JavaScript

- Oggetti built-in
- Eccezioni
- Closure
  - Paradigma IIFE (Indipendently Invoked Functional Expression)
- This
  - Bind
  - Apply e Call
- Ereditarietà
- Manipolazione del DOM (Document Object Model)
  - Selettori Javascript
  - Eventi
    - Associare un evento a un elemento
  - Manipolazione Nodo
  - Creazione Nodo
- Javascript Asincrono
  - Event Loop
  - Promises
    - Ciclo di vita
    - Creazione di una promise
    - Consumazione di una promise
  - Microtask
- AJAX
  - XMLHttpRequest
  - AJAX con Fetch
- ASYNC AWAIT
- CORS
  - Simple Request
  - Preflight Request
- Programmazione Server
  - NodeJS
    - Moduli

JAVASCRIPT

- Sito Guida Javascript
- Javascript gestisce il comportamento della pagina web, la parte dinamica:
  - Cambiare elementi e stili in una pagina (DOM manipulation)
  - Instant Search di Google
  - Chat
  - AJAX (Asynchronous Javascript and XML)
    - Server richiede pagina HTML, Javascript invia dati chiamate alle routine AJAX che aggiorna la pagina HTML per il server e aggiorna l'XML
    - Senza fare un *full page refresh*
  - Test capacità del browser e adattamento (polyfill)
    - Librerie JS che implementano funzioni non implementate in versioni precedenti, per questioni di retrocompatibilità
- **NON**:
  - Accede ai file locali del computer
  - Interagisce con un qualunque server remoto
- **Client side**: Javascipt è praticamente monopolista
  - Tanti framework
  - Sito funny javascript vanilla
- **Server side**: NodeJs
- Ogni Browser possiede una Javascript Virtual Machine
- **Transpilers**: Traduce linguaggi differenti in una specifica versione di JS (e.g. BabelJS)
  - CoffeeScript, TypeScript, Dart, Kotlin...

Caratteristiche del linguaggio

- Dinamico
  - Non compilato e gira in una macchina virtuale
- Loosely Typed
  - Non è tipizzato staticamente/non bisogna dire che tipo ha una variabile
- Case Sensitive
- Garbage Collector
- Shell interattiva nel browser (client side) oppure nella console di node (server side)
- Aggiungere a pagine web
  - file `.js`
  - `<script> ... </script>` (Embeded)

- `<script src="my_script.js"></script>` nell'head (External)
- `// Commento`
- `;` opzionale grazie a Automatic Semicolon Insertion (ASI)

Variabili

- Java ha 7 tipi primitivi: string, number, bigint, boolean, symbol, null, undefined
  - primitivi perché contengono un solo valore
  - hanno metodi come gli oggetti, infatti vengono creati dagli object wrapper
  - (usare **Tab** per vederli tutti quanti in console)
- `let` o `var`
  - `let` si usa più spesso
- `let nomeVariabile = "variabile"`
  - Variabile come intero, stringa, oppure non inizializzata
- Ogni dato appartiene a un tipo ma non serve specificarlo nella dichiarazione
  - `typeof(variabile)` per vedere il tipo
  - numero
    - esiste il `++` e il `--`
  - stringa
    - il `+` concatena
  - booleano (`true` o `false`)
  - `undefined`, variabili non inizializzate
  - `null`
- `const`: costante
- Conversione automatica tra stringhe e interi con concatenazioni o operazioni
  - `"La risposta è giusta è " + 42`, il 42 viene convertito in stringa e concatenato
  - `"42" - 3`, fa 39
  - `"42" + 3`, fa "423"... huh?

