been used

```
const something Undefined;
```

```
Let dog;
>> dog
← undefined
```

Null

A value of nothing that has been explicitly set to nothing

```
const something Null = null;
```

```
const mononymous Person = {
    first: 'Geralt',
    last: null
};
```

Lesson 11

What diet did the ghost go on?

Boolean!

true or false
on or off

```
10 === 10
```
triple equals check if both value and type of the thing on the left and on the right is the same

```
>> "10" == 10      >> "10" === 10
← true             ← false
```

Almost always you should use triple equal instead of double equal

Lesson 12

Functions allow us to group together sets of statements

```
Math.max(10.12);   ← javascript statement
```

- `Math.max` — function call/run
- `(10.12)` — arguments: pieces of data that you pass to the function in order for it to run

```
parseFloat('20.12343');//converting a string to a number
parseInt('20.12345');    //converting a string to whole number
Date.now();    //returns a timestamp which is number
// of milliseconds since god created earth,
// some people argue it's rather an amount
// of milliseconds since January 1st 1970 00:00
// 1578415831166 → epoch.now.sh → date
scrollTo({top:500,left:0,behavior:'smooth'}); //scrolling a page
```

Lesson 13 & 14

FUNCTION DEFINITION

```
function calculateBill(meal, taxRate = 0.05) {
  const total = meal * (1 + taxRate);
  return total;
}
```

- `function` — KEYWORD
- `calculateBill` — FUNCTION NAME
- `(meal, taxRate = 0.05)` — PARAMETERS/PLACEHOLDERS; `0.05` — DEFAULT VALUE
- `{` — SCOPE START
- `const total = meal * (1 + taxRate);` / `return total;` — FUNCTION BODY; `return total;` — RETURN STATEMENT
- `}` — SCOPE END

variable that is created inside of a function: block scoping.

temporary variable only available over inside of this function.

when the function is done running, this variable is cleaned up, "garbage collected", and it's no longer needed.

```
const myTotal = calculateBill(100, 0.13);
```

- `myTotal` — VARIABLE TO CAPTURE RETURNED VALUE
- `calculateBill` — NAME OR REFERENCE
- `(100, 0.13)` — CALL, RUN OR INVOKE; ARGUMENTS actual values

variable interpolation →
```
console.log(`Your total is ${myTotal}`);
```

function interpolation →
```
console.log(`Your total is ${calculateBill(20, undefined)}`);
```

#Lesson 15

Javascript functions are first class citizens. Javascript functions are values in themselves. They can be stored in variables. They can be passed into other functions.

function keyword

```
function add(a,b){
  const sum = a+b;
  return sum;
}
```

anonymous function

```
function (firstName) {
  return `Dr. ${firstName}.`;
};
```

function expression

```
const doctorize = function(firstName){
  return `Dr. ${firstName}.`;
};
```

arrow function

```
const add = (a,b) => a+b;
const inchToCM = inches => inches * 2.54;
const makeABaby = (first, last) => {
  const baby = {
    name: `${first} ${last}`,
    age: 0,
  };
  return baby;
};
// if your function needs to do some stuff
// inside of a block then a arrow function
// is perfectly fine:

// proper arrow function with implicit
// return of an object. not as easy
// readible as the previous one.
const makeABaby = (first, last) =>
({name: `${first} ${last}`, age: 0});
```

- addition to the javascript in the last couple years
- they offer a concise syntax and they are shorter
- handy to use with callback functions
- often a one liner
- they don't have their own scope
- they are an anonymous functions
- you always have to declare it and stick it to a variable
- implicit return (one line, no return keyword, no curly brackets)

```
const inchToCM = function(inches){
const inchToCM = (inches) =>
```

Curly brackets in JavaScript can be a creation of an object or they can be a block of code. In arrow function when you want to return an object in a curly bracket the interpreter/browser will think the curly bracket is a start of a block of code, and not the object.

To implicitly return an object in an arrow function you have to pop a set of parentheses around the thing (object) that you are returning and that will sort of contain it inside the parentheses and it won't think that it's actually the block of the function

Typing a return key is an explicit return

Implicit return is return without actualy having to type this keyword

IIFE

an immediately invoked function expression.
an immediately run function expression.

```
(function () {
  console.log('Running anon func');
  return 'You are cool';
})();
```

- anonymous function
- wrapped in parentheses.
- parentheses always run first in javascript and what it will do it will return a function value
- you can immediately run that function by putting parentheses on the end.
- it used to be very popular before we had modules, before what's called the block scope
- function declarers a own scope
- useful with closure

method

method is a function that lives inside of an object. there are 3 ways of defining it:

```
const wes = {
  name: 'Wes Bos',
  // Method!
  sayHi: function () {
    console.log(`Hey ${this.name}`);
    return 'Hey Wes';
  },
  // Short hand method
  yellHi() {
    console.log('HEY WES');
  },
  // Arrow function
  whisperHi: () => {
    console.log('hii im a mouse');
  }
}
```

callback functions

a function that get passed into another function and then is called by the browser at a later point in time

- something that will happen when something is done
- when somebody clicked something
- when this amount of time has passed
- can be defined outside of the handler.
- can be added as anonymous function as a value directly

Lesson #16 Debugging is tools + mindset. Most of debugging is done with console log, breakpoints, network requests.

console methods

console.log()
console.error()
Console.warn()
console.table()
↳ draws a table from given object
Console.count()
↳ counts how many times statement has been run.
console.group()
↳ neatly groups the console logs
Console.groupCollapsed()
↳ same as above, except collapsed

```
console.group('lorem');
console.log('ipsum');
console.warn('watch out!');
console.error('hey!');
console.groupEnd('lorem');
//start/end needs same value
```

callstack

stack trace, tells you what function called what function in console in what order and what type (named or anonymous).

grabbing elements

⌘⇧C

- when you inspect element then type in the console $0 it will select the picked element. Shortcut access to the element.
- "0" in $0 means the last element clicked. For example you can click 6 elements and then access them $0-$5 accordingly.
- $('p') finds first element that matches <p>.
- $$('p') access all <p> matching elements. You can only use this in the console. If jQuery is loaded on the page it won't work.

breakpoints

- type debuggers inside of a function. only when the devtools are open it will stop (pause) javascript from running
- sets out a breakpoint in the code at the certain specified time.
- you can set a breakpoint from a browser in the "sources" tab on developer tools (chrome) by opening the js file and clicking the chosen line.

network requests

network tab on developer tools (chrome) you can see what requests are are being sent of from the server

scope

Understanding how scope & closure works

break on:

1) attribute modification
2) subtree modification
   ↳ if a div was added
3) node removal
   ↳ if the node is removed

- Dev tool console → right click on element
- Browser only

XHR fetch breakpoint

any time fetch request is made, any time someone goes to external API, it will break.

Lesson #17 scope

- scope answers the question where are my variables in functions available to me.
- any time you declare a global variable it will be available anywhere in your application. you can access it through any javascript running on the page.
- in the console the global scope is a 'window'.
- all functions can be called from window.

Function scope - when variables are created inside of a function those variables are only ever available inside of this function.

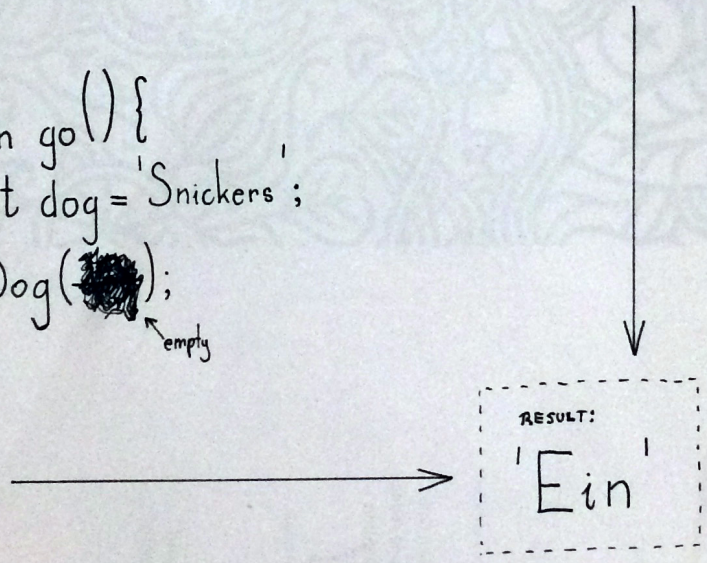
If variable is not found inside of a function, it will go a level higher, and if it's not available there it will go another level higher

You can name variables the same names if they are not already in the same scope. However it's not a good practice as it's limitating you.

```
const dog = 'Ein';

function logDog() {
  console.log(dog);
}

function go() {
  const dog = 'Snickers';
  logDog(); // empty
}

go();
```

Lexical scoping

or static scoping. Lookup for variable happens in the scope where the function was defined, not where it was run.



#Lesson 18 hoisting

Hoisting in JavaScript allows you to access functions and variables before they have been created.
Hoisting all the function and variable declarations to the top of the file.

2 things in javascript are being hoisted

function declarations — variable declarations

It is considered better to put all the functions on the top, or use separate modules (for.ex „utility functions", „math functions", and import them once you need them

```
console.log(age)
var age = 10;
```

>> console will log „undefined"

JavaScript will hoist the variable declaration but it won't hoist the actual setting of the value

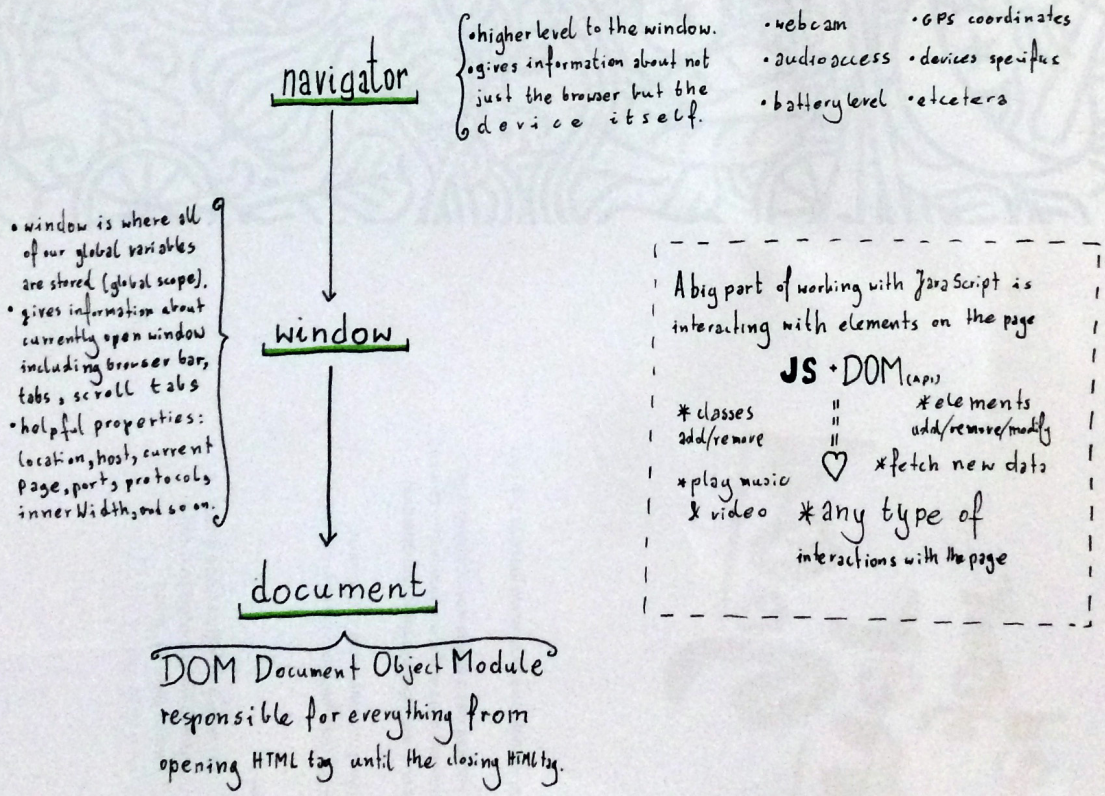
#Lesson 19 closures

Closure an ability to access a parent-level scope from a child-level scope, even after the parent function has been terminated.

```
function makeFunc() {
    var name = 'Mozilla';
    function displayName() {
        alert(name);
    }
    return displayName;
}

var myFunc = makeFunc();
myFunc();   //Mozilla
```

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2));   //7
console.log(add10(2));  //12
```

lesson 20 DOM introduction



#Lesson 21 DOM selecting elements

before you can work with items on the page you have to go get them

2 main ways on working with elements

selecting elements

generally you use them on „document." but that is not always the case

querySelector

finds first matching element

```
const p = document.querySelector('p');
```

querySelectorAll

finds multiple matching elements, returns a node list

```
const myImg = document.querySelectorAll('div .super img');
// searches for and returns all the img elements with 'div' as a one of the parent nodes and with '.super' class attached to it.
```

```
const item = document.querySelector('.item');
const itemImage = item.querySelector('img');
```

=> you can search for a new selector in another selector that has already been defined

```
document.getElementBy
    ID
    ClassName
    Name
    TagNames
    (xml only) <-> TagNameNS
```

=> an additional outdated way to select an element on the page

always put <script> just before </BODY> so that the script can operate on all the document after it is fully loaded. If you do it before it won't work as it can't operate on elements that don't yet exist.

=> unless... =>

```
function init(){
  const p = document.querySelector('p');
  console.log(p);
}
document.addEventListener('DOMContentLoaded', init);
// hardcore way around :O
```

#lesson 22 element properties and methods

```
const heading = document.querySelector('h2');
console.dir(heading); //shows all object's properties
heading.textContent = 'blabla2'; //using method
console.log(heading.textContent);
```

textContent vs innerText:

textContent gets the content of all elements including <script> & <style>. In contrast innerText only shows human-readible elements and also won't return hidden elements.

