

1 LEZIONE

Molte delle informazioni che vedremo riguardante **JavaScript**, sono strettamente legate al mondo dei linguaggi di programmazione **HTML** e **CSS**. Immaginiamo come se fosse una **TORTA** composta da 3 strati.

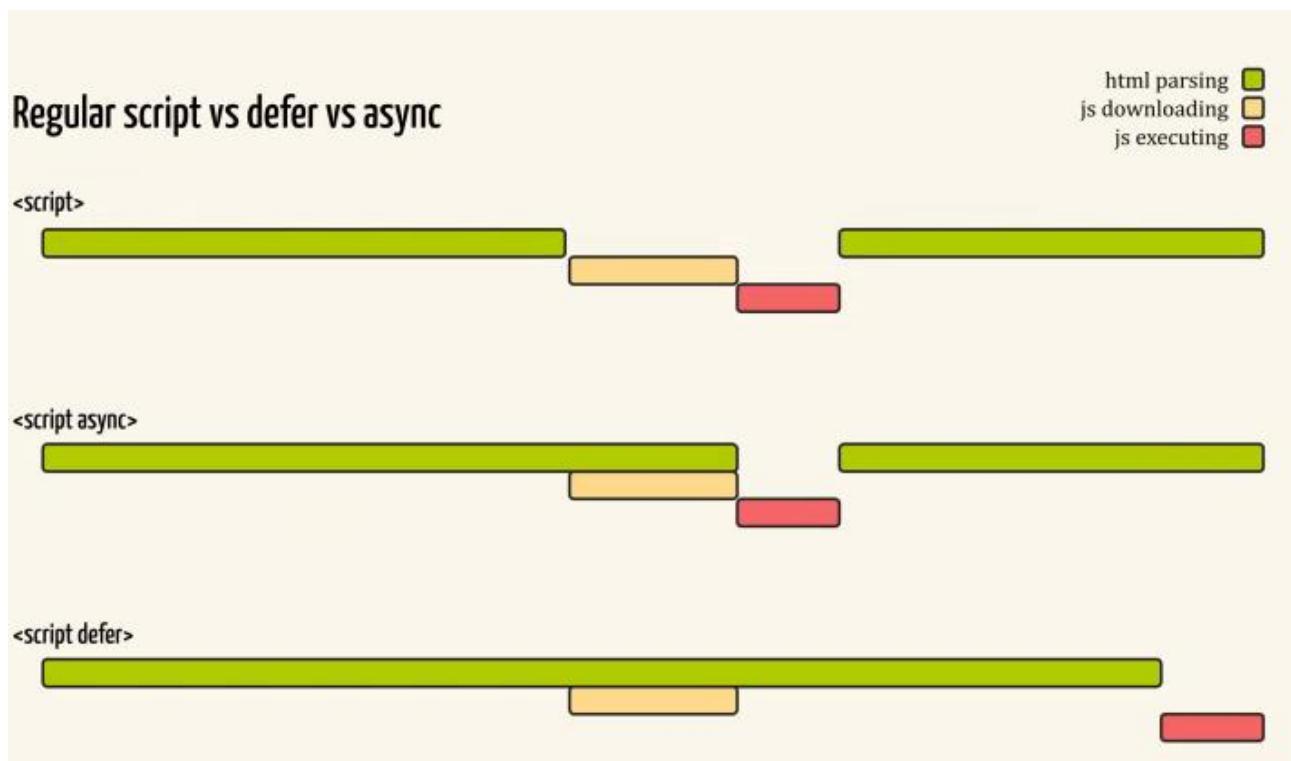
-**JavaScript** → è un linguaggio di programmazione **client-side** che permette di implementare la logica nelle nostre pagine web

un elemento di JavaScript vive all'interno di uno **<script>** all'interno della struttura dell'HTML.

-**ECMAScript** → rappresenta lo standard su cui si basa JavaScript, e le diverse versioni aggiungono nuove funzionalità (mantenuto dall'organizzazione **Ecma International**).

-**TypeScript** → aggiunge alcune funzionalità in più a JavaScript, in particolare il supporto per il typing statico. (serve principalmente per **REACT** e **ANGULAR**).

-**ASYNC & DEFER** → indica il modo in cui il js viene letto durante il parsing (lettura del document)



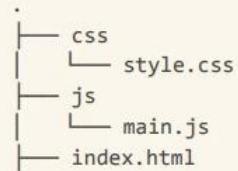
COLLEGARE UN FILE JAVASCRIPT A HTML

è molto più LOGICO, SINTETICO e ORDINATO distinguere i vari file di linguaggio diverso in locazioni diverse, possono essere INLINE oppure EXTERNAL, per collegare i file **JS** al **HTML**.

```
<script src="js/main.js"></script>
```

STRUTTURA delle cartelle e file:

```
<body>
  <!-- end of the body -->
  <script src="js/main.js"></script>
</body>
```



Esistono 2 tipi di “posizioni” di dove si trovano i file:

- **REMOTO**: quando il file da collegare NON è sulla stessa macchina.
- **LOCALE**: quando il file da collegare è sulla stessa macchina.

Remote:

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
```

Local:

```
<script src="js/main.js"></script>
```

INTRODUZIONE AL LINGUAGGIO DI PROGRAMM. JAVASCRIPT

si occupa della logica che permette di compiere delle azioni dinamiche, è un **linguaggio versatile**, utilizzato per sviluppare il lato client (nel browser)

JavaScript è responsabile per il comportamento nel sito! ci permette di implementare delle funzionalità complesse nelle nostre pagine, tra le principali abbiamo:

- aggiornare il contenuto del sito con nuovi contenuti.
- disegni, animazioni, gallerie di immagini...
- tenere traccia degli utenti tramite i COOKIE.
- Elementi interattivi (schede, cursori..)

INTRODUZIONE A NODE.JS

Node.js è un ambiente di runtime open-source JavaScript basato sul motore di Google Chrome (V8)

consente di eseguire codice JavaScript **lato server**, consentendo agli sviluppatori di creare applicazioni web scalabili, veloci ed ad alte prestazioni; inoltre semplifica lo sviluppo e la manutenzione del codice.

Node.js utilizza un modello basato sugli eventi che lo rende possibile Leggero ed efficiente, perfetto per applicazioni in tempo reale ad alta intensità di dati che vengono eseguiti su dispositivi distribuiti.

LIVELLI DEI LINGUAGGI DI PROGRAMMAZIONE

-**LINGUAGGI MACCHINA**: la lingua + vicina al computer stesso, un programma in linguaggio macchina è **costituito da una serie di modelli binari** (EX: 01011100), che rappresentano semplici operazioni che possono essere eseguite dal computer.

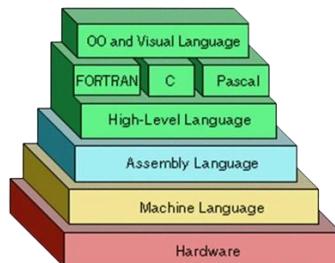
I programmi in linguaggio macchina sono **eseguibili**, significa che possono essere eseguiti direttamente.

-**LINGUAGGI ASSEMBLY**: rendono la **programmazione + facile per l'uomo**. **Le istruzioni in linguaggio macchina vengono sostituite da semplici abbreviazioni** (EX: ADD, MOV), prima dell'esecuzione, un programma in linguaggio assembly richiede la **traduzione in linguaggio macchina**.

traduzione viene eseguita da un programma per computer noto come **assembler**.

-**LINGUAGGI DI ALTO LIVELLO**: i linguaggi di alto livello (EX: C C++, JAVA...) sono più **simili all'inglese**, questo rende più facile ai programmati pensare nel linguaggio di programmazione. Anch'essi, richiedono anche la traduzione in linguaggio macchina prima dell'esecuzione.

Traduzione viene eseguita da un **compilatore** o da un **interprete**.



LINGUAGGIO COMPILOTATO E INTERPRETATO

il linguaggio compilato ed interpretato sono caratterizzati e **dipendono da dei processi di implementazione** che contano molto di + delle proprietà del linguaggio di programmazione.

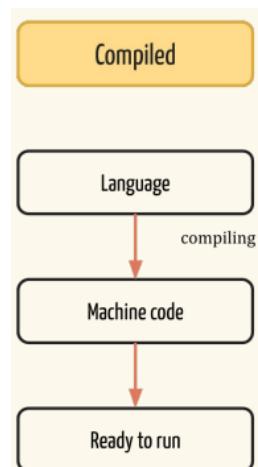
dicendo questo, **QUALSIASI** linguaggio potrebbe essere COMPILATO oppure INTERPRETATO a seconda dell'implementazione utilizzata.

- **LINGUAGGIO COMPILOTATO** → cosa è la COMPILAZIONE? (C, C++, Pascal...)

Il compilatore converte il programma direttamente in **codice macchina**, riferendosi al codice progettato per un determinato processore e OS, successivamente **il computer indipendentemente esegue il codice macchina**. comporta l'effetto negativo, che la compilazione sia lenta.

VANTAGGI:

- Esecuzione rapida.
- Ottimizzato per l'hardware di destinazione.



SVANTAGGI:

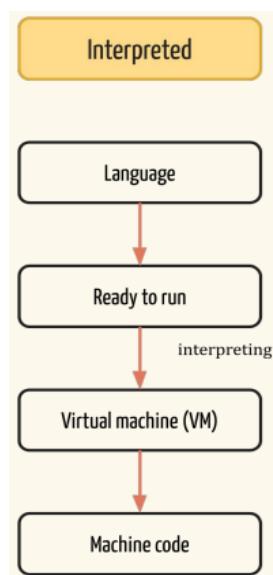
- Richiede un compilatore.
- Modifiche e Distribuzione del codice sono + lenti dell'interprete.

- **LINGUAGGIO INTERPRETATO** → cosa è l'INTERPRETAZIONE? (Python, PHP, Ruby, JavaScript...)

qui il codice sorgente **NON VIENE** eseguito direttamente dalla macchina, c'è la presenza di un altro programma (**interprete**) che lo legge ed esegue l'interprete è personalizzato per la macchina nativa.

VANTAGGI:

- Facile da imparare, usare.
- Consente di eseguire operazioni + complesse in pochi passaggi.
- Permettono l'aggiunta di attività dinamiche e interattive alle web.
- Modifica e Esecuzione del codice è veloce.



SVANTAGGI:

- Di solito funziona abbastanza lentamente.
- Comandi limitati per eseguire operazioni dettagliate sulla grafica.
- Accesso limitato al codice di basso livello e di ottimizzazione della velocità.
- Accesso limitato al dispositivo

LINGUAGGIO COMPILATI E INTERPRETATI DIFFERENZE IMP.

-**un compilatore** converte l'intero codice sorgente in codice oggetto, **e lo salva** come file prima di eseguirlo, tra i linguaggi più noti di questo tipo abbiamo (C, C++, COBOL, Fortran).

Come funziona un compilatore? :

- **Creazione del codice sorgente:** il codice sorgente è un pezzo di codice composto in un editor di testo e l'estensione del file per il codice sorgente.
- **Pre-elaborazione:** questo codice sorgente viene inizialmente trasmesso al preprocessore, che lo espande. Il codice ingrandito verrà fornito al compilatore dopo l'espansione.
- **Compilazione:** il codice espanso dal preprocessore viene passato al compilatore, che lo converte in codice assembly.
- **Conversione in codice oggetto** da parte di un assembler: utilizzando un assembler, il codice assembly viene trasformato in codice oggetto. Il file oggetto creato da un assembler ha lo stesso nome del file di origine.
- **Collegamento:** nel flusso di lavoro del compilatore, la funzione principale di un linker consiste nel connettere il codice oggetto dei dati della libreria di codifica con il codice oggetto di un programma.
- **Esecuzione:** il file eseguibile è il prodotto finale del linker.

-**un interprete** trasforma ed esegue il codice sorgente riga per riga, senza salvarlo e segnala gli errori lungo

il percorso. tra i linguaggi più noti di questo tipo abbiamo (Python, JavaScript, Perl e BASIC)

Come funziona un interprete? :

- **Creazione del codice sorgente:** questo passaggio della funzionalità è lo stesso di un compilatore. Durante il runtime, però, l'interprete trasforma il codice sorgente una riga alla volta.
 - **Interpretazione diretta:** un interprete traduce un programma linguistico di alto livello in un linguaggio a livello di macchina.
 - **Modifica del codice sorgente:** l'interprete consente la valutazione e la modifica del programma durante l'esecuzione in una finestra affiancata.
 - **Esecuzione:** l'esecuzione del programma è moderatamente lenta poiché tutti i collegamenti vengono eseguiti in fase di esecuzione senza un linker separato.
-

2 LEZIONE

Spiegazione pratica con persone di come funziona il linguaggio compilato e il linguaggio interpretato

-**REPL** → Read Execute Print Loop

Un **programma per computer** è un elenco di "istruzioni" da "eseguito" da un computer. In un linguaggio di programmazione, queste istruzioni di programmazione sono chiamate **istruzioni**.

Un **programma JavaScript** è un elenco di **istruzioni di programmazione**, sono composte da:

<i>Valori</i>	<i>Operatori</i>	<i>Espressioni</i>	<i>Parole chiave</i>	<i>Commenti</i>
---------------	------------------	--------------------	----------------------	-----------------

INTRODUZIONE ALLE VARIABILI

La variabile è una scatola che contiene delle informazioni di un certo tipo, una variabile può avere **1 solo valore**, il valore può essere inizializzato quando si crea la variabile o anche successivamente, tutte le variabili le inizializziamo con → **LET**

tra le variabili più note abbiamo:

- **Number** → contiene tutti i tipi di numeri (8, -4, 7.6) quindi int double float

```
let myAge = 28;  
let pi = 3.14;
```

- **String** → contiene un carattere o una sequenza di caratteri ("gabriele" , 'G') possiamo usare le " " o gli '' (cercare di non mischiarli, o si usa oppure un altro), tramite il comando length, possiamo vedere da quanti caratteri è composta la nostra stringa, se vogliamo muoverci sul singolo carattere, usiamo gli array[]

```
let alphabet = "abcdefghijklmnopqrstuvwxyz";
```

- **Boolean** → può contenere 2 valori logici: **true / false**

```
let catsAreBest = true;  
let dogsRule = false;
```

- **Undefined** → rappresenta un valore che non è stato definito