Metodi delle variabili

- A seconda del tipo una variabile ha dei metodi
- e.g.
  - `.toUpperCase()`: tutto in maiuscolo
  - `.toFixed()`: taglia il numero con la virgola arrotondando (solo la parte intera)
    - è possibile usare parametri per decidere il numero di numeri dopo la virgola

- Funzioni utili
  - `parseInt(string, radix)`: parsa una stringa e ritorna un intero in una determinata base, anche `parseFloat(string)`
  - `.toString()`
  - `Number("Numero")`
  - `String(num)`

Strict Mode

- `"use strict";` all'inizio del file
- non supportata da tutti i browser
- non è possibile dichiarare una variabile senza `let` o `var`
- `NaN = true;` non è possibile scrivere cose del genere

Interazioni Base

- Alcune funzioni utili all'interno del browser
- Le seguenti sono metodi dell'oggetto `window`

Alert

- Messaggio popup

Prompt

- Messaggio popup con input che verrà assegnato a una variabile

Confirm

- Messaggio popup che si può accettare o no

Operatori di Comparazione

- I soliti: `==, !=, <, <=, >, >=`
- nuovi: `===, !==`
  - identico e non identico (stesso dato e tipo)
  - 5 == "5" VERO
  - 5 === "5" FALSO

Operatori Logici

- `&&`, `||`, `!`
- nuovo: `??`, nulling coaleshing operator,
  - `a ?? b`: se `a` è null o undefined ritorna `b`, altrimenti `a`

IF Ternario

- `(condizione) ? if : else;`
- le parentesi negli if sono opzionali

Variabili truthy e falsy

- ""*Tendenti*" al vero o falso
- Truthy: stringe "piene"
- Falsy: stringhe vuote, 0, NaN, null, undefined
- oggetti vuoti dipende (liste, array)
- `true` è 1

Switch Case

- Solita sintassi `switch (expression) { case label_1: ... [break;] case ...: default: ... }`
- `[break;]`

Istruzioni Base

Funzioni

- `function nomeFunzione([parametri]){ ... }`: Dichiarazione
- `nomeFunzione([parametri]);`: Invocazione
- può ritornare un valore
- può non avere un nome (utile quando si passa una funzione o si aggiunge a un oggetto)
- i parametri mancanti sono impostati a `undefined`
- è possibile impostare dei valori di default per i parametri nella dichiarazione della funzione e.g. `somma(a, b = 1){}`
  - si può anche fare con degli if o con `??` (ma perché dovresti?)
- **One Function, One Action**
- le funzioni sono oggetti, quindi è possibile passare una funzione come argomento (**molto comune**)
  - queste funzioni si chiamano **Callback functions**

Arrow Function

- sistema sintetico per specificare una funzione
- e.g. `let somma = (a, b) => a + b;`
- scrivere ciò che si vuole ritornare

Espressione Funzionale

- definire una variabile e associarlo a una funzione
- e.g. `let somma = function (a, b) { ... }`
  `let s = somma(3, 2);`

- viene creata durante l'esecuzione del codice
- non c'è differenza nel funzionamento

Buona formattazione di funzioni

- JSDoc all'inizio
  - commento lungo /**/
  - @param {tipo} nome descrizione (anche più di uno)
  - @returns {tipo} descrizione
- Nome descrittivo in camel case
- definire "numeri magici" prima con const

Scope

- Visibilità di una variabile
- Variabili definite dentro una funzione hanno lo scope relativo alla funzione stessa (locale)
- Al di fuori globale (pericoloso)
- Namespace Pollution: meglio evitare variabili globali

Oggetti

- Non centra niente con un oggetto in Java
- Un contenitore di coppie "proprietà" "valore"
  - propertà: stringhe o simboli
  - valore: qualsiasi tipo primitivo, altro oggetto, o funzione
- `let object = {property: value, property2: value2, ...};`
  - Creazione Oggetto vuoto
    - `let object = {};`
    - `object = new Object();`
  - Modificare o aggiungere: `object.property = value;`
  - Accedere:
    - `object.property;`
    - `object[property];`
  - Cancellare:
    - `delete object.property;`: `object.property` diventa `undefined`
- **JavaScript Object Notation *(JSON)***
- *Non esistono valori "privati"*

Copiare un oggetto

- `let a = {nome: "pippo"}; let b = a`
- riferimento allo *stesso* oggetto, copiamo solo il riferimento a quell'oggetto con `let b = a;`
- `a == b;`: true
- `let a = {}; let b = {}; a == b;`: False
- va fatto a mano... proprietà per proprietà...