```
const element = document.querySelector('element');
console.log(element.innerHTML);
console.log(element.outerHTML);
```

```
<div id="element">
  <p>Hello World</p>   innerHTML
</div>                 outerHTML
```

```
element.insertAdjacentText(position, element)
<!-- before begin -->
<p>
  <!-- afterbegin -->
  foo!
  <!-- beforeend -->
</p>
<!-- afterend -->
```

Lesson 23 DOM working with classes

element → classes → methods

```
const pic = document.querySelector('.nice');
console.log(pic);
console.log(pic.classList); // gives you array of the classes of the element
```

proto methods for working with the classes

pic.className
property on the element
gives list of the classes

pic.classList
gives you array of all the classes.

```
pic.classList.add
              remove
              contains
              toggle
              ('open');
```

```
function toggleRound() {
  pic.classList.toggle('round');
}
pic.addEventListener('click', toggleRound);
```

misc: css:
rotate (1 turn)
rotate (360deg)

Lesson 24 THE DOM built in and custom data

```
<img class="custom" data-last="bos" data-name="wes" src="https://picsum.photos/200" />
const custom = document.querySelector('.custom');
console.log(custom.dataset);
console.log(`${custom.dataset.last}`);
```

```
console.log(pic.alt)
pic.alt = 'kups'
```

```
console.log(pic.naturalWidth)
pic.naturalWidth = 300
```

some attributes are for getting and setting. some are for getting only.

Lesson 25 THE DOM creating html

document.createElement

```
var element = document.createElement(tagName[, options]);
```

document.body
'body' is immediately available to us through document property

```
const myParagraph = document.createElement('p');
myParagraph.textContent = 'I am a P';
myParagraph.classList.add('special');
console.log(myParagraph);
```

```
const myImage = document.createElement('img');
myImage.src = 'https://picsum.photos/500';
myImage.alt = 'Nice photo';
console.log(myImage);
```

```
const myDiv = document.createElement('div');
myDiv.classList.add('wrapper');
console.log(myDiv);
```

```
document.body.appendChild(myDiv);
myDiv.appendChild(myParagraph);
myDiv.appendChild(myImage);
```

✗ **reflow.** Every single time that you use appendChild you are modifying the DOM which causes "reflow" of the browser.

```
myDiv.appendChild(myParagraph);
myDiv.appendChild(myImage);
document.body.appendChild(myDiv);
```

✓

- The best practice is to first create all the element separately and add it in one line in the end
- This example will actually cause 2 reflows. One the myDiv was added to the DOM. Second, when the image loads on the page.

targetElement.insertAdjacentElement(position, element)

works in the same pattern as in insertAdjacentText (#Lesson 21)

---

cloning node

list item element

 text

text is a child of an element

Let newClone = node.cloneNode([deep])

if [deep] is true it will clone whole subtree including text.
if it's empty, false, only node will be cloned.

#Lesson 26 the dom creating with strings

```
const item = document.querySelector('.item');

const myHTML = `
    <div class="wrapper">
        <h1> Wow! </h1>
        <p> Lorem Shmipsum</p>
    </div>
`;

item.innerHTML = myHTML;
console.log(item.innerHTML);
```

downsides

```
console.log(myHTML); // it's not an element, you can't apply element methods
console.log(typeof myHTML); // it's just a string
```

document.createRange()

document.createContextualFragment()

turn string into a DOM element

```
const myFragment = document.createRange().createContextualFragment(myHTML);
document.body.appendChild(myFragment);
```

⚠ beware of XSS (cross site scripting) security threat

#Lesson 27 the dom traversing

traversing

going up, down, from west to east etc. to move or travel through an area

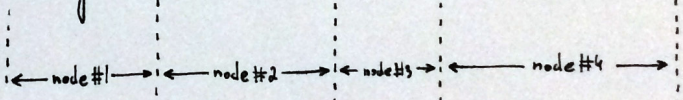
---

useful element's methods

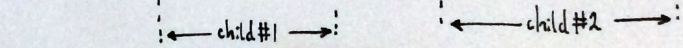
- childElementCount
- children
- firstElementChild
- lastElementChild
- previousElementSibling
- nextElementSibling
- parentElement
- childNodes

 I am Jack, I love miso and wakame!

jacek.childNodes



jacek.children



removing elements

```
const p = document.createElement('p');
p.textContent = 'I will be removed';
document.body.appendChild(p);
p.remove();
console.log(p);
// p still exists in javascript memory
// it's only removed from the DOM
```

Lesson #29 Events, event listener

```
const button = document.querySelector('.button');

button.addEventListener('click', handleClick);
```

- `button` — go get something
- `addEventListener` — listen for something
- `'click'` — listen for something
- `handleClick` — do something

binding: taking a function and listening for a specific action against the element

removing event listeners: unbinding

```
button.removeEventListener('click', handleClick)
```

* anonymous functions cannot be removed *

• Listen on multiple items

```
const buyButtons = document.querySelectorAll('button.buy');

function buyItem() {
  console.log('buying item');
}

buyButtons.forEach(button => button.addEventListener('click', buyItem));
```

Lesson #30 events, targets, bubbling, propagation, capture

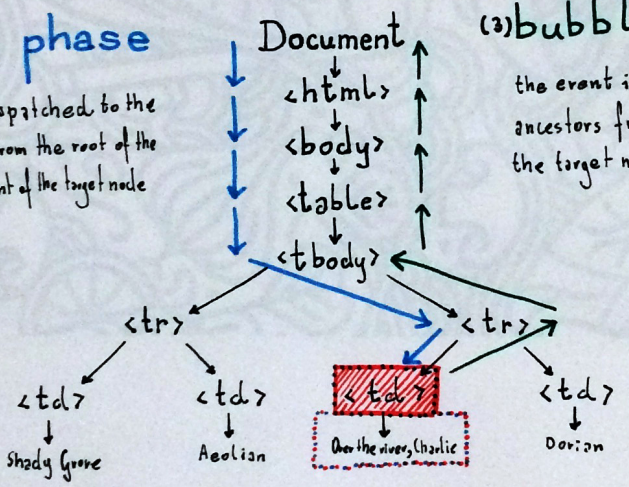
- graphical representation of an event dispatched in a DOM tree using the DOM event flow

(1) capture phase

the event is dispatched to the target's ancestors from the root of the tree to the direct parent of the target node

(3) bubbling phase

the event is dispatched to the target's ancestors from the direct parent of the target node to the root of the tree



(2) target phase

the event is dispatched to the target node

• phases

The event is dispatched following a path from the root of the tree to the target node. It can be then handled locally at the target node level or from any target's ancestors higher in the tree. The event dispatching occurs in 3 phases: capture, target, bubbling.

• propagation

activating two or more event listeners at the same time.

`event.stopPropagation();` method for stopping propagation

`parseFloat();` keeps decimals

`parseInt();` does not keep decimals

* capturing phase is rarely used. To catch an event on the capturing phase, we need to set the handler *capture* option to *true*

```
elem.addEventListener(..., {capture: true})
```

```
const photo = document.querySelector('.photo');

photo.addEventListener('mouseenter', function(e){
  console.log(e.currentTarget);
  console.log(this);
});
```

`e.currentTarget === this`

event.target vs **event.currentTarget**

event.target	event.currentTarget
thing that actually got clicked	thing that fired event listener

```
photo.addEventListener('mouseenter', e=>{
  console.log(e.currentTarget);
  console.log(this);
});
```

if you change the function to an arrow function, "this" won't be scoped anymore. (it won't work). don't use "this" in event listeners callbacks.

Lesson #31 EVENTS prevent default and fire events

preventDefault()

cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur

FOR EXAMPLE:
- clicking on a "Submit" button, prevent it from submitting a form
- clicking on a link, prevent the link from following the URL

```
const signupForm = document.querySelector('[name="signup"]');
// the best way to grab a form is by a name, not by a class
```

corresponding mouse and keyboard events:

MOUSE EVENT	KEYBOARD EVENT
mousedown	keydown
mouseup	keyup
click	keypress
mouseover	focus
mouseout	blur

keycode.info

event.key → k
event.which → 75
event.code → KeyK

Lesson #32 EVENTS accessibility, gotchas and keyboard codes

- buttons → are to be used for actions that happen inside of the application
- Links → are used to change the page
- Save → bad practice. the link that doesn't go anywhere is not a link
- things that are not keyboard accesible should not have clicks registered to them. For example images. If you mean for something to be a button that is not a button you can add role="button", but it's better to put image into a button
- <img class="photo" tabindex="0"... allows browser to select element through TAB button

```
const photo = document.querySelector('.photo');
function handlePhotoClick(event) {
  if(event.type === 'click' || event.key === 'Enter') {
    console.log('You clicked the photo');
    console.log(event.key);
  }
}
photo.addEventListener('click', handlePhotoClick);
photo.addEventListener('keyup', handlePhotoClick);
```

Lesson #33 ETCH-A-SKETCH

HTML
```
<canvas width="1600" height="1000" id="etch-a-sketch"></canvas>
```

JS
```
const canvas = document.querySelector('#etch-a-sketch');
const ctx = canvas.getContext('2d');
const {width, height} = canvas;
ctx.lineJoin = 'round';
ctx.lineCap = 'round';
ctx.lineWidth = 20;
ctx.strokeStyle = `hsl(100, 100%, 50%)`;
ctx.beginPath();
ctx.moveTo(x, y);
ctx.lineTo(x, y);
ctx.stroke();
ctx.clearRect(0, 0, width, height);
```

DESTRUCTURING

destructuring allows us to extract data from arrays, objects, maps and sets into their own variable. It allows us to extract properties from an object or items from an array, multiple at a time.

```
const person = {
  first: 'Scottes',
  last: 'Bossinski',
  pets: {
    dog: 'Skooper',
    cat: 'Captain Claw',
  }
}
const {first, last} = person;
const {dog, cat} = person.pets;
```

Lesson #34 click outside modal

CSS:

```
.modal-outer {
  opacity: 0;
  pointer-events: none;
  ...
}

.modal-outer.open {
  opacity: 1;
  pointer-events: all;
}
```

HTML

```
<div class="modal-outer">
  <div class="modal-inner">
    <p> w o o f </p>
  </div>
</div>
```

JS:

```
modalOuter.addEventListener('click', function (event) {
  const isOutside = !event.target.closest('.modal-inner');
  if (isOutside) {
    modalOuter.classList.remove('open');
  }
});

window.addEventListener('keyup', event => {
  if (event.key === 'Escape') {
    closeModal();
  }
});
```

`closest()`: if you click on the modal-inner element or inside of it it will return find the modal-inner, the reason closest() is bubbling up. if you click anything outside it won't find it

Lesson #35 scroll events and intersection observer

HTML

```
<button class="accept" disabled>Accept</button>
```

JS

```
const terms = document.querySelector('.terms-and-conditions');
const button = document.querySelector('.accept');

function obCallback(payload) {
  if (payload[0].intersectionRatio === 1) {
    button.disabled = false;
    ob.unobserve(terms.lastElementChild); // stop observing the button
  }
}

const ob = new IntersectionObserver(obCallback, { root: terms, treshold: 1 });
ob.observe(terms.lastElementChild);
```

CSS

```
button[disabled] {
  opacity: 0.1;
  transform: translateX(-200%) scale(0.5);
}

.terms-and-conditions {
  overflow: scroll;
}
```

Intersection Observer API

provides a way to asynchronously observe changes in the intersection of a target element with an ancestor element or with a top-level document's viewport.

- lazy-loading of images or other content as a page is scrolled.
- implementing "infinite scrolling" websites, where more and more content is loaded and rendered as you scroll, so that the user doesn't have to flip through pages.
- reporting of visibility of advertisements in order to calculate revenues
- deciding whether or not to perform tasks or animation processes based on whether or not the user will see the result.

#Lesson 36 tabs

JS

```
const tabs = document.querySelector('.tabs');
const tabButtons = tabs.querySelectorAll('[role="tab"]');
const tabPanels = Array.from(tabs.querySelectorAll('[role="tabpanel"]'));

function handleTabClick(event) {
  tabPanels.forEach(panel => {
    panel.hidden = true;
  });
  tabButtons.forEach(tab => {
    tab.setAttribute('aria-selected', false);
  });
  event.currentTarget.setAttribute('aria-selected', true);
  const { id } = event.currentTarget;   // → destructuring
  const tabPanel = tabPanels.find(
    panel => panel.getAttribute('aria-labelledby') === id
  );
  tabPanel.hidden = false;
}
tabButtons.forEach(button => button.addEventListener('click', handleTabClick));
```

"find()" can only work on arrays. The "Array.from(x)" transforms node list into an array

HTML

```
<div class="tabs">
  <div role="tablist" aria-label="pet">
    <button role="tab" aria-selected="true" id="dog">
      doggito
    </button>
  </div>
  <div role="tabpanel" aria-labelledby="dog">
    <p> dogito perrito muy lindo</p>
  </div>
</div>
```

Lesson #37 bedmas, order of operations in algebra basics and computer programming

commutative

B	E	D	M	A	S
brackets	exponents	division	multiplication	addition	subtraction

Lesson #38 if statements, function returns, truthy, falsy, regular expression

```
function slugify(sentence, lowercase){
    let slug=sentence.replace(/\s/g,'-');
        if (lowercase) {
            return slug.toLowerCase();
        } //else
        return slug;
}
```

(condition: lowercase; regex: /\s/g)

String.prototype.replace()

The replace() method returns a new string with some or all matches of a pattern replaced by a replacement. The pattern can be a string or RegExp, and the replacement can be a string or a function to be called for each match. If pattern is a string, only the first occurence will be replaced. The original string is left unchanged.

Regular expression flags

flag	description	corresponding property
g	global search	RegExp.prototype.global
i	case-insensitive search	RegExp.prototype.ignoreCase
m	multi-line search	RegExp.prototype.multiline
s	allows "." to match newline characters	RegExp.prototype.dotAll
u	"unicode"; treat a pattern as a sequence of unicode code points	RegExp.prototype.unicode
y	perform a "sticky" search that matches starting at the current position in the target string.	RegExp.prototype.sticky

- /[a-zA-Z09]/ === /[a-z09]/i

Regular expression

A regular expression (shortened as regex or regexp; also referred to as rational expression) is a sequence of characters that define a search pattern. Usually such patterns are used by string searching algorithms for "find" or "find and replace" operations on strings, or for input validation. It is a technique developed in theoretical computer science and formal language theory.

What can you use "==" for?

- You should avoid doubleequal signs at all costs.

You can use it to check if variable is *null* OR *undefined*.