```
let notDefinedYet;
```

- **Null** → se non vogliamo dare nessun VALORE alla VARIABILE

```
let goodPickupLines = null;
```

- **i** → NON è obbligatorio usarlo ma noi lo utilizzeremo lo stesso, viene utilizzato per separare + istruzioni

- **typeof** → viene utilizzato per capire il tipo di una variabile, JavaScript individua il tipo in base al VALORE, se concateniamo 2 valori diversi (number e string), converte anche il numero in string

```
A variable can only be of one type:  
let y = 2 + ' cats';  
console.log(typeof y);
```

→ //output STRING

- **let** → sono delle variabili in cui il loro valore e tipo cambia durante il programma, se viene dichiarata ad esempio all'interno di un ciclo for, una volta fuori il suo valore sarà UNDEFIND

- **const** → const sono delle variabili in cui il loro valore e tipo **RIMMARRA FISSO**, e non può essere cambiato, se viene dichiarata ad esempio all'interno di un ciclo for, una volta fuori il suo valore sarà SEMPRE QUELLO

https://www.w3schools.com/js/js_let.asp

INTRODUZIONE AGLI OPERATORI

Gli operandi possono essere da 1 a 3, parlando dei vari tipi di operatori abbiamo:

TIPO ARITMETICI

+ → let x = 12 + 5;

- → let x = 12 - 5;

* → let x = 12*5

/ → let x = 12/5

% → let x = 12 % 5 (il resto di 12/5 nella divisione matematica, è 2)

TIPO ASSEGNAZIONE

+= → x += y (è l'equivalente di x = x + y)

-= → x -= y (è l'equivalente di x = x - y)

*= → x *= y (è l'equivalente di x = x * y)

/= → x /= y (è l'equivalente di x = x / y)

%= → x %= y (è l'equivalente di x = x % y)

TIPO INCREMENTO

```
// increment occurs before a is assigned to b
let a = 1;
let b = ++a; // a = 2, b = 2;

// increment occurs to c after c is assigned to c
let c = 1;
let d = c++; // c = 2, d = 1;
```

TIPO COMPARAZIONE

== → confronta sia il valore, ma anche il tipo di variabile (confronta **uguaglianza STRETTA**)

!= → diverso sia il valore, diverso anche il tipo di variabile (confronta **disuguaglianza STRETTA**)

== → confronta il valore (confronta **uguaglianza NON STRETTA**)

!= → diverso dal valore (confronta **disuguaglianza NON STRETTA**)

< > >= <= → sono di comparazione **numerica**

NON CONFONDERE L'OPERATORE = CON == !!

TIPO LOGICO BINARIO

opera su dei valori ad esempio x = 6 y = 3

&& → and (x < 10 && y > 1) is true

|| → or (x == 5 || y == 5) is false

! → not !(x == y) is true

TIPO STRINGHE

OGGETTI MATEMATICI IN JS

[Oggetto matematico JavaScript \(w3schools.com\)](#)

OPERATORI PRECEDENZA IN JS

https://www.w3schools.com/js/js_precedence.asp

3 LEZIONE

FUNZIONI IN JS

Richiamano un pezzo del codice che permette di essere utilizzato e ri-utilizzato **+ volte**, molto utile perché se vogliamo modificare la funzione di nascita, viene modificata x TUTTI.

viene eseguita quando "qualcosa" la invoca (la chiama). https://www.w3schools.com/js/js_functions.asp

```
// declare
function sayMyName() {
    console.log('Hi Bob!');
}

// use
sayMyName();

// use again
sayMyName();
```

Possiamo anche concatenare un parametro (nome), che è valido solo nella funzione dato che è stato assegnato per essa, con un argomento (ciao!):

```
function sayMyName(name) {
    console.log('Hi, ' + name);
}

sayMyName('James');
sayMyName('Adam');
```

-Return → ritorna il valore calcolato all'interno di una funzione, che può essere assegnato a chi richiama la funzione, inoltre funziona come ultimo elemento di chiusura della funzione (tutto quello scritto dopo non è valido!)

```
function areaOfCircle(raggio) {
    let areaCerch = Math.PI * raggio * raggio;
    return areaCerch;
}

let areaCerchio = areaOfCircle(raggio);
```

-Circular dependencies → Viene chiamata **Recurzione**, è quando una funzione che va ad utilizzare una funzione che ha la funzione stessa, **PROBLEMA**: c'è il rischio che il compilaggio non termini mai

<https://medium.com/@williambdale/recursion-the-pros-and-cons-76d32d75973a>

```
function chicken() {
    egg();
}

function egg() {
    chicken();
}

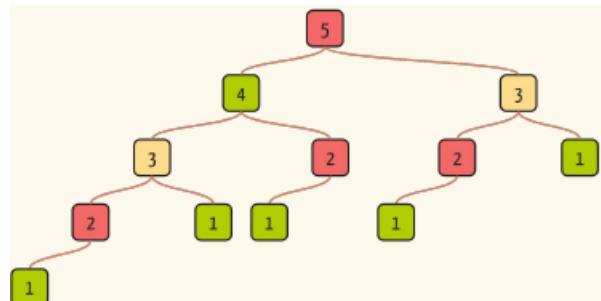
egg();
```

-Funzione di Fibonacci → i ms (millisecondi) impiegati per la sua esecuzione variano in base alla macchina, rete, sistema operativo ecc..

È un algoritmo recursivo con numeri esponenziali, + è alto è il numero, + sarà lungo il processo di elaborazione:

```
function fibonacci(n) {
    if (n < 2) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}

fibonacci(30); // 1439 ms
fibonacci(35); // 12765 ms
fibonacci(40); // 121211 ms
```



SCOPO DELLE VARIABILI

-Variabili locali → inizializziamo la variabile all'interno di una funzione, questa variabile una volta usciti dalla funzione, NON avrà nessun valore siccome è una variabile LOCALE che vive dentro la funzione (**undefined**), come in questo caso:

```
A variable with "local" scope

function addNumbers(num1, num2) {
    let localResult = num1 + num2;
    console.log("The local result is: " + localResult);
}

addNumbers(5, 7);
console.log(localResult);
//output → undefined
```

-Variabili globali → inizializziamo la **variabile globalmente**, questa variabile una volta usciti dalla funzione, **avrà il valore** calcolato all'interno della funzione, siccome è una variabile GLOBALE che **(number)**, come in questo caso:

```
A variable with "global" scope

let globalResult;

function addNumbers(num1, num2) {
  globalResult = num1 + num2;
  console.log("The global result is: " + globalResult);
}

addNumbers(5, 7);
console.log(globalResult); // output → 12
```

LET & VAR

Let è stato introdotto a partire da **JS ES6** (2015), dato che il var creava tanta confusione, parlando dei comportamenti possiamo dire:

-**let** → sono soggette a **block scope**. Ciò significa che sono visibili solo all'interno del blocco di codice in cui sono state dichiarate.

-**var** → sono soggette a **function scope**. Ciò significa che sono visibili all'interno della funzione in cui sono state dichiarate,

4 LEZIONE

Promemoria **importanza documentazione** e commenti da usare all'interno dei nostri esercizi, un esempio pratico:

```
/**  
 * Returns the sum of num1 and num2  
 * @param {number} num1 - the first number  
 * @param {number} num2 - the second number  
 * @returns {number} Sum of num1 and num2  
 */  
function addNumbers(num1, num2) {  
    return num1 + num2;  
}
```

il **FLOW**, indica e significa **flusso** e comprende decisioni e punti di diramazione cioè **if/else**, oppure cicli di ripetizioni come **for/while** (fai questa cosa TOT volte).

-**IF / ELSE** → se la condizione al suo interno è verificata (si trova in mezzo alle **()**) esegue l'espressione contenuta nel blocco, se NON è verificata, allora salta il blocco e vai quello successivo, ci possono essere **+ IF**

utilizziamo all'interno della condizione → **operatori di comparazione** (`= == === != !== < > <= >=`)... e **logico binari** (`&& || !`) ...