Metodi degli oggetti

- Un oggetto può avere tra le proprietà delle funzioni chiamate *metodi*
- `let a = {"nome": "pippo"};`
  `a.saluta = function() {alert("ciao sono " + this.nome);};` (`this` è valutato a "call time" non quando è definita la funzione)
- chiamare una funzione con `this` al di fuori di un oggetto si riferisce all'oggetto window
- per eseguire `a.saluta();` (`a.saluta;` ci ritorna la funzione)
  - `this` nelle arrow function si riferisce all'outer scope (l'oggetto che le contiene)

Costruttore

- Per creare oggetti uguali o simili usando funzioni
- e.g. `function User(name) { this.name = name; this.isAdmin = false; }`
  `let user = new User("Pippo");`: crea un nuovo oggetto e associa le proprietà con l'operazione di bind ("assegnamento" del `this`)
- se non si usa `new` il `this` punterà non all'oggetto ma alla finestra del browser boh, errore gravissimo

Garbage Collection

- JavaScript ha un garbage collector che elimina gli oggetti istanziati dinamicamente in memoria non più raggiungibili
- è sempre possibile che avvengano memory leak (riferimenti "circolari")

Array e stringhe

- Array e strings sono oggetti

Array

- contenitore di variabili eterogenee
- Oggetti con *proprietà numeriche* (index degli elementi, parte da 0) e metodi/attributi per la manipolazione
- Creare Array: `let arr = [elemtn, ement, ... ];`
  - `let a = []`, `let a = new Array()`, `let a = Array()`
- Modificare: `array[0] = bruh`

- Aggiungere:
  - `array.push("ciao")` alla fine
  - `array.unshift("bruh")` all'inizio
  - `array[10] = "ciao"` decimo posto (length diventerà almeno 11)
- Rimuovere:
  - `.pop`, `.shift`, `delete array[19]`, togliere e ritornare rispettivamente dalla fine e dall'inizio e 19simo posto
  - cancellare in posti specifici non accorcia l'array, diventano undefined
- Lunghezza: `arr.length;`
  - è possibile modificarla e imporre una lunghezza (???)
- Slicing: `slice(start_index, end_index)`
  - ritorna una porzione dell'array senza modificarlo
- Splice: `splice(index, n_elementi)`
  - rimuove n_elementi partendo da index modificando l'array
- Iteratori
  - `for..in`: itera sulle proprietà di un oggetto
    - `for (let i in arr)`
  - `for..of`: itera gli elementi di un array/mappa/set
    - `for (let i of arr)`
- Loop: `array.forEach(funzione)` applica una funzione a ogni elemento dell'array
- `array.indexOf(elemento)`: ti dice l'indice di un certo elemento, se non esiste ritorna -1
- `array.join(stringa)`: ritorna una stringa con tutti gli elementi separati dall'input

Map Reduce

- Map Reduce è un framework utilizzato per immagazzinare Big Data,
- contiene due funzioni: `map()` e `reduce()` che appartengono a ogni **Array**
- `map(funzione)` converte un array in un altro e lo ritorna
  - la funzione viene applicata a ogni elemento
  - la funzione prende l'elemento, l'indice dell'elemento e l'array (in questo ordine)
    - e.g. `let a = ["pippo", "pluto", "paperino"];`
      `a.map( (item, index, array) => item.lenght);`
- `reduce(funzione, [valore di inizio dell'accumulatore])`: serve a calcolare un singolo valore dall'array