```
>null === undefined
<false
>null == undefined
<true
```

---

bang operator

```
>10 !== '10'
<true
```

one more time, here we go, bang operator!

```
>10 != '10'
<false
```

```
>'awesome'.includes('wes');
<true
```

```
>'awesome'.includes('scott');
<false
```

Truthy and Falsy

As well as a type, each value also has an inherent boolean value, generally known as either truthy or falsy. Some of the rules are a little bizzare so understanding the concepts and effect on comparision helps when debugging JavaScript applications.

truthy

- '0' (a string containing a single zero)
- 'false' (a string containing the text "false")
- [] (an empty array)
- {} (an empty object)
- function(){} (an "empty" function)

falsy

- false
- 0 (zero)
- '' or "" (empty string)
- null
- undefined
- NaN

Lesson 39 # coercion, ternaries, conditional abuse

```
>const name = 'spike';
>name
<"spike" //string
>!name
<false //boolean
>!!name
<true //boolean
```

```
const a = 42;

const b = a + "";
//implicit coertion
const c = String(a);
//explicit coertion
```

Converting Values

Converting a value from one type to another is often called „type casting", when done explictly, and „coertion" when done implicitly. (forced by the rules of how a value is used).

ternary

shorthand if statement

(condition) ? (what to do if it's true) : (what to do if it's false);

```
const count = 2;
let word;
if (count === 1) {
    word = 'item';
} else {
    word = 'items';
}
const sentence = `You have ${count} ${word} in your cart`;
```

- ternary approach:

```
const word = count === 1 ? 'item' : 'items';
```

Intl

Intl.PluralRules object enables plural-sensitive formatting and plural-related language rules

The Intl object is the name space for the ECMAScript Internationalization API, which provides language sensitive string comparison, number formatting, and date and time formating. The Intl object provides access to several constructors as well as funcionality common to the internationalization constructors and other language sensitive functions.

```
const sentence = `You have ${count} item${count === 1 ? '' : 's'} in your car`;
```

The AND AND trick

```
function check1() {
    console.log('running check 1');
    return true;
}
function check2() {
    console.log('running check 2');
    return false;
}
function check3() {
    console.log('running check 3');
    return true;
}
if (check1() && check2() && check3()) {
    console.log('all checks passed');
} else {
    console.log('some checks failed');
}
<- "running check 1"
<- "running check 2"
<- "some checks failed"
```

```
function showAdminBar() {
    console.log('admin bar');
}
const isAdmin = true;
isAdmin ? showAdminBar() : null;
```

```
isAdmin && showAdminBar();
```

{ isAdmin && <AdminBar/> } //react

```
// if (isAdmin) {
//     showAdminBar();
// }
// if (isAdmin) showAdminBar();
```

Lesson #40 case switch and animating a turtle with css variables

switch statement

- clearly defined cases
- can't use >20 as in if statement

```
switch (expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

CSS custom properties

Custom properties (sometimes referred to as CSS variables or cascading variables) are entities defined by CSS authors that contain specific values to be reused throughout a document. They are set using custom property notation (e.g., --main-color:black;) and are accessed using the var() function (e.g., var(--main-color);).

CSS example

```
.turtle {
  position:relative
  --x:0px;
  --y:0px;
  transform:translateX(var(--x))→
  ↳translateY(var(--y));
}
```

JS example

```
(...)//working out x & y
turtle.setAttribute('style',
  `
    --x: ${x}px;
    --y: ${y}px;
  `
);
_ _ _ _ _ _ _ _ _ _ _ _
turtle.style.background='red'
turtle.style['background']='green'
```

Lesson #41 intervals and timers

setTimeout()

calls a function or evaluates an expression after a specified number of miliseconds

- clearTimeout()

```
function setImmediateInterval(funcToRun, ms) {
  funcToRun();
  return setInterval(funcToRun, ms);
}
// ~~first~~ second run will start after the set time
// first run will start immediately
```

setInterval()

calls a function or evaluates an expression at specified intervals (in miliseconds).

- clearInterval()

- the only way to clear timer or interval is to save a reference to it in a variable (which is a number)

Lesson #something extra#

CSS Grid Layout

CSS Grid Layout excels at dividing a page into major regions or defining the relationship in terms of size, position, and layer, between parts of a control built from HTML primitives.

Like tables, grid layout enables an author to align elements into columns and rows. However, many more layouts are either possible or easier with CSS grid than they were with tables. For example, a grid container's child elements could position themselves so they actually overlap and layer, similar to CSS positioned elements.

```
.one {
  grid-column: 1/3;
  grid-row: 1;
}

.two {
  grid-column: 2/4;
  grid-row: 1/3;
}

.three {
  grid-column: 1;
  grid-row: 2/5;
}

.four {
  grid-column: 3;
  grid-row: 3;
}

.five {
  grid-column: 2;
  grid-row: 4;
}

.six {
  grid-column: 3;
  grid-row: 4;
}

.wrapper {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-gap: 10px;
  grid-auto-rows: minmax(100px, auto);
}
```

Lesson #42 Objects

- everything in JavaScript is an object
- objects allow us to group together properties (keys) and values, store related data, store funcionality, create your own custom types.
- objects are used where the order of the properties doesn't matter
- although the properties of an object can change, the object itself can't be overwritten entirely (const)
- you can create new variable and assign the frozen state (immutable) of ony object `const wesFroze = Object.freeze(wes)`
- to remove property `delete wes.job` (returns true if successful)
- a function inside of an object is referred to as a method of that object
- if you pass in an object (or array) into a function and you modify that object (or array), the external object will also be updated as opposed to booleans, numbers, strings
- ways of declaring: `const myObject={};` or `const myObject=new Object({});`

Optional chaining

The optional chaining operator `?.` permits reading the value of a property located deep within a chain of connected objects without having to expressly validate that each reference in the chain is valid. The `?.` operator functions similarly to the `.` chaining operator, except that instead of causing an error if reference is nullish (**null** or **undefined**), the expression short-circuits with a return value of **undefined**. When used with function calls, it returns **undefined** if the given function does not exist.

- `const city = user?.address?.city;`
- if user or address is undefined or null, then the value of city becomes undefined

```
const propertyToCheck = 'clothing';
console.log(wes.propertyToCheck); ✗
console.log(wes[propertyToCheck]); ✓
```

```
console.log(wes.age); ✓
console.log(wes['age']); ✓
```

```
const person = {
  age: 100,
  name: 'wes',
};
```

Lesson #43 Object reference vs values

```
const person1 = {
  first: 'wes',
  last: 'bos',
};
```

`===` *false!*

```
const person2 = {
  first: 'wes',
  last: 'bos',
};
```

when you're comparing objects it is done by the reference to the object itself, not the values inside of it

DEAL WITH IT

```
const person3 = person1;
person3.first = 'Larry';
console.log(person3.first);
// "Larry"
console.log(person1.first);
// "Larry" :0
```

When objects, arrays, sets, maps are „copied" by reference, we create a variable that points to the original variable instead of making a copy of it.

spread operator

```
const person4 = { ...person1 }
```

spread, actual copy of an object.
spread creates a shallow copy, up to one level deep down the object

```
person4.first = 'wesly';
person4.clothing.shirts = 100;
delete person4.clothing.shirts;
```

wouldn't affect person1

it would affect person1

merging objects with spread

```
const inventory = {...meatInventory, ...veggieInventory}
```

any duplicates will be overwritten by the latest occurence

Lodash

Lodash is a JavaScript library which provides utility functions for common programming tasks using the functional programming paradigm

```
const person5 = _.cloneDeep(person1);
```

deep copy, external objects are copied as well, so the new, cloned object is completely independent from the oldone.

#Lesson 44 Maps

Map

The Map object holds key-value pairs and remembers the original insertion order of the keys. Any value (both objects and primitive values may be used as either a key or a value. A map object iterates its elements in insertion order - a for...of loop returns an array of [key, value] for each iteration

Objects vs Maps

Object is similar to map - both let you set keys to values, retrieve those values, delete keys and detect whether something is stored at a key. For this reason (and because there were no built in alternatives) Objects have been used as Maps historically (ES5 and earlier).

	map	object
accidental keys	A map does not contain any keys by default. It only contains what is explicitly put into it.	An Object has a prototype, so it contains default keys that could collide with your own keys if you're not careful.
key types	A Map's keys can be any value (including functions, objects, or any primitive)	The keys of an Object must be either a String or a Symbol
key order	The keys in Map are ordered. Thus, when iterating over it, a Map object returns keys in order of insertion.	The keys of an Object are not ordered
size	The number of items in a Map is easily retrieved from its size property	The number of items in an Object must be determined manualy
iteration	A map is an iterable, so it can be directly iterated	Iterating over an Object requires obtaining its keys in some fashion and iterating over them
performance	Performs better in scenarios involving frequent additions and removals of key-value pairs	Not optimized for frequent additions and removals of key-value pairs.
JSON	JSON does not support maps, sets, etc.	JSON does support Objects, arrays.

working with maps

```
const myMap = new Map();
myMap.set('name', 'wes');
myMap.set(2, 'this is a number');
myMap.set(person, 'an object reference');
myMap.set(true, 'bool1');
myMap.get(person);
myMap.delete('name');
myMap.size;
myMap.keys();
myMap.values();
myMap.entries();
myMap.forEach((value,key,map)=>{
  console.log(`${key}:${value}`);
});
```

```
let recipeMap = new Map([
    ['cucumber', 500],
    ['tomatoes', 350],
    ['onion',  50],
]);
```

trailing comma
comma-dangle

```
// iterate over keys (vegetables)
for (let vegetable of recipeMap.keys()) { alert (vegetable);} //cucumbers, tomatoes, onion

//iterate over values (amounts)
for (let amount of recipeMap.values()){alert (amount);} // 500, 350, 50

//iterate over [key, value] entries
for (let entry of RecipeMap) { // the same as of recipeMap.entries()
   alert (entry); // cucumber, 500 (and so on)
}
```

Lesson #45 Array #46 #47 #48

- used for holding a list of items where the order matters
- each thing in array is called an item and its position in array is called an index
- number of the items in array is called length
- array is not its own type, it is an object
- array must use integer as element indexes
- array's length and the type of its elements can change
- [...'wes'] → ['w', 'e', 's'] (3)
- prototype method e.g., numbers.pop(5);
- utility method e.g., Array.find();
- Array.from({length:10}) returns array with 10 empty spots

- Array.isArray()
- Array.of()
- concat()
- copyWithin()
- entries()
- every()
- fill()
- filter()
- find()
- findIndex()
- flat()
- flatMap()
- forEach()
- includes()
- indexOf()
- join()
- keys()
- LastIndexOf()
- map()
- pop()
- push()
- reduce()
- reduceRight()
- reverse()
- shift()
- slice()
- some()
- sort()
- splice()
- toLocaleString()
- toString()
- unshift()
- values()

- if you try splitting string that contains emoji split() function will split emoji into a few gibrish items. Es6 proposes using spread, stackoverflow says use grapheme-splitter
- if you want to use mutable method but you don't want original array to be mutated you have to take a copy of the original array.

names.push('lux') → [...names,'lux']; names.unshift('poppy') → ['poppy',...names]

```
const newBikes = [
  ...bikes.slice(0,2),
  'benotto',
  ...bikes.slice(2),
];
```

```
const newBikes2 = [
  ...newBikes.slice(0,3),
  ...newBikes.slice(4),
];
```

SPREAD IT LIKE YOU MEAN IT

```
function deleteComment(id,comments){
  const commentIndex = comments.findIndex(comment => comment.id === id)
  return [
    ...comments.slice(0,commentIndex),
    ...comments.slice(commentIndex+1),
  ];
}
```

```
function createRange(start,end){
  const range = Array.from({length:end-start+1},
      function(_item,index){
        return index + start;
      });
      return range;
}
```

```
Object.entries(meats).forEach(entry => {
  const key = entry[0];
  const value = entry[1];
  console.log(key,value);
});
```

→ destructuring opportunity

```
+ Object.entries(meats).forEach(entry => {
    const [key,value] = entry;
    paper.log(key,value);

+ Object.entries(meats).forEach(([key,value]) => {
    paper.log(key,value);
});
```

```
function findByWord(word){      // higher order function
  return function(singlefeedback){
    return singleFeedback.comment.includes(word);
  }
}
const burgRating = feedback.find(findByWord('burg'));
const smoothieRating = feedback.find(findByWord('smoothie'));

function filterByMinRating(minRating){
  return function(singleFeedback){
    return singleFeedback.rating > minRating;
  }
}
const goodReviews = feedback.filter(filterByMinRating(4));
const burgRatings = feedback.filter(findByWord('burg'));
const legitRatings = feedback.filter(single => single.rating !== 1);
```

```
const numbersSorted = numbers.sort((firstItem,secondItem) => firstItem - secondItem);
function numberSort = (a,b){
    return a-b;
}
console.log(orderTotals.sort(numberSort));
const productsSortedByPrice = Object.entries(prices).sort(function(a,b){
    const aPrice = a[1];
    const bPrice = b[1];
    return aPrice - bPrice;
});
console.table(Object.fromEntries(productsSortedByPrice));
```