```
if (condition) {  
    // statements to execute  
}  
let x = 5;  
  
if (x > 0) {  
    console.log('x is a positive number!');  
}
```

-**SWITCHCASE** → lo so dai...

- **TRUTHY / FALSEY** → ci sono alcuni valori che sono per definizione “**false**”, quindi se mettiamo una condizione simile a:

```
let points = 0;
if (points) {
  console.log('You have ' + points + ' points');
}
```

In questo caso **NON** verrà eseguito il codice nelle graffe, perché alcuni dei seguenti valori sono FALSEY:

```
Values that are "false-y":
false, the empty string (""),
the number 0, the number -0, undefined, null, NaN
```

- **SHORT-CIRCUIT EVALUATION** → JS valuta gli operatori logici da sinistra a destra e interrompe la valutazione non appena conosce la risposta.

In questo caso si fermerà alla **prima condizione** denominatore != 0 e NON controlla + avanti:

Examples:

```
let nominator = 5;
let denominator = 0;
if (denominator != 0 && (nominator/denominator > 0)) {
  console.log('That's a valid, positive fraction');
}
```

CICLI DI RIPETIZIONE

- **WHILE** → i know bruh

- **FOR** → è composto da 3 parti principalmente:

```
for (initialize; condition; update)
  // statements to repeat
}

for (let i = 0; i < 5; i = i + 1) {
  console.log(i);
}
```

-**BREAK** → anche se il loop può continuare anche se una condizione al suo interno risulta vera, con il break possiamo uscire PERMANENTEMENTE dal loop:

```
for (let current = 100; current < 200; current++) {
  console.log('Testing ' + current);
  if (current % 7 == 0) {
    console.log('Found it! ' + current);
    break;
}
```

5 LEZIONE

STRINGHE

le stringhe sono un INSIEME di caratteri o anche un SINGOLO carattere (NON esiste char in JS!), racchiudiamo la nostra stringhe tra doppi “ “, oppure singole ‘ ’

```
// this is a string
let client = "James";

// this is also a string
let bestFriend = 'Robbie';

There are cases when it's useful to mix quotes:

let status = "It's raining";
let answer = "The password is 'Bigfoot'";
let alternative = 'The password is "Bigfoot"';
```

METODI STRINGHE → https://www.w3schools.com/js/js_string_methods.asp

-**.length** → va a leggere la lunghezza della mia stringa, la variabile contenente questa funzione prenderà come tipo NUMBER (nel caso dell'alfabeto uscirà **26**) → E UNA PROPRIETA!!

[arguments.length - JavaScript | MDN \(mozilla.org\)](#)

```
// length
const alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
let alphabetLength = alphabet.length;
```

26

number

- **.concat()** → concatena una stringa ad un a + stringhe e restituisce una nuova stringa, e sarà di tipo **STRING**:

[String.prototype.concat\(\) - JavaScript | MDN \(mozilla.org\)](#)

```
const str1 = 'Hello';
const str2 = 'World';

console.log(str1.concat(' ', str2));
```

Hello World

- **.charAt()** → la variabile a cui diamo questo metodo, prenderà il valore (carattere) della posizione data in numero di quella lettera (da utilizzare al posto delle [] per accedere, può causare confusione con gli array), sarà di tipo **STRING**:

[String.prototype.charAt\(\) - JavaScript | MDN \(mozilla.org\)](#)

```
// charAt()
let greeting = "HELLO WORLD";
let result = greeting.charAt(0);
```

H

string

- **.indexOf()** → andiamo a cercare la posizione di una sottostringa all'interno di una stringa, iniziando a contare da 0 dal primo carattere, il risultato che uscirà sarà il primo carattere di quella sottostringa di tipo **NUMBER** (in questo esempio uscirà: **13**)

[String.prototype.indexOf\(\) - JavaScript | MDN \(mozilla.org\)](#)

```
// indexOf()
let statement = "Hello world, welcome to the universe.";
let wordPosition = statement.indexOf("welcome");
```

13

number

- **.split()** → prende un modello e divide questa stringa in un elenco ordinato di sottostringhe cercando il modello, inserisce queste sottostringhe in un array e restituisce l'array.

[String.prototype.split\(\) - JavaScript | MDN \(mozilla.org\)](#)

```
const greeting = 'Hello world';

let saluti = greeting.split("He");
console.log(saluti);
```

[, 'llo world']

-.slice() → va ad estrarre una sezione della stringa e lo restituisce come nuova stringa, senza andare a modificare la stringa originale! (conteggio parte da 0), e sarà di tipo **STRING**:

[String.prototype.slice\(\) - JavaScript | MDN \(mozilla.org\)](#)

```
let str = 'Hello world! today im happy!';
console.log(str.slice(6));

```

è string

composto da un **indexStart** ex (4) indice del primo carattere da includere nella sottostringa ed è composto da un **indexEnd** (4, 10) indice del primo carattere da escludere nella sottostringa.

in caso non venisse inserito un indexEnd, si andrà ad estrarre fino al fondo della stringa (come nell'esempio precedente).

```
let str = 'Hello world! today im happy!';
console.log(str.slice(6, 21));
// 6 indexStart    21 indexEnd
```

È possibile utilizzare **indici negativi** per indicare la posizione dalla fine della stringa `.slice(-4)`, con un indice -4 andiamo ad estrarre gli ultimi 4 caratteri della stringa:

```
let str = 'Hello world! today im happy!';
console.log(str.slice(-6));
```

`.substring()` → è uno dei 2 metodi **molto simile a `.slice()`**, e fa l'identico lavoro dello slice

[String.prototype.substring\(\) - JavaScript | MDN](#) (mozilla.org)

```
let str = "Hello, world!";
let subStringed = console.log(str.substring(1,5));

```

la differenza principale è che il `substring()` **NON consente l'uso di indici negativi** per indicare posizioni all'indietro rispetto alla fine dell'array, e li tratta come se fosse 0:

```
let str = "Hello, world!";
let subStringed = console.log(str.substring(-6));

```

Hello, world!

`.replace()` → [String.prototype.replace\(\) - JavaScript | MDN \(mozilla.org\)](#)

-**.substr()** → Sebbene alcuni browser possano ancora supportarlo, potrebbe essere già stato rimosso dagli standard Web pertinenti, oggi risulta **DEPRECATO**

[String.prototype.substr\(\) - JavaScript | MDN \(mozilla.org\)](#)

restituisce una sottostringa della stringa originale, a partire dall'indice iniziale **indexStart(2)** incluso, e con **length** indica la lunghezza della sottostringa da estrarre:

```
string.substr(startIndex, length)
```

```
let str = "Hello, world!";
let subStringed = console.log(str.substr(7,5));
```

world

Nel caso non venisse specificato il **length**, verranno restituiti tutti i caratteri dalla posizione **startIndex** fino alla fine della stringa:

```
let str = "Hello, world!";
let subStringed = console.log(str.substr(7));
```

world!

NON consente l'uso di indici negativi per il length per indicare la lunghezza della sottostringa da estrarre, e li tratta come se fosse **0**:

```
let str = "Hello, world!";
let subStringed = console.log(str.substr(7,-4));
```

-**.toUpperCase() / .toLowerCase()** → trasforma la stringa in maiuscolo o minuscolo

[String.prototype.toUpperCase\(\) - JavaScript | MDN \(mozilla.org\)](#)

[String.prototype.toLowerCase\(\) - JavaScript | MDN \(mozilla.org\)](#)

-**.trim()** → rimuove gli spazi vuoti da entrambe le estremità di questa stringa e restituisce una nuova stringa, senza modificare la stringa originale, sarà di tipo **STRING**:

[String.prototype.trim\(\) - JavaScript | MDN \(mozilla.org\)](#)

```
const greeting = '      Hello world!    ';
console.log(greeting.trim());
```

Hello world!

Per restituire una nuova stringa con spazi vuoti tagliati da una sola estremità, utilizzare [trimStart\(\)](#) o [trimEnd\(\)](#).

STRING IMMUTABLE

LE STRINGHE IN JS **SONO IMMUTABILI** E NON SONO "MODIFICABILI", POSSIAMO MODIFICARE IL VALORE CREANDO E ASSEGANDOLO AD UNA NUOVA VARIABILE O RI-INIZIALIZZANDO LA STESSA VARIABILE:

```
// example 1
let str = 'hello';
str[0] = 'H'; // try to modify the string
console.log(str); // output: hello

// example 2
let originalString = 'hello';
let modifiedString = originalString.toUpperCase();
console.log(originalString); // Output: hello
console.log(modifiedString); // Output: HELLO
```

TEMPLATE STRINGS

Tramite ALT + 96, possiamo creare un apice strano che è sensitive quando si va a capo, e scrivere codice + moderno (non tutti i codici possono supportare queste specifiche su [JS 2015](#))

```
const message = `Can be
on multiple
lines`;
console.log(message);
```

Can be
on multiple
lines

REGULAR EXPRESSION

Le usiamo per cercare qualcosa all'interno di una stringa

VEDERE SLIDE [js-05b-regex](#), PER TUTTE LE COMBINAZIONI = **PATTERN (ex: a|b) + FLAG (ex: gm)**

FONTI → https://www.w3schools.com/js/js_regexp.asp

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

6 LEZIONE

INTRODUZIONE AGLI ARRAY

Un **array** è una sequenza di valori (ricordiamo un valore è ex → stringa, number undefind ...), come in C++ la sequenza parte dall'**indice 0**

possono contenere anche una sequenza di valori di **tipologia differente**:

```
// Let arrayName = [element0, element1, ...];
const rainbowColors = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet'];
let raceWinners = [33, 72, 64];
let myFavoriteThings = ['Broccoli', 60481, 'Love Actually'];
```

possiamo anche verificare la lunghezza del mio array tramite **.length**

```
console.log(rainbowColors.length);
```

- **accedere all'array** → per accedere agli array utilizziamo la cosiddetta "**bracket notation**" si riferisce al metodo di accesso agli elementi di un array utilizzando le []. Al loro interno inseriamo l'indice dell'elemento che si desidera accedere:

```
let arrayItem = arrayName[indexNum];
const rainbowColors = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet'];
let firstColor = rainbowColors[0];
let lastColor = rainbowColors[6];
```

Nel caso andassimo a mettere un indice non consono, verrà restituito come valore **undefined**

- **modificare un array** → per modificare un l'elemento di un array, anche qui utilizziamo anche qui la "**bracket notation**"

```
let myFavoriteThings = ['Broccoli', 60481, 'Love Actually'];
myFavoriteThings[0] = 'Celery Root';
```

Ma anche aggiungere degli elementi ad un array, sempre tramite la "**bracket notation**"

```
myFavoriteThings[4] = 'Playgrounds';
```

ma anche tramite il metodo **.push()** → aggiunge quell'elemento automaticamente nella posizione successiva all'ultima. **RITORNA LA NUOVA LUNGHEZZA DELL'ARRAY**

```
myFavoriteThings.push('Dancing');
```

- **loops con array e stringhe** → per loopare tutti gli elementi di un'array, usare "SEMPRE" un ciclo for:

```
const rainbowColors = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet'];
for (let i = 0; i < rainbowColors.length; i++) {
  console.log(rainbowColors[i]);
}
```

Ma possiamo anche loopare per uno un elemento di tipo **string: MEGLIO USARE charAt()**

```
const rainbowColorsLetters = 'ROYGBIV';
for (let i = 0; i < rainbowColorsLetters.length; i++) {
  console.log(rainbowColorsLetters[i]); // using [] is confusing; use charAt()
}
```

TEMPLATE LITERALS \${ } (parte stringhe)

Il \${ } è una sintassi speciale in JavaScript chiamata template literals o string interpolation.

questa sintassi consente di incorporare espressioni JavaScript all'interno di una stringa delimitata da backtick (` `), consentendo una formattazione più flessibile e leggibile delle stringhe:

```
let name = "John";
let greeting = `Hello, ${name}!`;
console.log(greeting); // Output: Hello, John!
```

METODI X ARRAY → https://www.w3schools.com/js/js_array_methods.asp

- .slice() → restituisce una copia superficiale di una porzione di un array in un nuovo oggetto array. Non modifica l'array originale ma ne crea uno nuovo.

```
const array = [1, 2, 3, 4, 5];
const newArray = array.slice(1);
console.log(newArray);
```

(4) [2, 3, 4, 5]

è composto da un indexStart ex (4) indice dal quale parte l'estrazione, e da un indexEnd (4, 10) indica l'indice prima del quale terminare l'estrazione.

in caso non venisse inserito un indexEnd, si andrà ad estrarre fino IN FONDO:

```
const array = [1, 2, 3, 4, 5];
const newArray = array.slice(1, 4); // il 4 è ESCLUSO
console.log(newArray);
```

(3) [2, 3, 4]

- `.splice()` → questo metodo modifica il contenuto di un array rimuovendo o sostituendo gli elementi esistenti e/o aggiungendo nuovi elementi all'interno dell'array.

[Array.prototype.splice\(\) - JavaScript | MDN \(mozilla.org\)](#)

È composto principalmente da 3 parametri:

- `indexStart` → è l'indice da cui iniziare a modificare l'array
- `deleteCount` → è il numero di elementi da rimuovere dall'array (se impostato su 0, nessun elemento viene rimosso).
Se viene omesso, o se il suo valore è maggiore o uguale al numero di elementi dopo la posizione specificata da , tutti gli elementi dopo l'`indexStart` della matrice verranno eliminati
- `item` → sono gli elementi da aggiungere all'array (0-N), iniziando dall'`indexStart`, verranno inseriti prima dell'`indexStart`

Esempi utilizzando `indexStart` e `deleteCount`:

```
const array = [1, 2, 3, 4, 5];
array.splice(3);
console.log(array);
```

▶ (3) [1, 2, 3]

```
const array = [1, 2, 3, 4, 5];
array.splice(1, 2);
console.log(array);
```

▶ (3) [1, 4, 5]

Esempi utilizzando `indexStart` e `deleteCount` e `items`:

```
const array = [1, 2, 3, 4, 5];
array.splice(2, 1, "ciao");
console.log(array);
```

▶ (5) [1, 2, 'ciao', 4, 5]

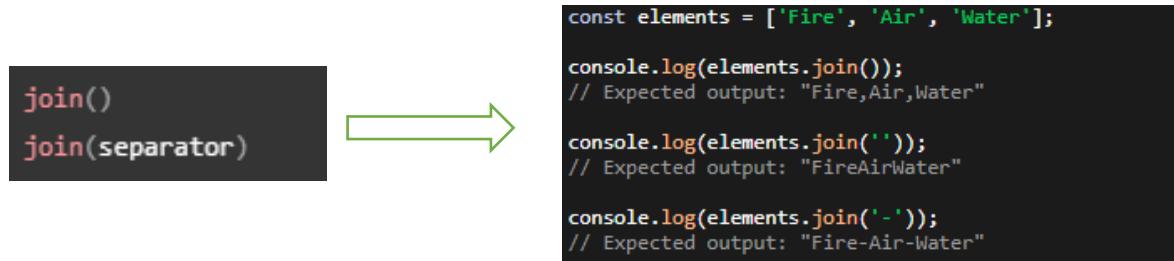
!! RESTITUISCE GLI ELEMENTI ELEMINATI ANCHE!! VEDERE MEGLIO [SU MDN](#) !!

PUO ESSERE UN ARRAY VUOTO (NESSUN ELEMENTO ELIMINATO)

PUO RESTITUIRE UN ARRAY CON GLI ELEMENTI ELIMINATI

- **.join()** → crea e restituisce una nuova stringa concatenando tutti gli elementi di questo array, separati da virgole o da una stringa di separazione specificata. Se l'array ha solo un articolo, quindi quell'articolo verrà restituito senza utilizzare il separatore.

[Array.prototype.join\(\) - JavaScript | MDN \(mozilla.org\)](#)



```
const elements = ['Fire', 'Air', 'Water'];

console.log(elements.join());
// Expected output: "Fire,Air,Water"

console.log(elements.join(''));
// Expected output: "FireAirWater"

console.log(elements.join('-'));
// Expected output: "Fire-Air-Water"
```

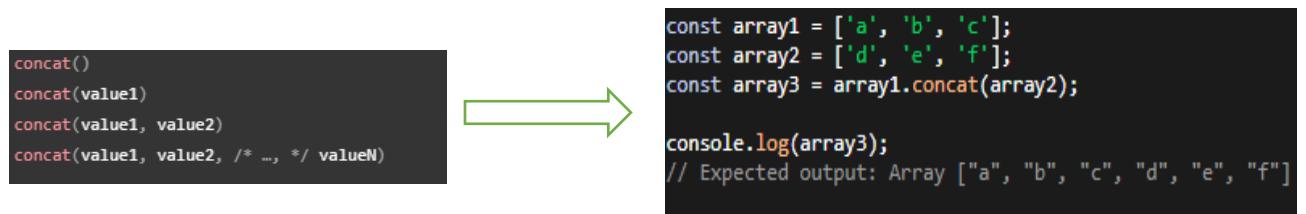
Converte in modo ricorsivo OGNI elemento, incluse altre matrici, in stringhe.

```
const matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9],
];

console.log(matrix.join()); // 1,2,3,4,5,6,7,8,9
console.log(matrix.join(";")); // 1,2,3;4,5,6;7,8,9
```

- **.concat()** → viene utilizzato per unire due o più array. Questo metodo non modifica gli array , ma restituisce un nuovo array.

[Array.prototype.concat\(\) - JavaScript | MDN \(mozilla.org\)](#)



```
const array1 = ['a', 'b', 'c'];
const array2 = ['d', 'e', 'f'];
const array3 = array1.concat(array2);

console.log(array3);
// Expected output: Array ["a", "b", "c", "d", "e", "f"]
```

- **.sort()** → riordina numericamente e alfabeticamente gli elementi (PRIMA NUMERI POI STRINGHE SE SONO MIXATI) **OCCHIO PER I NUMERI!**

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort

- **.find()** → SIMILE AL foreach

[Array.prototype.find\(\) - JavaScript | MDN \(mozilla.org\)](#)

.includes() → determina se un array include un certo valore tra le sue voci, come valore di ritorno è di tipo booleano, a seconda dei casi: “true” “false”.

È composto da un searchElement (valore da cercare) formIndex (Indice dove iniziare la ricerca)

[Array.prototype.includes\(\) - JavaScript | MDN \(mozilla.org\)](#)

```
includes(searchElement)  
includes(searchElement, fromIndex)
```



```
const array1 = [1, 2, 3];  
console.log(array1.includes(2));  
// Expected output: true  
  
const pets = ['cat', 'dog', 'bat'];  
console.log(pets.includes('cat'));  
// Expected output: true  
  
console.log(pets.includes('at'));  
// Expected output: false
```

7 LEZIONE

INTRODUZIONE AGLI OGGETTI (OBJECT)

È una variabile di tipo OBJECT, sono una collezione di **proprietà valori / metodi** in correlazione tra di loro:

```
const persona = {  
    nome: "Gabriele"  
}
```



andiamo a vedere nel dettaglio un esempio di un oggetto persona a cui sono associati dei dati al suo interno che hanno un senso logico tra di loro SEPARATI CON UN VIRGOLA (,):

```
const persona = {  
  
    // dei dati string numbers dentro l'oggetto  
    nome: "Gabriele",  
    cognome: "Speciale",  
    eta: 19,  
  
    // un ARRAY dentro l'oggetto  
    hobby: ["videogiochi", "cinema", "nuoto"],  
  
    // un OGGETTO dentro l'oggetto  
    indirizzo : {  
        via: "corso Garibaldi",  
        cap: "23000",  
        citta: "Torino"  
    },  
  
    // una FUNZIONE dentro l'oggetto  
    saluti: function()  
        console.log("ciao mondo!");  
    }  
}
```

un oggetto persona come in questo caso possiede sia delle proprietà che metodi:

PROPRIETA → cioè delle variabili: nome, cognome, eta, hobby, indirizzo.

La domanda è: queste proprietà appartengono alla persona?

METODI → sono le funzioni di particolari oggetti ex: salutati: function() è il metodo di persona

1.ACCEDERE AGLI OGGETTI | DOT NOTATION

Con questo modo andremo a “prendere” (accedere) alle PROPRIETA / METODI di un oggetto, tramite i punti singoli (.)

Andiamo ad utilizzarla quando dobbiamo fare qualcosa di **NON dinamico**

vediamo alcuni esempi usando l’oggetto persona di prima:

```
let myName = persona.nome;  
console.log(myName);
```



Gabriele

```
console.log(persona.eta);
```



19

```
let myHobby = persona.hobby[0];  
console.log(myHobby);
```



videogiochi

```
let myCap = persona.indirizzo.cap;  
console.log(myCap);
```



23000

```
persona.saluti()
```



ciao mondo!

2.ACCEDERE AGLI OGGETTI | BRACKET NOTATION

Con questo modo andremo a “prendere” (accedere) alle PROPRIETA / METODI di un oggetto, tramite le parentesi quadrate, e il contenuto con le doppi virgolette ([“ ”])

Andiamo ad utilizzarla quando dobbiamo fare qualcosa di **dinamico**.

vediamo che avrà lo stesso effetto rispetto alla dot notation:

```
let myHobby = persona["hobby"][0];  
console.log(myHobby);
```

E =

```
let myHobby = persona.hobby[0];  
console.log(myHobby);
```

```
let myCap = persona["indirizzo"]["cap"];  
console.log(myCap);
```

E =

```
let myCap = persona.indirizzo.cap;  
console.log(myCap);
```

QUANDO USARE DOT NOTATION | BRACKET NOTATION

```
/* returns an array containing the keys to an object, it also used  
for determinate the length of the object*/  
let arrayFromObj = Object.keys(cartForParty);  
  
console.log(arrayFromObj); // ['banana', 'handkerchief', 'Tshirt', 'apple', 'nalgene', 'proteinShake']  
  
/* Let's go to cycle all the keys contained by the arched array from the object */  
for (let i = 0; i < arrayFromObj.length; i++) {  
    cashOut += +cartForParty[arrayFromObj[i]]; // put the "+" to convert it into a number type  
}
```

AGGIORNARE UN OGGETTO

Per andare a modificare il nostro oggetto, possiamo usare sia la DOT NOTATION che la BRACKET [] NOTATION.

vediamo alcuni casi differenti su come possiamo aggiornare l'oggetto **persona** usato prima:

1. MODIFICARE VALORE DI PROPRIETA GIA ESISTENTI

!! IMPORTANTE !! → NON andiamo a modificare l'oggetto persona anche perché non si potrebbe dato che è una costante, ma andiamo a modificare il valore di una proprietà !!

```
persona.cognome = "Bianchi";  
console.log(persona.cognome);
```

 Bianchi

2. AGGIUNGERE NUOVE PROPRIETA ALL'OGGETTO

```
// AGGIUNGO proprietà all'oggetto persona  
persona.altezzaCM = 177;  
  
// assegno la nuova proprietà ad una variabile  
let myAltezza = persona.altezzaCM;  
console.log(myAltezza);
```

 177

3. ELIMINARE DELLE PROPRIETA ALL'OGGETTO → **delete**

```
// ELIMINO proprietà all'oggetto persona  
delete persona.eta;  
  
console.log(persona.eta);
```

 undefined

PAROLA CHIAVE “THIS”

Significa letteralmente **questo**, in cui andiamo a fare riferimento allo stesso oggetto che contiene questo metodo, vediamo un esempio:

```
let myCat = {  
    age: 8,  
    name: 'Cleo',  
    sayName: function() {  
        console.log('I am ' + this.name);  
    },  
};
```

OBJECT.KEYS() | OBJECT.VALUES() | OBJECT.ENTRIES()

metodo statico restituisce **un array di nomi di proprietà con chiave stringa** enumerabili di un **determinato oggetto**. Di seguito vediamo il suo funzionamento in diversi esempi

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys

```
// Simple array  
const arr = ["a", "b", "c"];  
console.log(Object.keys(arr)); // ['0', '1', '2']  
  
// Array-like object  
const obj = { 0: "a", 1: "b", 2: "c" };  
console.log(Object.keys(obj)); // ['0', '1', '2']  
  
// Array-like object with random key ordering  
const anObj = { 100: "a", 2: "b", 7: "c" };  
console.log(Object.keys(anObj)); // ['2', '7', '100']
```

Se hai bisogno dei **valori** delle proprietà → usa [Object.values\(\)](#)

Se sono necessari sia le **chiavi** che i **valori** della proprietà → utilizzare [Object.entries\(\)](#).

```
> Array [Array ["a", "somestring"], Array ["b", 42]]
```

[VEDERE MEGLIO L’ULTIMA PARTE DEGLI OBJECT METODI NUMBER e MATH](#)

COPYING VALUE OR BY REFERENCE

Al contrario di variabili come le stringhe e numeri, c'è una questione di cambio del valore per referenza, verso gli oggetti:

```
// copying by reference (objects)
let obj1 = { name: 'John' };
let obj2 = obj1; // obj2 is now a reference to obj1

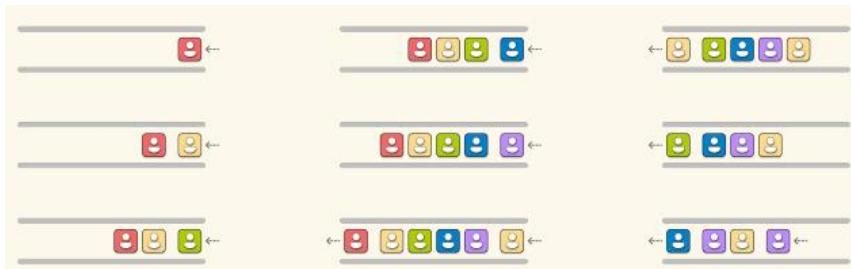
obj2.name = 'Jane'; // changing obj2 will also affect obj1

console.log(obj1.name); // output: Jane
console.log(obj2.name); // output: Jane
```

8 LEZIONE → [slide prof](#)

FIFO

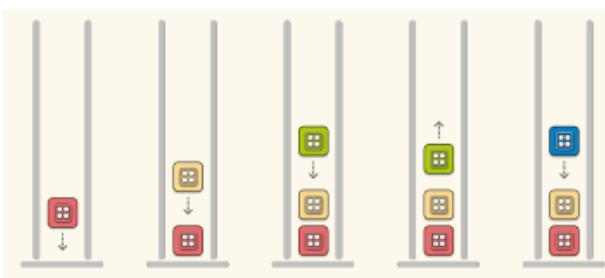
Il primo ad entrare è il primo ad uscire:



Priority queue → agevolazioni su determinati processi, ad esempio se abbiamo una fila loro salteranno tutta la fila

LIFO

Ogni processo viene messo uno sopra l'altro, quindi il primo ad entrare è l'ultimo ad uscire



PUSH → INSERIRE ALLA FINE

UNSHIFT → INSERIRE ALL'INIZIO

POP → TOGLIERE ALLA FINE

SHIFT → TOGLIERE ALL'INIZIO

EVENT LOOP

Il ciclo di eventi consente a JavaScript di gestire operazioni asincrone (**setTimeout()** **setInterval()**) e fornire l'illusione di essere multithreading nonostante sia single-thread:

con codice asincrono, le istruzioni NON vengono necessariamente eseguite nell'ordine in cui appaiono.

JS può effettuare un'operazione alla volta

Codice sincrono / codice asincrono

codice asincrono --> il codice viene eseguito in momenti differenti da diversi fattori `setTimeOut()`, e non per forza in ordine di riga di codice

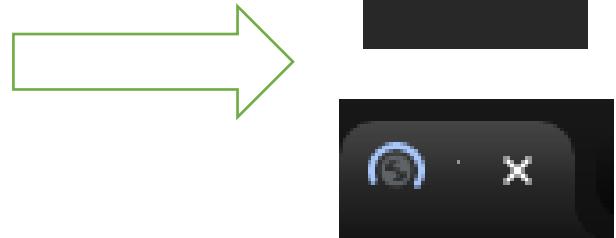
codice sincrono --> SONO FUNZIONI NORMALI SENZA TIMEOUT, VERRANNO ESEGUITE PRIMA DI ESSE, FINO A QUANDO NON VERRANNO ESEGUITE (POTREBBERO RIMANERE BLOCCATE), NON VERRANNO ESEGUITI IL CODICE ASINCRONO