- la funzione prende un accumulatore, l'elemento, l'indice dell'elemento e l'array

Stringhe

- Creazione
  - `let s = "stringa"`
  - `let s = new String("bruh")`
  - `let s = "stringa che \`
    `va a capo";`
- `s.indexOf("a");`: Ritorna indice prima occorrenza
- `.trim()`: toglie whitespace all'inizio e fine stringa
- `.charAt(num)`: Ritornare il carattere in un indice
- `.toUpperCase()`
- `.toLowerCase()`
- `.replace(da ripiazzare, rimpiazzo)` (anche Regex)

Formattazione Stringhe

- `let lezione = 3;`
  `console.log(`Questa è la lezione numero ${lezione}`);`

Oggetti built-in

- `Date`
- `Math`
- `JSON`
  - `.stringify(object)`: convertire oggetto in stringhe
  - `.parse(objectString)`: convertire stringa in oggetto
- `window`
  - Rappresenta la finestra del browser
  - chiamiamo i metodi indirettamente (senza fare `window.alert("alert");`)
  - `.alert()`, `.confirm()`, `.prompt()`
  - `.open()`, `.close()`: apre e chiude la finestra
  - `.print()`: stampa la pagina
  - `.ScrollTo()`: scroll fino a coordinate
  - `.setInterval(), .clearIntervail()`: richiama o annulla una funzione ogni tot tempo
  - `.setTimeout(), .clearTimeout()`: imposta o annulla un timer
    - i set ritornano un numero che diamo come input ai clear

- utile: `typeof` e `istanceof Oggetto`

Eccezioni

- indicano che qualcosa è andato storto
- può essere un qualsiasi tipo di dato
- il frammento di codice lancia un'eccezione con `throw` che può essere gestita con un `catch`
  - `throw eccezione`
  - `try { ... } catch (e) { ... } finally { // eseguita sempre ... }`
- spesso si lancia un oggetto di tipo `Error`
  - `throw (new Error("Error Message"));`

Closure

- Pattern di programmazione
- **Reminder**:
  - lo scope di `var` è il *functional block* più vicino (anche al di fuori di parentesi graffe)
  - lo scope di `let` è l'*enclosing block* più vicino (solo all'interno delle parentesi graffe)
  - se ci dimendichiamo di scrivere `var` o `let` diventerà una ***proprietà dell'oggetto*** `window`
- Possiamo definire funzioni all'interno di altre funzioni, ognuna con il suo scope (per `let`) (scope annidati)
  - inner function può accedere allo scope dell'outer function ma NON viceversa
- Una funzione può ritornare il return di una funzione, e queste funzioni annidate possono accedere a variabili della outer function
  - se invece si ritorna la funzione stessa è possibile "dividere" gli argument della chiamata a funzione in modo da usare quella ritornata
- **Closure**: una inner function dove lo scope "si chiude" su quello del padre (chiamato *ambiente*)
  - workaround per avere variabili e metodi privati
  - simulazione di OOP
  - utile anche per evitare variabili globali
  - e.g. `function salutatore(name) {`
    `let text = "Ciao" + name; // variabile locale`
    `let diCiao = function() { alert(text); }`
    `return diCiao;`
    `}` una specie di costruttore che ritorna un insieme di funzioni
    `let s = salutatore("Lorenzo");`
    `s(); // alerts "Ciao Lorenzo"`

Paradigma IIFE (Indipendently Invoked Functional Expression)

- per evocare una funzione anonima immediatamente e non avere variabili globali
- `(function () { ... }) ();`
- molto utilizzato

This

- un oggetto può avere come proprietà una funzione (ovviamente duh)
- la parola `this` usata dentro una funzione indica l'oggetto che la contiene
- se si è fuori da un oggetto `this === windows`

Bind

- permette di definire chi è il `this` per una funzione (metodo di una funzione -> la funzione è un oggetto)
- `bind(oggetto)`
  - `apply`: chiama una funzione impostando un certo this passando argomenti come array