Lesson #49 Looping and iterating – forEach

Array.prototype.forEach()

The forEach method executes a provided function once for each array element.

arr.forEach(callback(currentValue[, index [, array]])[, thisArg])

- use:
  - loop over some data
  - do something with each piece of data:
    - log a value
    - attach event listener for a click
    - display data on a page
- side effects
  - when you are inside of the function and you reach outside of that function to do something else

map — filter — reduce — map — filter — reduce — map — filter — reduce — map — filter — reduce — map — filter — reduce — map — filter — reduce

The Big Three Legendary Sannin

Map()

map([ ], cook)
⇒ [ ]

Filter()

filter([ ], isVegetarian)
⇒ [ ]

Reduce()

reduce([ ], eat)
⇒ [ ]

Lesson #50 Looping and iterating – mapping

Map()

- takes in data, performs an operation, spits it out
- will always return the same length of array that it starts with
- "data massaging"

```
const fullNames = ['wes', 'kait', 'poppy'].map(name => `${name} bos`);
▶ Array(3) ["wes bos", "kait bos", "poppy bos"]
```

interpolation to interpolate a variable

```
function bosify(name){
  return `${name} Bos`;
}
function capitalise(word){
  return`${word[0].toUpperCase()}${word.slice(1)}`; // strings work in similar way to arrays
const fullNames = ['wes', 'kait', 'poppy'].map(capitalise).map(bosify); //you can chain maps on each other
```

```
const ordersTotalWithTax = ordersTotals.map(total => total * 1.17);
```

```
const cleanPeople = people.map(function (person){
    const birthday = new Date(person.birthday).getTime(); //converts the date into a timestamp
    const now = Date.now(); //creates a timestamp, which counts milliseconds since the creation of the world
    const age = Math.floor((now - birthday) / 31536000000); //year in milliseconds (ms)
    return {
      age, // same as age : age,
      name: `${person.names.first} ${person.names.last}`,
    }
});
```

miscellaneous self-explanatory:

```
'x'.repeat(199);
Array(3).fill('x');
Array(5).fill('y').join(' ');
```

Lesson #51 Looping and iterating – higher order functions

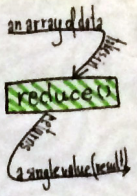
```
function findByProp(prop, propWeAreLookingFor){ // flexible higher order function
    return isStudent(student){
        return student[prop] === propWeAreLookingFor;
    }
};
const student = students.find(findByProp('id', '5662'));
const student2 = students.find(findByProp('first_name', 'Michi'));
```

Lesson #52 Looping and iterating - reduce

Array.prototype.reduce()

The reduce() method executes a reducer function (that you provide) on each element of the array, resulting in a single output value.

- The reducer function takes four arguments:
  1. accumulator (acc)
  2. current value (cur)
  3. current index (idx)
  4. source array (src)
- Your reducer function's returned value is assigned to the accumulator, whose value is remembered across each iteration throughout the array, and ultimately becomes the final, single resulting value.




```
arr.reduce(callback( accumulator, currentValue[, index[, array]] )[, initialValue])
```

- counting totals - example

```
function tallyNumbers(tally,currentTotal){
  console.log(`The current tally is ${tally}`);
  console.log(`The current total is ${currentTotal}`);
  console.log('--------');
  return tally + currentTotal;
}
const allOrders = orderTotals.reduce(tallyNumbers,0);
```

- counting instances - example

```
function inventoryReducer(totals,item){
// totals[item.type] = totals[item.type] + 1 || 1;          //logical OR operator
// // totals[item.type] ? totals[item.type] +1 : totals[item.type] = 1;  //ternary operator
// // totals[item.type] = (totals[item.type] || 0) + 1;      //logical OR operator
  return totals;
}
const inventoryCounts = inventory.reduce(inventoryReducer,{});
const totalInventoryPrice = inventory.reduce((acc,item) => acc + item.price, 0);
```

- redux challenge exercise. Lesson #53

```
const text = `*<--contains thousands of random letters,
                numbers and symbols-->*`;

const isValidChar = char => char.match(/[a-z0-9]/i);
const lowercase = char => !char.match(/[0-9]/) && char === char.toLowerCase() ? char : char.toLowerCase();
function instanceCounter(counts,char){
    counts[char] ? (counts[char]+=1) : (counts[char]=1);
    return counts;
}
function sortByValue(a,b) {
    return a[1] - b[1];
}
const letters = [...text]
    .filter(isValidChar)
    .map(lowercase)
    .reduce(instanceCounter,{});
const sortedResult = Object.entries(letters).sort(sortByValue);
```

Array.length

The value of the **length** property is an integer with a positive sign and a value less than 2 to the 32nd power (2^{32})

```
const nameListA = new Array(4294967296); // 2 to the 32nd power
Const nameListC = new Array(-100); // negative sign

console.log(nameListA.length); // Range error: Invalid array length
console.log(nameListC.length); // Range error: Invalid array length

const nameListB = [];
nameListB = Math.pow(2,32)-1 //set array length 1 less than 2^32
console.log(nameListB.length); //4294967295
```

Lesson #54 Looping and iterating - for, for in, for of, while

For

- traditionally used to loop over something. an array or numbers
- nowadays it's easier to use for.Each, map, reduce
- useful on canvas because you can increment by 4 (one pixel takes 4 spots in array)

```
for([InitialExpression];[condition];
    [incrementExpression])
    statement
```

variable is scoped to this block. you're unable to reuse it

```
for(let i=0;numbers.length;i++){
  console.log(numbers[i]);
}
```

For of

- fairly new to the language
- used for looping over iterables
- mostly used for arrays/string
- useful for promises, allows you to use await
- can handle emojis when logging letters
- spread can also log emojis

```
for(const letter of name){
    console.log(letter);
}
```

- doesn't give you index, doesn't allow you to use filter etc.

```
for(const number of numbersArray){
    console.log(number);
}
```

For in

- used for looping over keys of an object

```
for(const prop in wes){console.log(prop);}
```

- For in vs Object.entries():

```
const baseHumanStats = {
  feet:2, arms:2, eyes:2, head:1, };
function Human(name){
  this.name = name;
}
Human.prototype = baseHumanStats;
const wes2 = newHuman('wes');
console.log(Object.entries(wes2));
//doesn't include prototype methods & properties
for(const prop in wes2) {
  console.log(prop);
}
//logs everything from prototype that
  it was made from (so all the prototype's
  methods and properties.
```

While

- takes condition and runs indefinitely until condition is false

while checks the condition before the first run

```
let cool=true;
let i = 1;
while(cool===true){
 console.log('You are cool.');
 i+=1;
 if(i>100){
   cool=false;
 }
}
```

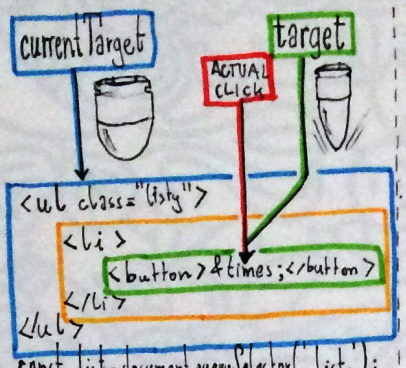
do while checks the condition after the first run

```
do
  statement
while (condition);
```

lesson #57 shopping form with custom events, delegation

- get all the HTML elements
- create an array to hold our state
- add an event listener to form
  - wait for submit
  - grab submitted item from the form
  - add preventDefault() to stop the page from sending a request (reloading)
- store the data about the newly submitted item in the items array
  - const item = {
    name,
    id: Date.now(), //unique id
    complete: false, //checkbox status
    }
  - push the item into our state
  - clear the form e.target.reset()
- create a display function that will rerender the list upon state update
- use event system in javascript to fire off our own events, and then listen for those events.
  - fire off a custom event that will tell anyone else who cares that the items have been updated.
- create a function that will mirror the data to local storage that will fire off upon custom event 'itemsUpdated'
- restore the data from the local storage when loading the page
- event delegation: we listen for the 'click' on the list <ul> but then delegate the click over to the button if that what was clicked.
- create an item delete function
  - filter the state and exclude the chosen item by the previously passed id.
  - dispatch 'itemsUpdated' event
- create a function that takes care of checking and unchecking of the data
  - find the chosen item
  - change its 'complete' property to opposite:

  `itemRef.complete = !itemRef.complete` // great trick to change (switch) variable's boolean value to opposite
  - dispatch 'itemsUpdated' event




```
const listy = document.querySelector('.listy');
listy.addEventListener('click', callback function);
```

Parcel

Blazing fast, zero configuration, web application bundler. Parcel is a build tool. Build tool takes all of your different parts of your website (images, html files, js files, css preprocessor files (sass/less) and converts it into different bundles: developer or production. It takes files of your website and turns them into something browsers totally understand

- to install globally: (sudo) install -g parcel-bundle

JSON

The JSON object contains methods for parsing **JavaScript Object Notation (JSON)** and converting values to JSON. It can't be called or constructed, and aside from its two method properties, it has no interesting functionality of its own.

- JSON.stringify(myObject);
- JSON.parse(myObject);

window.localStorage

- mini database that lives inside of your browser and allows you (within same browser to pick up where you last left.
- ability to save data in users browser
- DEVELOPER TOOLS/APPLICATION/LOCAL STORAGE
- localStorage.setItem('name', 'wes')
- localStorage.getItem('name');
- when storing an object you have to first ~~parse~~ stringify it to text with JSON. when retrieving item which is an object you have to parse it with JSON.

custom event

- When you click on something the browser dispatches a click event. When you focus on something the browser dispatches focus event. To use a custom event you need to dispatch an event from chosen place.
- dispatchEvent() lives on all DOM elements.
- Custom events let you pull the function you want to run afterwards out of the script itself.
- You can include an event listener in a different script or file, and attach multiple event listeners to the same event.

```
list.dispatchEvent(new CustomEvent('itemsUpdated'));
list.addEventListener('itemsUpdated', displayItems);
list.addEventListener('itemsUpdated', mirrorToLocalStorage);
```

event delegation

Event delegation allows you to avoid adding event listeners to specific nodes; instead, the event listener is added to one parent. That event listener analyzes bubbled events to find a match on child elements.

```
list.addEventListener('click', function(e) {
  const id = parseInt(e.target.value);
  if(e.target.matches('button')) {
    deleteItem(id);
  }
  if(e.target.matches('input[type="checkbox"]')) {
    markAsComplete(id);
  }
});
```

#Lesson 58 building a gallery exercise

```
if (!gallery){
    throw new Error('No gallery found!');
}
```

- Error objects are thrown when runtime errors occur. The Error object can also be used as a base object for user-defined exceptions. Runtime errors result in new Error objects being created and thrown.

- modal, html structure

```
<div>
 <div
  <button>
  <figure>
   <img>
   <figcaption>
    <h2>
    <p>
   </figcaption>
  </figure>
  <button
 </div>
</div>
```

- modal, another way of escape

```
if(e.target === e.currentTarget){
    closeModal();
}
```

- accesing data attribute

```
<img ... data-description="Jenny">
el.dataset.description
```

#Lesson 59 building a slider

Slider

„A fancy class adder & remover"
~WesBos

- if you need to pass an argument in callback function in event listener you need to use arrow function.

- I love to eat pizza

el.nextSibling ⟶ " to eat "
el.nextElementSibling ⟶ pizza

#Lesson 60 the new keyword, classes, prototypes, new & this keyword

new operator

The new operator lets developers create an instance of a user-defined object type or of one of the built-in object types that has a constructor function.
The new keyword does the following things:

I: creates a blank, plain JavaScript object;
II: links (sets the constructor of) this object to another object;
III: passes the newly created object from Step I as the this context;
IV: returns this if the function doesn't return its own object

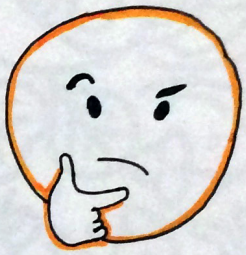
new *constructor*[([arguments])]

constructor: a class or function that specifies the type of the object instance

[arguments]: a list of values that the constructor will be called with

```
let s_prim = 'foo';
let s_obj = new String(s_prim)

console.log(typeof s_prim) // logs "string"
console.log(typeof s_obj)  // logs "object"
```

- JavaScript distinguishes between String objects and primitive string values.



```
>> span instanceof Element && span instanceof Node && span instanceof HTMLSpanElement
<- true
```

```
>> typeof myArray
<- object
```

```
>> myArray instanceof Array
<- true
```

```
function Pizza(){
  console.log('Making a pizza');
}

const pepperoniPizza = new Pizza();
console.log(pepperoniPizza);
▶ Pizza {}
console.log(pepperoniPizza.constructor);
▶ Pizza () {
    console.log('Making a pizza');
  }
console.log(pepperoniPizza instanceof Pizza);
▶ true
```

- when you use *new* keyword on a function it creates a new object that is an instance of whatever function you have made it from.

- it tells what function made this object

```
const name = ['john', 'neo']  ⟶ array literal syntax
const name = new Array('john', 'neo');

const john = {name: 'john'};  ⟶ object literal syntax
const John = new Object({name: 'john'});
```

✖ Date doesn't have literal syntax, therefore we have to use *new*

```
const myDate = new Date('August 30, 1990');
```

Lesson #61 the this keyword

this

- this refers to the instance of the object that the function is bound to;
- a function's this keyword behaves a little differently in JavaScript compared to the other languages. It also has some differences between strict mode and non-strict mode.

```
const button1 = document.querySelector('.one');
const button2 = document.querySelector('.two');
function tellMeAboutTheButton() {
    console.log(this);
}
// this function is bound to this button
button1.addEventListener('click', tellMeAboutTheButton);
button2.addEventListener('click', tellMeAboutTheButton);
```

```
tellMeAboutTheButton = () => {
  console.log(this)
}
```

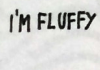
- if used with an arrow function it won't work in the same way because this keyword is always scoped to a function.
- arrow functions don't bind their own scope, but inherit it from the parent one

```
function tellMeAboutTheButton() {
  console.log('outside', this);
  setTimeout( () => {
    console.log('inside', this);
    this.textContent = 'You clicked me';
  }, 1000);
}
```

if you use an arrow function (no scope), this keyword will be equal to whatever this keyword was as at the higher level function

if there is no higher function that is wrapped around this then this keyword will be equal to the window.

every time you create a function the value of this will change, unless you use an arrow function so this will be equal to this in a higher scope




```
function Pizza(toppings = [], customer) {
    console.log('Making a pizza');
    // save the toppings that were passed in
    // to this instance of pizza
    this.toppings = toppings;
    this.customer = customer;
    this.id = Math.floor(Math.random() * 16777215).toString(16);
}
const pepperoniPizza = new Pizza(['pepperoni'], 'John Wick');
const polishPizza = new Pizza(['onions', 'potatoes', 'more onions'], 'Jan Kowalski');
```

random cute id generator

Random Hex Color Code Generator in JavaScript

```
'#'+Math.floor(Math.random()*16777215).toString(16);
//16777215 === ffffff in decimal
```

~ @paul_irish

Lesson #62 prototype refactor of the gallery exercise, the benefits of code refactoring

code refactoring

Code refactoring is a process or logical technique of changing the actual computer program's source code not changing its external functional behaviour so that some non-functional attributes would be upgraded. The most essential benefits of refactoring are enhanced code readability and decreased complexity. It can be make the code design modern and easier to work with. Through this process, it becomes possible to boost the maintainability and extensibility of the source code, also a more eloquently internal architecture or object model. ••• Put in simple terms: refactoring is a series of actions to analyze, define and simplify the design of actual code, without changing its behaviour. It also could be a good tip in understanding an unfamiliar code base.

THE VISION OF OVERALL PICTURE
The approach would be too long and extremely complicated, in case there is one primary method that manages all functionality. However, you divide it into parts, it's easy to figure out what is really being done.

PRODUCTIVITY
Code refactoring could be thought-out as an investment. The effort required for future changes to the code is reduced so that the efficiency is improved.

MAKE IT READABLE
Don't write it for yourself. Make it easy to understand.

REDUCTION OF COMPLEXITY
Make it easier to work on the project.

MAINTAINABILITY
Pay attention to the integration of updates and upgrades. This is a continous inevitable process that should be welcomed. It is difficult for developers to make any changes, in case the codebase is messed up. So with code refactoring will help to develop the product on a clean foundation and will be ready for the future updates.

SOURCE
https://quasarbyte.com

Lesson #63 prototypes and prototypal inheritance

Inheritance and the prototype chain

- JavaScript is a bit confusing for developers experienced in class-based languages (like Java or C++), as it is dynamic and does not provide a class implementation per se (the class keyword is introduced in ES2015, but is syntactical sugar, JavaScript remains prototype-based).
- When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property which holds a link to another object called its prototype. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. By definition, null has no prototype, and acts as the final link in this prototype chain.
- Nearly all objects in JavaScript are instances of Object which sits on the top of a prototype chain.
- While this confusion is often considered to be one of JavaScript's weaknesses, the prototypal inheritance model itself is, in fact, more powerful than the classic model. It is, for example, fairly trivial to build a classic model on top of a prototypal model.

Inheritance with the prototype chain

inheriting

properties

JavaScript objects are dynamic "bags" of properties (referred to as own properties). JavaScript objects have a link to a prototype object. When trying to access a property of an object, the property will not only be sought on the object but on the prototype of the object, the prototype of the prototype, and so on until either a property with a matching name is found or the end of the prototype chain is reached.

methods

JavaScript does not have "methods" in the form that class-based languages define them. In JavaScript, any function can be added to an object in the form of a property. An inherited function acts just as any other property, including property shadowing as shown above (in this case, a form of method overriding). When an inherited function is executed, the value of this points to the inheriting object, not to the prototype object where the function is an own property.

```
>pepperoniPizza
▶ Pizza {toppings: Array(1), customer: "John Wick", id: "9f4920", slices: 5}
   ▶ toppings: ["pepperoni"]
     customer: "John Wick"
     id: "9f4920"
     slices: 5
   ▼ __proto__:
     ▶ eat: f()
     ▶ constructor: f Pizza(toppings=[], customer)
     ▶ __proto__: Object
```

instances

- prototype lookup: first looks for instances. if it doesn't exist it will go to prototype

RANDOM

{[native code]}
implemented in whatever language the browser is written in.

RANDOM

- window.dogSlider = dogSlider
- right-click in console, store as global variable.

Ways of accessing your variables in browser, when using parcel. For debugging purposes.

> technicaly you are able to modify, create the built in prototype methods, for example: String.prototype.myMethod, but you should NOT do it.

Polyfill

A polyfill is a piece of code (usually JavaScript on the WEB) used to provide modern funcionality on older browsers that do not natively support it. Other times, polyfills are used to adress issues where browsers implement the same features in different ways. For example, a poly fill could be used to mimic the funcionality of an HTML Canvas element on Microsoft Internet Explorer 7 using a Silverlight plugin or mimic support for CSS rem units, or text-shadow, or whatever you want

Lesson #64 prototype refactor of the Slider Exercise

```
nextButton.addEventListener('click', move); // refactor move into this.move, but don't let this to get attached to nextButton, but instead to one function scope higher

this.move = this.move.bind(this);
ooo.ooo('click', () => this.move());  ||  ooo.ooo('click', this.move.bind(this));  ||  ooo.ooo('click', this.move);
```

Lesson #65 bind, call and apply. 3 functions that are used to change the scope of what this is equal to, inside of a function or inside of the method.

```
const person = {
  name: 'Spike',
  sayHi() {
    return `hey ${this.name}`;
  }
};
person.sayHi()
"hey Spike"
const sayHi = person.sayHi
sayHi()
"hey"
const sayHi = person.sayHi.bind(person)
sayHi()
"hey Spike"
const jenna = {name: 'Jenna'};
const sayHi = person.sayHi.bind(jenna);
sayHi()
"hey Jenna"
const sayHi = person.sayHi.bind({name: 'Wes'});
sayHi()
"hey Wes"
```

```
<div class="wrapper">
  <p>Hey I'm in a wrapper</p>
</div>
<p>Hello</p>
```

method calling against object

```
const wrapper = document.querySelector('.wrapper');
const p = wrapper.querySelector('p');
console.log(p); // -> <p>Hey I'm in a wrapper</p>
```

✗ `const $ = document.querySelector` — `console.log($('p'));` // returns error: the object that the method was called against was taken away. method not bound to anything

✓ `const $ = document.querySelector.bind(document);` // manually passing the reference to the thing you want this to be equal to.

```
console.log($('p')); // returns "<p>Hello</p>"
// by calling bind against querySelector, we say: when the $ function is run, use 'document' as the 'this' value.
```

```
const bill = {
  total: 1000,
  calculate: function(taxRate) {
    return this.total + (this.total * taxRate);
  }
}
const total = bill.calculate(0.13); // classic approach
```

using bind will change the context of what this is equal to inside of a function or inside of a method.

```
let boundFunc = func.bind(thisArg, arg1, arg2, ...argN)
const calc = bill.calculate.bind({total: 500}, 0.06);
const total2 = calc(); // need to call it manually

func.call([thisArg, arg1, arg2, ... argN])
const total3 = bill.calculate.call({total: 200}, 0.07);

func.apply(thisArg, [argsArray])
const total4 = bill.calculate.apply({total: 300}, [0.08]);
const numbers = [5, 6, 2, 3, 7];
const max1 = Math.max.apply(null, numbers); // 7
const max2 = Math.max(...numbers); // 7
```

bind

The bind() method creates a new function that, when called, has its this keyword set to the provided value, with a given sequence of args preceding any provided when the new function is called.

call

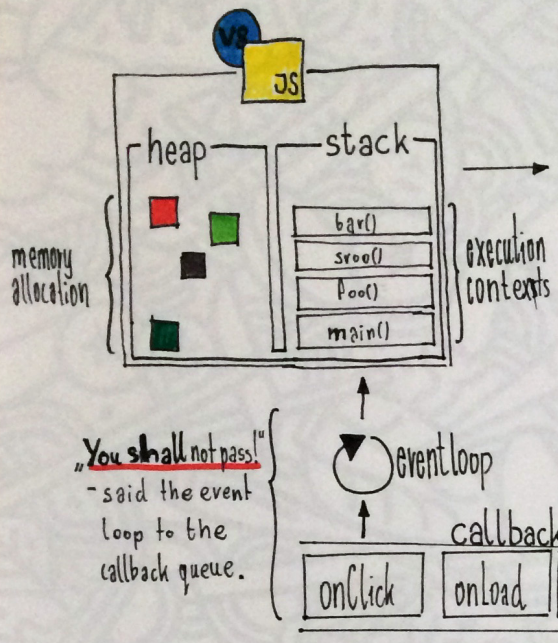
Almost same as bind except, call() method calls a function with a given this value and arguments provided individually

apply

Almost identical to call() functions except apply() accepts a single array of arguments.

{ when the first argument is undefined or null, similar outcome can be obtained using the array spread syntax.

Lesson #66 The Event Loop and Callback Hell. What the heck is the event loop anyway? - Philip Roberts | JSConf EU



Web APIs

- DOM (document)
- ajax (XMLHttpRequest)
- Timers (setTimeout)

Concurrency

- Concurrency means multiple computations are happening at the same time.
- JavaScript has a concurrency model based on an event loop, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. This model is quite different from models in other languages like C and Java.
- JavaScript runtime can do only one thing at the time, except... not really?
- The reason we can do things concurrently is that the browser is more than just the runtime. The browser gives us WebAPIs which are effectively threads that you can make calls to

blocking

- code that is slow and blocks the stack
- examples:
  - image processing
  - network requests
  - looping over big numbers
- solutions, e.g. asynchronous callbacks

WebAPIs

WebApis, in the context of the browser, simply is an API, provided by the browser and that we can communicate with using JavaScript in order to solve our front-end problems.

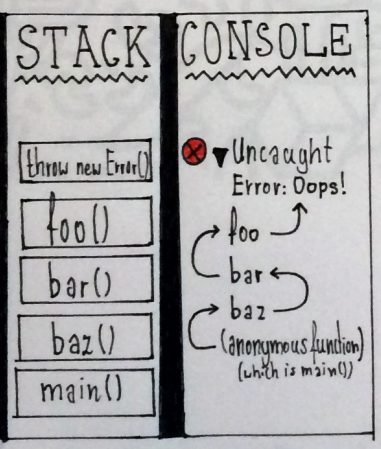
call stack

- data structure which records where in a program we are
- if we step into a function, we push something on to top of stack.
- if we return from a function we pop off the thing off the top of the stack

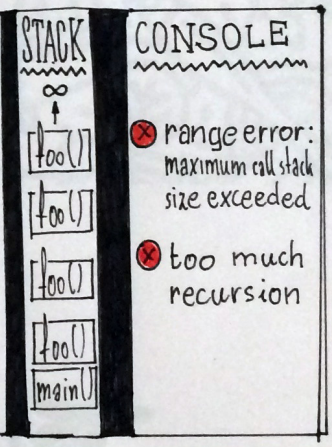
V8 (Javascript engine)

V8 is Google's open source high-performance JavaScript and WebAssembly engine, written in C++. It is used in Chrome and in Node.js, among others.

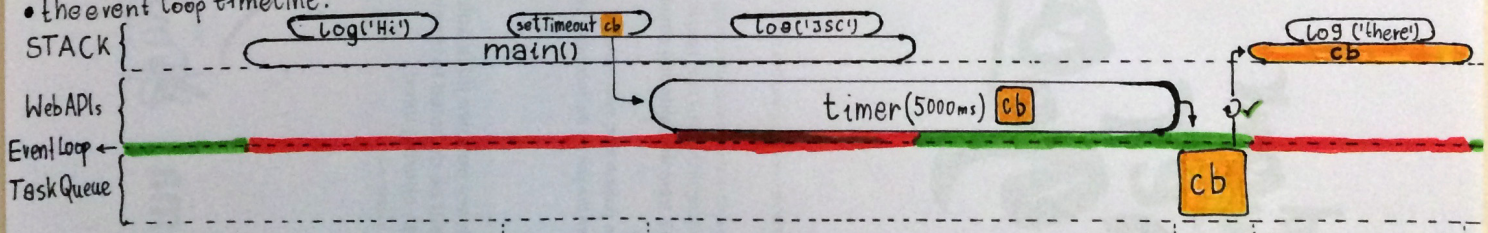
```
function foo() {
  throw new Error('Oops!');
}
function bar() {
  foo();
}
function baz() {
  bar();
}
baz();
```




```
function foo() {
  return foo();
}
foo();
```



- the event loop timeline:




```
console.log('hi');
setTimeout(function() {
  console.log('there');
}, 5000);
console.log('JSConf EU');
```

console:
"Hi"
"JSConf EU"
"there"

- setTimeout(cb) is called.
- setTimeout is a WEB API provided by a browser. It doesn't live in V8 source
- the browser kicks off the timer for you and holds it in WEB API until it is completed.
- once the timer is completed the Web API can't start modifying your code. It can't just chuck stuff onto a stack, cause if it did it would appear randomly in the middle of your code.
- when the timer is ready, Web Api pushes the callback on to the task queue.
- the callback lands in task queue and shortly after is dispatched to the stack (that is clear).

event loop: has 2 jobs:

- look at the stack and at the task queue.
- if the stack is empty (green) it takes the first thing from task queue and pushes it on stack, which runs it.