Apply e Call

- `apply`: chiamare una funzione impostando un certo this e passando gli argomenti come array
  - `myFunction.apply(myObject, ["Susan", "Teacher"]`
- `call`: simile ad `apply`, ma gli argomenti sono passati esplicitamente
  - `myFunction.call(myObject, "Susan", "Teacher"`
- utili quando si crea un nuovo oggetto con costrutture

Ereditarietà

- gli oggetti JS hanno un prototipo (un altro oggetto da cui eredita tutte le proprietà)
- quando chiamiamo una proprietà di un oggetto:
  1. Cerchiamo tra le proprietà dell'oggetto
  2. Cerchiamo tra le proprietà del prototipo
  3. Cerchiamo tra le proprietà del prototipo del protitipo e così via
- `.__proto__`: vedere il prototipo di un oggetto
- modificare il prototipo modifica gli oggetti che ne sono derivanti
- `.hasOwnProperty(proprietà)`: per vedere se una proprietà appartiene all'oggetto o al protitipo
- ogni funzione ha la proprietà `"prototype"` il cui valore è un oggetto (quindi può contenere proprietà)
  - tutti gli oggetti creati col costruttore con un `prototype` ottengono le proprietà incluse nel prototipo (che possiamo aggiungere)

        - `__proto__` e `prototype` sono gli stessi in quel caso (creati da una funzione costruttore)
- motivazioni:
    - modificabilità di proprietà e funzione su molti oggetti creati con un costruttore
    - evitare replicabilità di instanziazione di funzioni e di codice

Manipolazione del DOM (Document Object Model)

- Il DOM è un'interfaccia di programmazione per HTML e XML
- Mappa strutturata del nostro documento e i metodi per interfacciarsi con gli elementi
    - Ogni elemento della pagina è un nodo
    - La radice è "document"
        - "document" ha una serie di proprietà standard
        - come `.getElementById(Id)`
        - e.g. `let foo = document.getElementById("miodiv").innerHTML;` (foo conterrà il contenuto HTML del div con id "miodiv")

Selettori Javascript

- Proprietà di `document`
- `.getElementById("Id")`: ritorna un node (id è univoco)
- `.getElementByTagName("Tag")` (per esempio p): ritorna una nodelist
- `.getElementsByClassName("classe")`: ritorna una nodelist
- `.querySelector("query")` (selettori css e ritorna una NodeList (simile a un array) e.g. `"p. warning"`)

Eventi

- JavaScript può "captare" diversi eventi che avvengono nella pagina e utilizzarli per far eseguire codice, si aggiungono alle tag
- e.g. `<button onclick="..."> ... </button>`
- `onblur/onfocus`: elemento prende/perde il focus
- `onclick`: click del mouse
- `onerror`: errore nel caricamento
- `onload`: la pagina ha finito di caricarsi
- `onkeydown/onkeypress/onkeyup`: un tasto viene premuto/tenuto premuto/rilasciato
- `onmousedown/onmouseup`: bottone del mouse viene premuto/rilasciato
- `onmousemove/onmouseout/onmouseover`: mouse è stato spostato/spostato fuori da un elemento/spostato sopra un elemento
- Per form

- `onchange`: quando cambia il contenuto del form
- `onsubmit`: il form è stato submittato

Associare un evento a un elemento

1. Attributo HTML
   - `<body onclick="myfunction();">`
2. Metodo
   - `window.onclick = myFunction;`
   - `document.getElementById("miodiv").onclick = ...`
     3. con `addEventListener` (scelta preferita)
   - `windows.addEventListener("evento", myFunction);`