Lesson #66 continues... repaint, callback hell, javascript engine(s), Brendan Eich

- if the timeout time was set initialy to zero, the timer will be immediately completed in WebAPi and instantly pushed to the task queue. The callback function still has to wait for the green light from the Event Loop to be pushed on to the top of the stack. And the signal (green light) comes when the stack becomes clear.

- async apis
- timeout callbacks get queued
- setTimeout set time value is not a guaranteed time of execution, it's the minimum time of the execution.
- setting time to 0 doesn't run the code immediately, it runs the code nextish some time.

- callbacks can be two things, depending on who you speak to, and how they phrase things:
  1. any function that another function calls.
  2. asynchronus callback as in one that will get pushed back on the callback queue at some point in the future.

Synchronous function

```
[1,2,3,4].forEach(function(i){
    console.log(i);
});
```

- the render queue is blocked through out the runtime of the function as the function iterations ~~take up~~ keep filling the stack until the function is done.

asynchronous function

```
function asyncForEach(array,cb){
    array.forEach(function(){
        setTimeout(cb,0);
    })
asyncForEach([1,2,3,4],function(i){
    console.log(i);
}
```

- the render queue is only slightly blocked in the beginning when all the timeout functions are being send to WebApi, and to task queue. However sending the timeout function takes relatively quick.
- once the functions ~~is iterating from the~~ callbacks are all set and ready to jump to the stack, it allows the render queue to start rerendering in between of the cb being called

repaint, rendering the browser

- browser would like to repaint the screen every 16.6ms (60 frames/second).
- browser is constrained by what you are doing in JS for different reasons.
- browser can't do a render if there is code on the stack.
- render call is almost like a callback itself, it has to wait till stack is clear.
- render has higher priority than callbacks.

Javascript Engine

- A JavaScript engine is a computer program that executes JavaScript (JS) code. The first JavaScript engines were mere interpreters, but all relevant modern engines utilize just-in-time compilation for improved performance.
- JavaScript engines are typically developed by web browser vendors, and every major browser has one. In browser, the javascript engine runs in concert with the rendering engine via the DOM.
- The use of JavaScript engines is not limited to browsers. For example, the Chrome V8 engine is a core component of the popular Node.js runtime system. Since ECMAScript (ES) is the standardized specification of JavaScript, ECMAScript is another name for these engines.
- notable engines
  - Google — V8 — Chrome
  - Mozilla — SpiderMonkey — Firefox
  - Apple — JavaScript Core — Safari
  - Microsoft — Chakra — Internet Explorer
  - Facebook — Hermes — Android Apps

- scrolling issue
  - scroll events in DOM triggers on every frame every 16 milliseconds
  - to debounce it que up these events, but do work every few seconds or until the user stops scrolling for some amount of time

callback hell

if you need to do one thing after another you must nest the callbacks inside of each other, because they all depend on the previous callback to being called before it can then go ahead and run. solution to callback hell is the promise land.

Brendan Eich

- An American technologist and creator of the Javascript programming language. He co-founded the Mozilla project, the Mozilla Foundation and the Mozilla Corporation, and served as the Mozilla Corporation's chief technical officer and briefly, as its chief executive officer. He is the CEO of Brave Software.
- He started work at Netscape Communications Corporation in April 1995. Eich originally joined intending to put Scheme "in the browser", but his Netscape superiors insisted that the language's syntax resemble that of Java. The result was a language that had much of the functionality of Scheme, the object orientation of Self, and the syntax of Java. The first version was completed in 10 days in order to accommodate the Navigator 2.0 Beta release schedule, and was called Mocha, but renamed to LiveScript in September 1995 and later JavaScript in December. Eich continued to oversee the development of SpiderMonkey, the specific implementation of JavaScript in Navigator.

#67 Promises

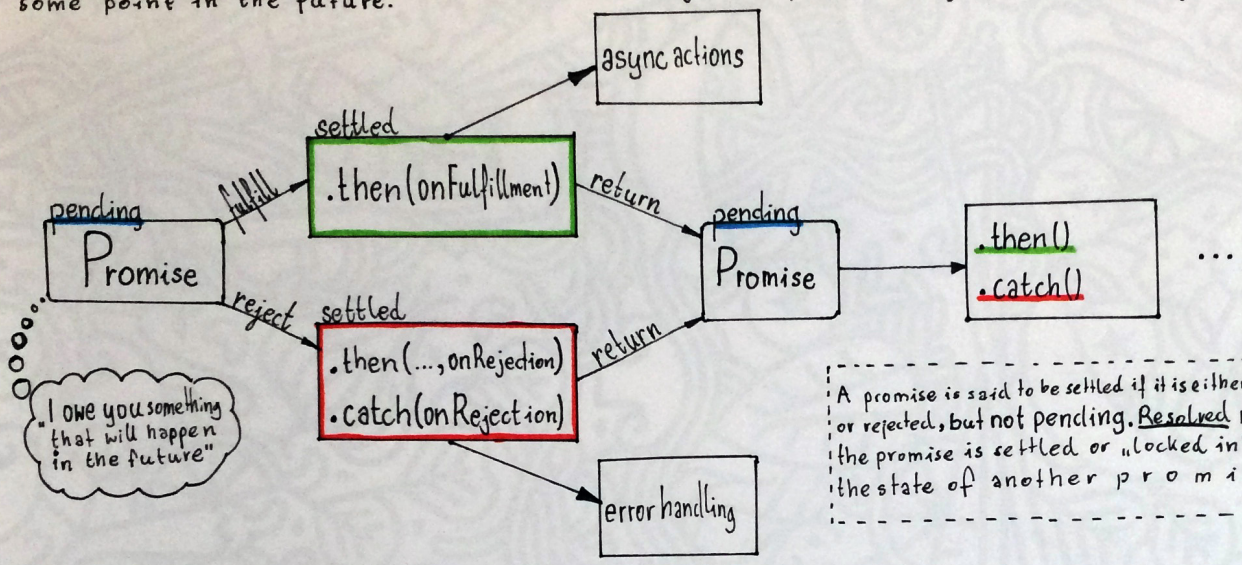
Promise

- The Promise object represents the eventual completion (or failure) of an asynchronous operation, and ~~it~~ its resulting value.
- A promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

A Promise is in one of these states:

- **pending** — initial state, neither fulfilled nor rejected.
- **fulfilled** — meaning that the operation completed successfuly.
- **rejected** — meaning that the operation failed.

A pending promise can either be fulfilled with a value or rejected with a reason (error). When either of these options happens, the associated handlers queued up by a promise's .then method are called. (If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.)



A promise is said to be settled if it is either fulfilled or rejected, but not pending. Resolved means that the promise is settled or "locked in" to match the state of another promise.

Constructor Syntax

const promiseObj = new Promise(executor)

Arguments: executor

A function to be executed by the constructor, during the process of constructing the promiseObj. The executor is custom code that ties an outcome to a promise. You, the programmer, write the executor.

The signature of this function is expected to be

```
function(resolutionFunc, rejectionFunc) {
  // typically, some asynchronous operation.
}
```

```
function makePizza(toppings = []) {
  return new Promise(function(resolve, reject) {
    if (toppings.includes('pineapple')) {
      reject('seriously?! Get out!');
    }
    const amountOfTimeToBake = 500 + toppings.length * 200;
    setTimeout(function() {
      resolve(`Here is yor pizza 🍕 with the toppings: ${toppings.join(' ')}`);
    }, amountOfTimeToBake);
  });
}
```

Greek-Canadian Sam Panopoulus claimed that he created the first Hawaiian (pineapple) pizza at the Satellite Restaurant in Chatham, Ontario, Canada in 1962.

- **sequentially**

```
makePizza(['pepperoni'])
  .then(function(pizza) {
    console.log(pizza);
    return makePizza(['ham', 'cheese']);
  })
  .then(function(pizza) {
    console.log(pizza);
    return makePizza(['chillies', 'onion', 'feta']);
  })
  .then(function(pizza) {
    console.log(pizza);
  });
```

- **concurrently**

```
const pizzaPromise1 = makePizza(['pepperoni']);
const pizzaPromise2 = makePizza(['ham', 'cheese']);
const pizzaPromise3 = makePizza(['chillies', 'onion', 'feta']);

pizzaPromise1.then(pizza => console.log(pizza));
pizzaPromise2.then(pizza => console.log(pizza));
pizzaPromise3.then(pizza => console.log(pizza));

// instead of the above way we can also group all the
// promises into one. Promise.all will take all your
// promises and will only resolve when all of the
// sub promises have been resolved.
const dinnerPromise = Promise.all([pizzaPromise1, pizzaPromise2, pizzaPromise3]);

// RETURNS ARRAY (1)
dinnerPromise.then(pizzas => console.log(pizzas));

// DESTRUCTURING IN FUNCTION'S SCOPE (2)
dinnerPromise.then(function(pizzas) {
  const [pepperoni, cheesyham, spicypizza] = pizzas;
  console.log(pepperoni, cheesyham, spicypizza);
});

// DESTRUCTURING ARGUMENT DIRECTLY (3)
dinnerPromise.then(function([pepperoni, cheesyham, spicypizza]) {
  console.log(pepperoni, cheesyham, spicypizza);
});
```

Promise.race returns the first resolved promise from the group of promises

```
const firstPizzaPromise = Promise.race([pizzaPromise1, pizzaPromise2, pizzaPromise3]);
firstPizzaPromise.then(pizza => {
  console.log('here is your first pizza');
  console.log(pizza);
});
```

Lesson #68 Promises Error Handling

- rejecting – opposite of resolving.
- the way you catch an eror in a promise is you chain .catch at the end.
- .then will only happen when the promise resolves successfuly.
- .catch will only run when the promise doesn't go successfuly.
- error in promise chain will stop the rest of the promise chain. Promise chain technique does not have a solution for that. If you want the promises to carryon after catching an eror in the middle you have to use a different approach[redacted]
- not every single promise needs a .catch at the end. It is sufficient to chain only one .catch at the end of the promises chain.
- ful filled is a another word for resolved.
- Promise.ALL if all the promises were resolved it will return an array of resolves. OR If one of the promises breaks, all promise will break and will return a single reject reason.

vs

- Promise.allSettled always returns an array of objects containing promises and their statuses. It will never reject. It will resolve once all Promises in the array have either rejected or resolved.

```
makePizza(['cheese','pineapple','ham']).then(pizza =>{
    console.log(pizza);
}).catch(err => {
    console.log('Oh no!');
    console.log(err);
});
```

// or handle the error elsewhere .catch(handleError);

```
if(toppings.includes('pineapple')){
    reject('Seriously? Get out!');
}
```

```
const p1 = makePizza(['pepperoni']);
const p2 = makePizza(['pineapple']);
const dinnerPromise = Promise.allSettled([p1, p2]);
dinnerPromise.then(results =>{
    console.log(results);
});
```

```
▼(2)[{...}, {...}]
  ▼0:
    status: "fulfilled"
    value: "Here is your Pizza 🍕 with the toppings pepperoni"
  ▶__proto__: Object
  ▼1:
    status: "rejected"
    reason: "Seriously? Get out!"
  ▶__proto__: Object
  length: 2
▶__proto__: Array(0)
```

Lesson #70 Async await

async function

An async function is a function declared with the async keyword. Async functions are instances of the AsyncFunction constructor, and the await keyword is permitted within them. The async and await key words enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains. Async functions may also be defined as expressions.

```
function wait(ms = 0){
    return new Promise(resolve=>{
        setTimeout(resolve,ms);
    });
};

async function go(){
    console.log('starting');
    await wait(2000);
    console.log('ending');
};
go();
```

- you can only use async await inside of a function that's marked as async.
- async await allows us to put a keyword await in front of a promise based function and it will temporary pause that function from running until that function is resolved.
- you can't do top level await (document). Await is only valid in async functions. However, top-level await works in dev tools.
- when you ~~make~~ mark a function as async it will immediately return a promise to you. Async functions themselves are promises. Therefore we can use .then & .catch on them if we want to.
- you can mark with async any type of function:

```
// function declaration
async function fd(){};
// arrow function
const arrowFunction = async ()=>{};
// callback function
window.addEventListener('click', async()=>{});
// in object
const person = {
    // method function
    async sayHi(){},
    // method shorthand
    async sayHello(){},
    // function property
    sayHey: async()=>{},
};
// immediately invoked function expression IIFE
(async function(){})();
```

```
async function makeDinner(){
    const pizza1 = makePizza(['pepperoni']);
    console.log(pizza1)
}
makeDinner(); //returns a promise
```

if you want an actual data that comes back from the promise you must await it.

↓

```
const pizza1 = await makePizza(['pepperoni']);
console.log(pizza1); //will log the result
```

- synchronously waiting for the pizza to be done. When it's done it will console log it.
- why is await an await and not wait? Because it's asynchronously waiting. Meaning that it's not going to pause all of JS from running in different places on the page.

```
async function makeDinner(){
    const pizzaPromise1 = makePizza(['pepperoni']);
    const pizzaPromise2 = makePizza(['mushrooms']);
    const [pep, mush] = await Promise.all([pizzaPromise1, pizzaPromise2]);
    console.log(pep, mush);
}makeDinner();
```

- pizza Promise 1 & pizza Promise 2 are running concurrently. await Promise.all will return both pizzas (data) when they are done. Without "await" it would return a promise.

Lesson #71 async await error handling

1 try & catch

- you can make multiple promises in here
- you have to deal with a lot of brackets

```
async function go(){
  try{
    const pizza = await makePizza(['pineapple']);   // same zone: if anything breaks it will land safely in catch
    console.log(pizza);
  } catch(err){
      console.log("Oh no!");   // this is where you want display UI to user that something went wrong or send of data to error tracking service
      console.log(err);
  }
}
go();
```

2 mix & match async await + use of promise syntax for the error handling

- mix & match is the best of both ~~word~~ worlds:
  - using await to get the data of the promise instead of .then
  - still using older syntax .catch on the end of the function.
- helpful for when you want to handle the error at the time of defining the function

```
function handleError(err){
    console.log('ohhhhhh nooo ayayay terrible news');
    console.log(err);
}
```

2A

- when you want to handle the error at the time that you define a function
- useful if you need to do something with error inside of that function. for example display a special modal box at the time of definition rather than at the time of call.

```
async function go(){
  const pizza = await makePizza(['pineapple']).catch(handleError);
  console.log(pizza);
}
go();
```

2B

- when you want to check error when calling a function you you attach .catch to the fun when you are calling it.

```
async function go(){
  const pizza = await makePizza(['pineapple']);
  console.log(pizza);
  return pizza;
}
go().catch(handleError);
// every function that is marked with async
// will automatically return a promise which
// allows you to chain that function
// with .catch() or with .then().
```

3

- you can nest promises as deep as you want.
- it's pretty common to have a good number of your functions marked as async. Promises happening inside of promises

```
async function go(){
  const pizza = await makePizza(['pineapple']);
}
async function goGo(){
  const result = await go();
}
goGo().catch(handleError);
```

4 higher order function

- function that returns other function
- this approach is common in nodejs and express

1: Go ahead and define all of your functions just as if you were never to have any errors. Remember to write async functions but don't worry about error handling inside of those functions.

2: When the time for calling that function comes you have two options. You can attach .catch to function at runtime: (2B) Or you can make a safe function with a higher order function (HOF):

```
function makeSafe(fn, errorHandler){
    return function(){
      fn().catch(errorHandler)
    }
}
const safeGo = makeSafe(go, handleError);
safeGo();
```

Lesson #72 async await prompt UI

- prompt interface:
  - click on the page
  - pop up the box
  - ask for something
  - get that data back
  - display it on the page
- prompt('how old are you?!'):
  - a built in browser prompt
  - limited to only one input box
  - blocking interface: can't do anything on the page
  - can't add any image etc
  - re-implement this with promises and async await
- <fieldset>: It's like div. It groups together form inputs. It has some handy stuff like "disable" parameter which can disable all inputs inside of it. Much more semantic element to use than just a div.

CSS

```
.popup {
  ...
  ...
  pointer-events: none;
  --opacity: 0;
  opacity: var(--opacity);
}
.popup.open {
  --opacity: 1;
  pointer-events: all;
}
```

JS

```
document.body.appendChild(popup);
// put a very small timeout before we add the open class.
setTimeout(function(){
  popup.classList.add('open');
}, 10);
// it will give space for transition
// "0" could also work but depends on the browser
// could also use await wait(10); instead
```

- how to check if a property on an object exists, regardless of which value it is set to:

```
'cancel' in button.dataset
// checking for data-cancel property

const wes = { name: 'wes' }
'name' in wes // true
'age' in wes // false
button.hasAttribute('data-cancel');
```

```
const questions = [{title: 'What is your name?'}, {title: 'What is your age?', cancel: true},
                   {title: 'What is your dogs name?'}, ];
```

- Promise.all will fire them all concurrently (at the same time):

```
Promise.all([ask(questions[0]), ask(questions[1]), ask(questions[2])]).then(answers => {
    console.log(answers);
});
```

- Map approach: It will loop over each of the questions and pipe it into an ask function, which returns a promise that will return to us an array. Wraps it in a promise.all which allows us to listen with .then. Yet, it still runs concurrently.

```
Promise.all(questions.map(ask)).then(data => {
    console.log(data);
});
```

- Unlike forEach and map for of allows you to pause a function and wait for something, which means that you will be able to work asynchronously:

```
async function askMany(){
  for(const question of questions){
    const answer = await ask(question);
    console.log(answer);
  }
}
askMany();
```

- to achieve exactly above but with using utility function *asyncMap* that returns an array we need to do this

```
async function asyncMap(array, callback){
  // make an array to store our results
  const results = [];
  // loop over our array
  for(const item of array){
    const result = await callback(item);
    results.push(result);
  }
  // when we are done with loop, return results
  return results;
}
```

```
async function go(){
  const answers = await asyncMap(questions, ask);
  console.log(answers);
}
go();
// top level function instead
// of the method that lives on array
```

- removing an element from the DOM element.remove() does not remove it from JavaScript memory entirely. But: popup.remove(); + popup = null, does.

- remember about {once: true} or {capture: true} in options of EventListener. that these things exist :)

Lesson #73 async typer UI

```
function getRandomBetween(min=20, max=150) {
    return Math.floor(Math.random()*(max-min)+min);
}
```

✗

```
function getRandomBetween(min=20, max=150, randomNumber=Math.random()){
    return Math.floor(randomNumber*(max-min)+min);
}
```

✓

Pure function

- Pure function is a function that when given the same input (arguments) will always return the same output.
- With random numbers and dates you never know what the values in the output are going to be.

```
<p data-type-min="5" data-type-max="60">Too tired to type</p>
const el = document.querySelector('p');
>>> el.dataset
    ▶ DOMStringMap(2) {typeMin: "5", typeMax: "60"}
>>> const {typeMin, typeMax} = el.dataset;
```

HTML	kebab	data-type-min
JS	camel	dataset.typeMin

- async for of loop

```
async function draw(el) {
  const text = el.textContent;
  let soFar = '';
  for (const letter of text){
    soFar += letter;
    el.textContent = soFar;
    //wait for some amount of time
    const {typeMin, typeMax} = el.dataset;
    const amountOfTimeToWait = getRandomBetween(typeMin, typeMax);
    await wait(amountOfTimeToWait);
  }
}
```

```
function wait(ms=0){
  return new Promise(resolve=>{
    setTimeout(resolve, ms)
  });
}
```

- recursion

```
function draw(el) {
  let index = 1;
  const text = el.textContent;
  const {typeMin, typeMax} = el.dataset;
  async function drawLetter(){
    el.textContent = text.slice(0, index);
    index += 1;
    const amountOfTimeToWait = getRandomBetween(typeMin, typeMax);
    //exit condition
    await wait(amountOfTimeToWait);
    if (index <= text.length) {
      drawLetter();
    }
  }
  // when this function draw() runs, kick off drawLetter
  drawLetter();
}
```

```
document.querySelectorAll('[data-type]').forEach(draw);
```

Lesson #74 ajax and APIs

Ajax

- **A**synchronous **J**ava Script **a**nd **X**ML, while not a technology itself, is a term coined in 2005 by Jesse James Garret, that describes a "new" approach to using a number of existing technologies together, including: HTML or XHTML, CSS, JavaScript, DOM, XML, XSLT, and most importantly the XMLHttpRequest object.
- When these technologies are combined in the Ajax model, web applications are able to make quick, incremental updates to the user interface without reloading the entire browser page. This makes the application faster and more responsive to user actions.
- Although **X** in Ajax stands for **XML**, **JSON** is used more than XML nowadays because of its many advantages such as being lighter and a part of JavaScript. Both JSON and XML are used for packaging information in the Ajax model.

- most popular way to pull data from an existing website, when the API is offerred is through JSON
- with JSON viewer extension you can make JSON to look nicer in browser
- JSON.parse(data) converts string into an object
- lots of API will allow you to pull the data from their server into your own (open API). However there are many that do not allow it and require API keys.
- Nowadays XML is not teinpased. However you may still find it in outdated projects.
- github.com/public-apis/public-apis - a categorised list of many public open APIs
- fetch library - returns a promise. there are number of methods that we can use on the response.
- every API might be implemented differently, offering diffent access. Always read the docs.

```
async function displayData(query){
  dataEL.textContent = 'Loading...'; //we can enable a spinner here
  const response = await fetch(`${endpoint}?${query}`);
  const data = await response.json();
  dataEL.textContent = `${data.title} - ${data.description}`;
}

function handleError(err){
  console.log('OH NO!');
  console.log(err);
  dataEL.textContent = 'An error has occurred!';
}

displayData(myQuery).catch(handleError);
```

DOUBLE PROMISE

- the first one gets a response
- the second one turns response into JSON

MIX & MATCH error handling

- checking for error when calling a fetch function by attaching .catch to the call of the function

LOAD & DISPLAY

- manipulate the element beforehand to let know the user that the content is loading. for ex by adding a spinner
- display the content once the data is succesfully obtained

Lesson #75 CORS and recipes

CORS

- Cross-Origin Resource Sharing is a mechanism that uses additional HTTP headers to tell browsers to give a web application running at one origin, access to selected resources from a different origin. A web application executes a cross-origin HTTP request when it requests a resource that has a different origin (domain, protocol, or port) from its own.

origin - Web content's origin is defined by the scheme (protocol), host (domain), and port of the URL used to access it. Two objects have the same origin only when the scheme, host and port all match. Some operations are restricted to same-origin content, and this restriction can be lifted using CORS

- An example of a cross-origin request: the front-end JavaScript code served from:
  - https://domain-a.com uses XMLHttpRequest to make a request for:
  - https://domain-b.com/data.json.
- For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts. For example, XMLHttpRequests and the Fetch API follow the same-origin policy. This means that a web application using those APIs can only request resources from the same origin the application was loaded from, unless the response from other origins includes the right CORS headers.

```
const endpointBase = 'http://www.recipepuppy.com/api';
const proxy = 'https://cors-anywhere.herokuapp.com/';
async function fetchRecipes(query){
  const res=await fetch(`${proxy}${endpointBase}?q=${query}`);
  const data = await res.json();
  console.log(data);
}
fetchRecipes('pizza');
```

In other words, easier words...

- By default websites can't talk to each other, from one domain name to another domain name.
- CORS changes that 'default' and allows the communication, however, it has some rules not making it simple
- In order to communicate with other domain using CORS you must provide Origin header which works kind of like a key.
- Source domain will accept only chosen headers from some specific domains, which means, we probably won't be included on this special invitation list, therefore we will be rejected.
- There is a way to spoof a header and eventually get the access
- You can't spoof a header from the browser because the browser doesn't allow JavaScript to override the Origin header.
- However you are able to override the Origin Header from the server side.
- You can simply achieve it by using CORS proxy which will work like a safe passage between you and for example API from another domain.
- popular CORS proxy: cors-anywhere.herokuapp.com

lesson #76 Dad Jokes

- an example of different format api provider: icanhazdadjoke.com
- API response format
- All API endpoints follow their respective browser URL, but we adjust the response formatting to be more suited for an API based on the provided HTTP `Accept` header.

Accepted `Accept` headers:

- `text/html` HTML response, default response format
- `application/json` JSON response
- `text/plain` plain text response

Note: Requests made via `curl` which do not set an `Accept` header will respond with `text/plain` by default

CHROME ⊗ Uncaught (in promise) SyntaxError: Unexpected token < in JSON at position 0

FIREFOX ⊗ SyntaxError: JSON.parse: unexpected character at line 1 column 1 of the JSON data

- Most likely you try to parse something that is not JSON. "<" suggests that it could be, and most probably is a HTML
- We need to pass an accept header which is either text/html, or JSON or plain text

↓

- We pass a 2nd object to fetch, I mean, 2nd argument that is an object

↓

```
const response = await fetch(`
  ${endpoint}`,
  headers:{
    Accept:'application/json',
  },
);
```

```
•a random utility function
•pun intended
function randomItemFromArray(arr, not){
  const item=arr[Math.floor(Math.random()*arr.length)];
  if (item === not) {
    console.log('Ahh we used that one last time, look again');
    return randomItemFromArray(arr, not);
  }
  return item;
}
//randomly chosing an item from (arr) Array.
//the chosen item must be different
//to the one (not) passed as argument
```

Lesson #77 Currency converter

- select all the elements on the page using querySelector
- define the endpoint (`https://api.exchangerates.api.io/latest`)
- define an object ratesByBase for storing fetched data
- define an object in which keys will store ISO currency code ("USD"), and values the full name of the currency {PLN: 'PolishZloty',}
- define a function that will generate options in both of the select sections:

```
function generateOptions(options){ //passing the options from the above object later on
  return Object.entries(options)
    .map( ([currencyCode, currencyName]) => //cool destructuring
      `<option value="${currencyCode}">${currencyCode}-${currencyName}</option>`)
    .join('');
}
```

- populate the options elements using the function from above

```
const optionsHTML = generateOptions(currencies);
fromSelect.innerHTML = optionsHTML; //options were generated only
toSelect.innerHTML = optionsHTML; //once and used for 2 elements
```

- define a function for fetching rates. Instead of fetching all the currencies we define function in a way that it will fetch only the chosen currency

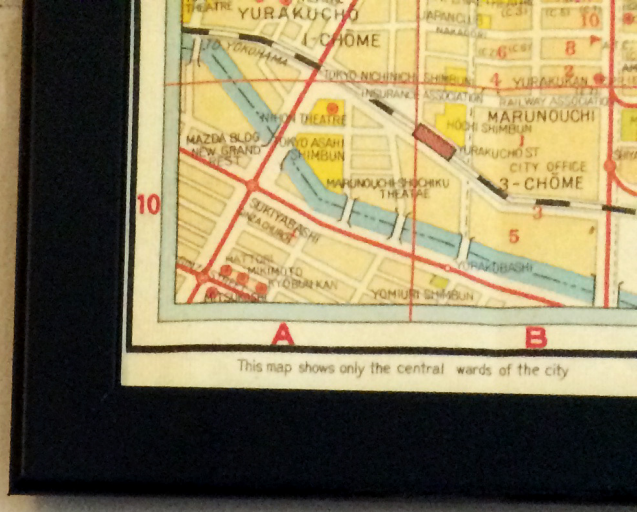
```
async function fetchRates(base = 'USD') {
  const res = await fetch(`${endpoint}?base=${base}`);
  const rates = await res.json();
  return rates;
}
```