- ATTENZIONE: codice JavaScript deve essere eseguito dopo che la pagina è caricata, perché la pagina viene "letta" dall'alto verso il basso
- Il browser quando vede `<script>...</script>` lo scarica e continua a caricare la pagina, questo può causare il malfunzionamento del codice perché è possibile che il codice non vede elementi ancora caricati della pagina:
  1. Mettere il codice alla fine (sotto il `<body>...</body>`) (potremmo poter dover attendere troppo)
  2. Usare window.onload = function() {}
  3. usare un file javascript separato con le keyword `async` `defer` in `<script>`
     - `defer`: browser continua il caricamento della pagina e carica lo script in background e lo esegue quando è carica (la pagina)
     - `async`: rende lo script indipendente dal caricamento della pagina e viene eseguito appena è carico (lo script)

Manipolazione Nodo

- `.getAttribute("attributo")`: legge un attributo
- `.setAttribute("attributo", "nuovo valore attributo")`: scrivere attributo
- `.innerHTML = "nuovo codice html"`: leggere/modificare HTML
- `.style.proprietàCSSInCamelCase = "nuovo valore proprietà"`

Creazione Nodo

- proprietà di `document`, da assegnare a una variabile, crea un nodo
  - `.createElement("nome elemento")` (per esempio "p")
  - `.createTextNode("testo nel nodo")`: crea un nodo contenente testo
- proprietà di un nodo del DOM:
  - `.appendChild(nodo)` (modo di utilizzo: append del testo nel nuovo nodo, poi append del nodo a un elemento del DOM)

- `.insertBefore(newNodo, nodo)`: inserisce il nodo newNodo prima del nodo nodo, hanno lo stesso genitore
- `.replaceChild(newNodo, oldNodo)`
- `.removeChild(nodeToRemove)`

Javascript Asincrono

- **Sincrono**: il codice è eseguito linea dopo linea, le operazioni lunghe bloccano l'esecusione del programma
- **Asincrono**: questo non accade, non aspetto la fine della riga 1 per eseguire la riga 2
  - per esempio utilizzando `setTimeout(funzione)` per eseguire una funzione in un altro momento, non quando la linea prima viene eseguita
  - anche `.addEventListener`
  - il codice sincrono continua la sua esecuzione
  - il codice asincrono è collegato a funzioni particolari
  - **le callback non rendono il codice asincrono**
  - le immagini sono caricate in modo asincrono
    - un evento `load` di un'immagine viene eseguito alla fine del caricamento

Event Loop

- Quando eseguiamo il nostro codice viene caricato nell'interprete e lo esegue riga per riga
- Se trova un'operazione di b``ackground utilizza le API del browser e fa partire un thread in background
- Le risposte arrivano dalla *coda delle callback* (*message queue*)
- Un **Event Loop** controlla se ci sono messaggi nella message queue, se ci sono verranno inseriti nel Call Stack, e verranno eseguiti quando il codice sincrono finisce

Promises

- Una **Promise** è un oggetto che rappresenta l'*esito* di un'operazione asincrona
  - Contenitore per un valore assegnato in modo asincrono
  - Contenitore per un valore futuro
  - Non serve più un evento e una callback per gestire il risultato asincrono
  - Concatenazione promises possibile
- `fetch(url)` crea un oggetto fetch e chiama una task in background (Producing Code)
  - fa una GET e ritorna una *promessa* di un risultato
  - `.then(response => response.json())` esiti positivi e negativi
  - `.catch(response => response.json())` gestione errori
  - (Consuming Code)

Ciclo di vita

- pending -> settled
- ogni promise generata non ha uno stato (l'evento ancora non è terminato)
- quando termina termina in modo negativo (`reject(error)` resituendo errore) o positivo (`resolve(value)` restituendo il risultato)
- la promise non può tornare indietro

Creazione di una promise

- ```
  let promise = new Promise(function(resolve, reject) {
  // la funzione è eseguita automaticamente quando la promise è costruita
  // dopo 1 secondo segnala che l'esecuzione è terminata con il risultato "done"
  setTimeout(() => resolve("done"), 1000);
  });
  ```
- la funzione deve avere due parametri: `resolve` e `reject`
- lo stato della promise inizia a `undefined`, quando arriva `resolve("done")` quello diventa lo stato
- a `reject()` può essere inviato un tipo `Error` e.g. `reject(new Error("Whoops!"))`
- capita poco di generare una promise