- define our brain of the operation, the convert function to which we will pass the values from the form later on.

```
async function convert(amount, from, to) {
  //first check if we already have the rates for that currency already
  //fetched. If not let's fetch it and add it to our ratesByBase object
  if(!ratesByBase[from]) {
    const rates = await fetchRates(from);
    ratesByBase[from] = rates;
  }
  //convert that amount, we have everything we need
  const rate = ratesByBase[from].rates[to];
  const convertedAmount = rate * amount;
  console.log(`${amount} ${from} is ${convertedAmount} in ${to}`);
  return convertedAmount;
```

- add an event listener to the form, listen for any input change (input amount, or select option), and upon the change run convert function with updated values and ultimatelly display the converted amount to the user.

```
async function handleInput(e){
  const rawAmount = await convert(
    fromInput.value,
    fromSelect.value,
    toSelect.value
  );
  toEl.textContent = formatCurrency(rawAmount, toSelect.value);
}

form.addEventListener('input', handleInput);
//Listening on form will listen on any change of input or option select on that form
```



Intl.NumberFormat

v3.dev

The Intl.NumberFormat object is a constructor for objects that enable language sensitive number formatting.

```
const number = 123456.789;
new Intl.NumberFormat('de-DE', {style:'currency', currency:'EUR'}).format(number);
// '123.456,79€'
```

- de-language code
- DE-country code
- style, options:
  - decimal
  - currency
  - percent
  - unit
  - notation
  - signDisplay

- In its most basic form Intl.NumberFormat lets you create a reusable formatter instance that supports locale-aware number formatting. Just like other Intl.*Format APIs, a formatter instance supports both format and formatToParts method.
- In addition to Numbers, Intl.NumberFormat can now also format BigInts.
- Units of measurement, Intl.NumberFormat currently supports the following so-called simple units:
  - angle: degree
  - area: acre, hectare
  - digital: bit, byte, kilobit, kilobyte, megabit, megabyte, gigabit, gigabyte, terabit, terabyte, petabyte
  - duration: millisecond, second, minute, hour, day, week, month, year
  - length: millimeter, centimeter, meter, kilometer, inch, foot, yard, mile, mile-scandinavian
  - mass: gram, kilogram, ounce, pound, stone
  - temperature: celsius, fahrenheit
  - volume: liter, milliliter, gallon, fluid-ounce, liter-scandinavian
  - concentration: percent

```
const formatter = new Intl.NumberFormat('en', {style:'unit', unit:'kilobyte'}).format(1.234); // '1.234 kB'
```

you can combine units: {style:'unit', unit:'meter-per-second'} // m/s

- Compact, scientific, and engineering notation • Compact notation uses locale-specific symbols to represent large numbers. It is a more human friendly alternative to scientific notation. {notation:'standard'} is default. {notation:'compact'} for compact notation.

```
// 1234 → 1.2K   123456 → 123K   123456789 → 123M
```

- {notation:'scientific'} // '2.998E8 m/s'
- {notation:'engineering'} // '299.792E6 m/s'

- {signDisplay:'always'} explicitly display a sign, even when number is positive
- {signDisplay:'exceptZero'} prevent showing the sign when the value is 0.

```
function formatCurrency(amount, currency){
  return Intl.NumberFormat('en-US', {
    style:'currency',
    currency,
  }).format(amount);
}
```

modules

- being able to share a code across multiple javascript files, and across multiple projects
- modules are way to structure and organise your JS
- modules have their own scope. Variables in modules are scoped to the module and they can only be used inside of its module file.
- modules are often called ECMAScript modules, or simply ESG modules.
- you can't use modules unless you're running your code on a server, which enables CORS.
- if your module does only one thing, use a default export. Otherwise use multiple names export.
- <script src="./scripts.js" type="module"></script>

import

- The static import statement is used to import read only live bindings which are exported by another module. Imported modules are in strict mode wether you declare them as such or not. The import statement cannot be used in embedded scripts unless such script has a type="module". Bindings imported are called live bindings because they are updated by the module that exported the binding.
- There is also a function-like dynamic import(), which does not require scripts of type="module"
- Backward compatibility can be ensured using attribute nomodule on the script tag.

export

- The export statement is used when creating JS modules to export live bindings to functions, objects, or primitive values from the module so they can be used by other programs with the import statement. Bindings that are exported can still be modified locally; when imported, although, they can only be read by the importing module the value updates whenever it is updated by the exporting module.
- Exported modules are in strict mode whether you declare them as such or not. The export statement cannot be used in embedded scripts.

import syntax

```
import defaultExport from 'module-name';
import * as name from 'module-name';
import { export1 } from 'module-name';
import { export1 as alias1 } from 'module-name';
import { export1, export2 } from 'module-name';
import {foo,bar} from "module-name/path/to/specific/un-exported/file";
import {export1, export2 as alias2,[...]} from 'module-name';
import defaultExport,{export1[,[...]]} from 'module-name';
import 'module-name';
var promise = import('module-name');
```

- **defaultExport:**
  Name that will refer to the default export from the module.
- **module-name:**
  The module to import from. This is often a relative or absolute path name to the .js file containing the module. Certain bundlers may permit or require the use of the extension; check your environment. Only single quoted and double quoted Strings are allowed.
- **name:**
  Name of the module object that will be used as a kind of namespace when referring to the imports.
- **exportN:**
  Name of the exports to be imported.
- **aliasN:**
  Names that will refer to the named imports.

• • • example of on demand import:

currencies.js
```
currencies = {[,[...]]}
export const localCurrency = 'CAD'; // name export
export default currencies;          // default export (one per file only!)
```

script.js
```
async function handleButtonClick() {
  const {localCurrency, default:currency} = await import('./currencies.js');
  console.log(localCurrency, currency);
}
// loads currencies.js only when the button is clicked
```

// default is a reserved word we have to rename it

// destructuring from the module object

export syntax

- There are two types of exports:
  1. Named Exports (zero or more exports per module)
  2. Default Exports (one per module)

- Exporting individual features
```
export let name1, name2, ..., nameN; // also var, const
export let name1 = ..., name2 = ..., ..., nameN; // also var, const
export function functionName() {...};
export class ClassName {...};
```
- Export list
```
export { name1, name2, ..., nameN };
```
- Renaming exports
```
export { variable1 as name1, variable2 as name2, ..., nameN };
```
- Exporting destructured assigments with renaming
```
export const { name1, name2: bar } = o;
```
- Default exports - ~~variable or function has to be defined before~~
```
export default expression;
export default function (...) {...} // also class, function*
export default function name1(...) {...} // also class, function*
export { name1 as default, ... };
```
- Aggregating modules
```
export * from ...; // does not set the default export
export { name1, name2, ..., nameN } from ...;
export { import1 as name1, import2 as name2, ..., nameN } from ...;
export { default } from ...;
```
- **nameN:**
  Identifier to be exported (so that it can be imported via import in another script).

• • • example

dog.js
```
const dog = {
  name: 'Doggo';
  age: 3;
};

export function woof() {
  console.log('Woof!');
}
export default dog;
```

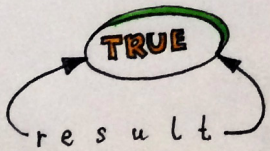
script.js
```
import blabla, {woof} from './dog.js';
// we can name default export
// anything we want

console.log(blabla);  //=> Object {name:'Doggo', age:3}
woof();               // 'Woof!'

import blablba, {woof as bark} from './dog.js';
console.log(blablba); //=> Object {name:'Doggo', age:3}
bark();               // 'Woof!'
```

Lesson #82 using open source npm packages

npm

- npm is the world's largest software regigistry. Open source developers from every continent use npm to share and borrow packages, and many organizations use npm to manage private development as well.
- npm consists of three distinct components:
  - the website
  - the Command Line Interface (CLI)
  - the registry
- Use the website (https://npmjs.com) to discover packages, set up profiles, and manage other aspects of your npm experience. For example, you can set up Orgs (organizations) to manage access to public or private packages.
- The CLI runs from a terminal, and is how most developers interact with npm.
- The registry is a large public database of JavaScript software and the meta-information surrounding it.
- Use npm to...
  - Adapt packages of code for your apps, or incorporate packages as they are.
  - Download standalone tools you can use right away
  - Run packages without downloading using npx.
  - Share code with any npm user, anywhere.
  - Restrict code to specific developers.
  - Create Orgs (organizations) to coordinate package maintenance, coding, and developers.
  - Form virtual teams by using Orgs.
  - Manage multiple versions of code and code dependencies.
  - Update applications easily when underlying code is updated
  - Discover multiple ways to solve the same puzzle.
  - Find other developers who are working on similar problems and projects.

1. ~$ npm i is-thirteen
2. import is from 'is-thirteen'
3. console.log(is(13).thirteen());
4. ~$ npm start
5. open the browser's console



- starting a new project with lots of modules at hand
  1. Create a new folder and enter ther through cmd line.
  2. ~$ npm init
  3. ~$ npm i parcel-bundler -D
  4. ~$ npm i faker date-fns await-to-js lodash axios waait
  5. Create index.html
  6. Create index.js
  7. Add to package.json "start":"parcel index.html" under "scripts" section
  8. Add to package.json "browserslist":["last chrome versions"]
  9. Import your modules. e.g: "import wait from 'waait';" "import {name} from 'faker';"
  10. Use your newly added modules according to their docs.
  11. ~$ npm start

- faker.js

Generates massive amounts of fake data in the browser and in node.js

- import faker from 'faker'; ECMAScript modules
- var faker = require('faker'); outdated way, older node.js syntax
- const randomName = faker.name.findName(); // Kiarra O'Hara
- destructure name on import → import {name} from 'faker';
  - console.log(name.firstName());
  - array of name + surname:
    - const fakeFullNames = Array.from({length:10}, () => `${name.firstName} ${name.lastName}`);

- date-fns

provides collection of functions that allows you to work with date values

- import {formatDistance, format} from 'date-fns';
  - formatDistance(
    new Date(1930,3,4,11,32,0),
    new Date(1930,3,4,10,32,0),
    {addSuffix:true}
    ); // "in about 1 hour"
  - format(new Date(), `LLLL 'the' do y`);
    // "May the 25th 2020"

1.js — compare any 2 dates, will result with "in" or "ago"

- import {formatRelative, subDays} from 'date-fns'
  import {es} from 'date-fns/locale';
  const date = new Date();
  const whenInSpanish = formatRelative(subDays(date,3), date, {locale:es})
  console.log(whenInSpanish);
  // "el viernes pasado a las 19:34"

2.js

- import {format} from 'date-fns';
  format(new Date(), `'Today is a' iiii`);
  // "Today is a Monday"

3.js

Lesson #82 continues, npm packages

• axios

- Promise based HTTP client for the browser and node.js
- Make XMLHttpRequests from the browser.
- Make http requests from node.js
- Supports the Promise API.
- Intercept request and response
- Transform request and response data
- Cancel requests
- Automatic transforms for JSON data
- Client side support for protecting against XSRF

index.js

```
import axios from 'axios';
async function getJoke(){
  const res=await axios.get('https://icanhazdadjoke.com/', {
      headers: { Accept: 'application/json' }
  });
  return res.data.joke;
}
```

• await-to-js

Async await wrapper for easy error handling

- import to from 'await-to-js';

OUR PROMISE

```
function checkIfNameIsCool(firstName){
    return new Promise( function (resolve,reject){
        if (firstName === 'wes'){
            resolve('cool name');
            return;
        }
        reject(new Error('Bad name!'));
    });
};
```

regular approach

```
async function checkName(){
  const nameDesc=await checkIfNameIsCool('snickers');
  console.log(nameDesc);
}
checkName();
```

await-to-js approach

```
async function checkName(){
  const [err,successValue]=await to(checkIfNameIsCool('snickers'));
  if (err){
    console.log(err);
  } else {
    console.log(successValue);
  }
}
checkName();
```

Lesson #83 security

- Javascript is public
  - anything that you put in your source code is not safe.
  - don't put any sensitive information in your client side JavaScript
- API keys, prices
  - if you calculate a price on client side remember to recalculate it on server side and compare.
- XSS, cross site scripting
  - sanitise your inputs
  - any time you take a data from a user and embed it in HTML (innerHTML, insertAdjacent, etc) you must sanitise it. [textContent is safe].
- ways of performing XSS:

```
<img src="..." alt="..." onload="alert('hacked')">
<img src="someInvalidSrc.com" onerror="alert('pwned')">
```

- use only APIs that are encrypted

https://dogsapi.com/

https is a secure extension of http. The communication protocol is encrypted using Transport Layer Security, or, formerly, Secure Sockets Layer (SSL).

• dompurify

- DOMPurify is a DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, MathML and SVG.
- import {sanitize} from 'dompurify'

```
(...)
input.addEventListener('input', () => {
  const clean=sanitize(input.value, {
      FORBID_ATTR:['width','height','style'],
      FORBID_TAGS :['style'],
  });
  output.innerHTML=clean.replace(/\n/g,'<br>');
});
```

- Website security threats:
  - Cross-Site Scripting (XSS)
  - SQL injection
  - Cross-Site Request Forgery (CSRF)
  - Clickjacking
  - Denial of Service (DoS)
  - Directory Traversal
  - File Inclusion
  - Command Injection