Consumazione di una promise

- i `resolve` e i `reject` si intercettano e consumano con la funzione `.then`
  - ha due parametri
  - `.then (function(result) {...}, function(error) {...});`
  - il secondo parametro non si utilizza perchè è possibile utilizzare il metodo `.catch` per intercettare gli errori
  - `.then(...{...}).catch(...{...});`
- `.finally` viene sempre intercettato
- è possibile concatenare `.then` (codice non più innestato ma lineare)

Microtask

- Utilizzando le promise il codice che intercetta è un codice speciale che si chiama **Microtask**
- quindi le callback che vengono passate alle promise
- non va in collisione con gli eventi di sistema
- ma appartengono a una coda a priorità maggiore rispetto alla coda delle callback chiamata **Microtask Queue**

AJAX

- **A**synchronous **J**avaScript **A**nd **X**ML
- Framework storico richieste HTTP asincrone
  ```

- Combinazione di:
  - Un oggetto `XMLHttpRequest` incorporato nel browser
  - JavaScript e HTML DOM
- Funzioni:
  - Aggiornare pagina web senza ricaricarla
  - Richiedere e ricevere dati a un server dopo che la pagina è stata caricata
  - Inviare dati a un server in background

XMLHttpRequest

- Un oggetto incorporato nel browser
- Ha la funzione di creare una richiesta web
- `.open('TIPO', url, se è asincrono o no (booleano))` (TIPO è GET, POST, etc...s): crea la richiesta
- `.send()`: invia la richiesta
- `.responseText`: la risposta (DOMString)
- `.onreadystatechange = function` usata in caso asincrono

AJAX con Fetch

- Fetch è un framework API che utilizza *promises*
- rimpiazza `XMLHttpRequest` (wrapper)
- `fetch(url)` ritorna una promise
- Pipeline
  1. Fetch ritorna una promise che avrà una respose `res`
  2. Vediamo lo status (404, 200, 500, ...)
  3. Analisi della response con `.text()`, `.json()`
  4. Manipolazione DOM in base al risultato ritornato
     . Possono avvenire errori tra le fasi che verranno catturati dalle `catch()`
- è possibile fare anche `POST` per inviare dati al server

ASYNC AWAIT

- Costrutto che semplica la leggibilità del codice delle richieste di rete
- Il compilatore convertira nel modo tradizionale con `.then()` concatenati e `.catch()`
- `async`: si dichiara davanti a una funzione, **la funzione ritorna sempre una promise**, ***non*** diventa asincrona, ma può gestire codice asincrono
  - quindi possiamo usare i `.then()` per gestirla quando termina
- `await`: rende il codice asicrono e aspetta la risposta (si usa nelle funzioni con `async`)

    - il codice si blocca, va in background e aspetta fino a che il codice asincrono si risolve e.g. `await promise1;`
- gli errori si gesticono col buon vecchio `try catch`
- nasconde i `.then()`

CORS

- Cross-origin HTTP Request
- un browser permette agli script contenuti in una pagina web di accedere ai dati contenuti in un'altra risorsa web solo se entrambe le pagine hanno la stessa origine (*Same Origin Policy*)
- Uno script js che fa una chiamata http a un differente dominio, protocollo o porta
- CORS sta per **C**ross **O**rigin **R**esource **S**haring
- è uno standard *W3C* che permette di accedere a risorse ospitate su un dominio diverso
- viene implementando inviando degli Header HTTP in req/resp
    - **Semplice**: GET, POST, HEAD + header standard
    - **Preflight**: PUT, DELETE, header custom, ecc.
- **gettonato per l'esame**

Simple Request

- GET, HEAD, POST
- Header permessi: Accept, Accept-Language, Content-Language, Content-Type (solo application/x-www-form-urlencoded, multipart/form-data e text/plain)
- Si aggiunge all'header un elemento di tipo `Origin:`

Preflight Request

- Quando devo inviare dati (e.g. autenticazione)
- Fa una prima chiamata al server di tipo `OPTIONS` che dichiara tutti gli header che l'utente vuole utilizzare (e.g. cookie, authentication, ..., tipi di contenuti)
- Il server invia una risposta 204 No Content che autorizza o meno tutto o niente di quello che l'utente vuole fare
- Successivamente si può fare la richiesta

Programmazione Server

- **Frontend:** Browser, HTML, CSS, JavaScript e librerie/framework annesse (con pattern reattivi per far collaborare HTML e JS);
- **Backend:** Web Server con Server HTTP, App e file, Database
    - **PHP** Tecnologia più usata
- Si passa da siti statici (generati dall'HTML, CSS e JS) a siti dinamici (dove il sito è generato al *runtime* partendo da dati nel DB con un Site Builder che si trova nel Web Server)

- I siti dinamici si evolvono con l'utilizzo di API JSON che generano dati prendendoli dal DB, il Site Builder si sposta nel Browser

NodeJS

- Ambiente runtime JavaScript costruito sopra l'engine JavaScript V8 di Google
- Ci permette di scrivere codice JavaScript su qualsiasi macchina dove NodeJs può essere installato
- Contiene:
  - **V8**
  - **libuv**: libreria C per creare l'event loop (gestione task, microtask, queue) e thread pool (mandare in background i processi)
  - http-parser
  - z-lib
  - OpenSSL
  - c-ares

Moduli

- Pezzo riutilizzabile di codice che incapsula un'implementazione
- Moduli possono chiamare altri moduli ed esportare/importare funzionalità tramite direttive speciali
- Moduli in JS:
  - AMD: antico, inizialmente implementato dalla libreria `require.js`
  - **CommonJS:** sistema di moduli creato per il server Node.js
  - ES6 Modules: sistema di moduli a livello di linguaggio apparso nello standard nel 2015
- `require("modulo")`: importa un modulo, da assegnare a una variabile
  - e.g. `const http = require("http")`
- Tipi di modulo con NodeJs
  - **Core Modules**: integrati con Node
    - `http`: include classi, metodi ed eventi per creare un server http Node.js
    - `url`: include metodi per la risoluzione e parsing di URL
    - `querystring`: include metodi per occuparsi di stringhe di richieste/query
    - `path`: include metodi per occuparsi di percorsi file
    - `fs`: include classi, metodi ed eventi per lavorare con I/O di file
    - `util`: include funzioni di utilità utili per i programmatori
  - **Local Modules**: creati localmente, anche un semplice file .js che fa hello world è un modulo
  - **Third Party Modules**: s.e. (e.g. *Express*)
- Creazione di Moduli con l'oggetto `exports.funzione = funzione`

- oppure con `module.exports = funzione` che cambia l'oggetto esportato, sostituendolo con la funzione (per chiamarla bisogna usare il nome della variabile al posto di farlo tramite l'oggetto)
- I moduli esportano un oggetto, se si usa `exports` si può aggiungere a questo oggetto, mentre `module.exports` sostituisce direttamente
    - ricordare `/` prima del nome del file js per indicare che è locale se facciamo path relativo
- è possibile importare solo determinate funzioni con `require` assegnando a un oggetto senza valori (solo chiavi) con il nome delle funzioni al modulo
    - e.g. `const {funzione1, funzione2} = require("modulo")`
- Si può "spacchettare" un `require` e assegnare nuovi nomi alle funzioni con **Destructuring**
    - `const {nome_funzione_nel_modulo: nome_funzione_nel_codice, ...} = require(...)`