```
// creazioni funzioni
let f = () => console.log('first');
let h = () => console.log('hi');
let b = () => console.log('bye');

// FUNZIONI ASINCRONE (schedulazione)
setTimeout(h, 0);
setTimeout(b, 4000);
setTimeout(b, 10000);

// LA FUNZIONE F è SINCRONA
f();

// LA FUNZIONE A è SINCRONA ma resterà bloccato qua
// f h b non verranno eseguite!
let a = () => {
  while (true)
    a;
}
a();
```



9 LEZIONE → timing and interval slide prof

Possiamo decidere di non eseguire una funzione immediatamente, ma in un determinato momento (**TIMING**), si può applicare tramite 2 modi:

-**setTimeout()** → consente di eseguire una funzione una volta dopo un intervallo di tempo, dopodichè, non fa più nulla, muore.

-**setInterval()** → consente di eseguire regolarmente una funzione con un intervallo di tempo specificato.

1. setTimeout syntax

```
setTimeout(function, milliseconds, param1, param2, ...)
```

```
setTimeout(funzioneProva, 4000, 6, "ciao")
```

Andiamo ad analizzare la sintassi dei singoli parametri:

-**function** → è **OBBLIGATORIO!** È la funzione di callback che verrà eseguita dopo un tempo di attesa:

```
function sayHi() {
  console.log('Hello');
}

setTimeout(sayHi, 1000);
```



- **setTimeout con funzione anonima** : usa e getta (la usi solo quella volta!):

```
// passiamo alla funzione anonima un parametro
setTimeout(function prova(str){
  console.log(str);
}, 2000, "ciao!");
```

-**milliseconds** → è **OPZIONALE!** È il numero di millisecondi di attesa prima di eseguire il codice, se omesso, viene utilizzato il **valore 0**

-**params** → sono **OPZIONALI!** Parametri aggiuntivi da passare alla funzione. Non supportato nei browser molto vecchi

clearTimeout

quando vogliamo che non succeda quella cosa nella funzione, quindi cancella la sua esecuzione

```
let timerId = setTimeout(function () {
  console.log('never happens');
}, 1000);
clearTimeout(timerId);
```



```
PROVA ESERCIZI\JS> node main
PROVA ESERCIZI\JS> []
```



2.setInterval syntax

```
setInterval(function, milliseconds, param1, param2, ...)
setInterval(funzioneProva, 3000, "zaino", 4);
```

come si nota la sintassi è uguale alla sintassi del **setTimeout**, ovviamente cambia il funzionamento, la funzione verrà eseguita ogni tot secondi:

```
function sayHi() {
  console.log('Hello');
}

setInterval(sayHi, 1000);
```



```
PS C:\Users\ICT
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```



clearInterval

idem del **clearTimeOut**

Date and Time → https://www.w3schools.com/js/js_dates.asp

La sintassi della data può essere composta in 9 maniere differenti, tramite **new Date ()**

!Va seguita la gerarchia dell'ordine delle date!!

(year, month, day, hour, minute, second, millisecond)

```
new Date()
new Date(date string)

new Date(year,month)
new Date(year,month,day)
new Date(year,month,day,hours)
new Date(year,month,day,hours,minutes)
new Date(year,month,day,hours,minutes,seconds)
new Date(year,month,day,hours,minutes,seconds,ms)

new Date(milliseconds)
```

-**new Date()** → creates a date object with the current date and time:

```
const live = new Date();
console.log(live);
```



Fri May 17 2024 09:15:04 GMT+0200
(Ora legale dell'Europa centrale)

-**new Date(date string)** → creates a date object from a date string:

```
const date = new Date("2004 10 7 11:13:00");
console.log(date);
```



Thu Oct 07 2004 11:13:00 GMT+0200 (Ora legale dell'Europa centrale)

```
const live = new Date("2004-10-07");
console.log(live);
```



Thu Oct 07 2004 02:00:00 GMT+0200
(Ora legale dell'Europa centrale)

-**`new Date(year, month, day...)`** → creates a date object with a specified date and time:

year, month, day, hour, minute, second, and millisecond

in talcaso venisse omesso il MESE e inserito solo 1 parametro, quel parametro varrà come millisecondi

il mese parte da 0! come gli array! → jan:0 dec:11

```
const live = new Date(2004, 10, 7, 8, 57, 30, 4000);
```



```
Sun Nov 07 2004 08:57:34 main.js  
GMT+0100 (Ora standard dell'Europa  
centrale)
```

-**`new Date(milliseconds)`** → creates a new date object as milliseconds + / - zero time:

zero time → parte sempre da 01 January 1970

```
const date = new Date(1000); //1000 = 1 sec  
console.log(date);
```



```
Thu Jan 01 1970 01:00:01 GMT+0100 (Ora standard  
dell'Europa centrale)
```

Date Methods

Anche per le date esistono dei metodi, tra cui:

-**`.get (metodo di accesso)`** → dato che non possiamo estrarre dei singoli valori da un date object, tramite il metodo `.getX` è possibile farlo.

in questi esempi andiamo ad estrarre valori della **!DATA DI ESEMPIO!** (2024/5/18 (sabato), 11:59:50)

ANNO (4 CIFRE)

```
// data e orario attuale  
const date = new Date();  
  
// estraiamo solo l'anno!  
const year = date.getFullYear();  
console.log(year);
```



2024

MESE (0-11)

```
// data e orario attuale  
const date = new Date();  
  
// estraiamo solo il mese! (4 = maggio)  
const month = date.getMonth();  
console.log(month);
```



4

GIORNO DEL MESE (1-31)

```
// data e orario attuale  
const date = new Date();  
  
// estraiamo solo il giorno!  
const day = date.getDate();  
console.log(day);
```



18

ORA (0-23)

```
// data e orario attuale
const date = new Date();
// estraiamo solo l'ora!
const hour = date.getHours();
console.log(hour);
```



11

MINUTI (0-59)

```
// data e orario attuale
const date = new Date();
// estraiamo solo i minuti!
const min = date.getMinutes();
console.log(min);
```



59

SECONDI (0-59)

```
// data e orario attuale
const date = new Date();
// estraiamo solo i secondi!
const sec = date.getSeconds();
console.log(sec);
```



50

MILLISECONDI (0-999)

```
// data e orario attuale
const date = new Date();
// estraiamo solo i millisecondi!
const millisec = date.getMilliseconds();
console.log(millisec);
```



511

GIORNO DELLA SETTIMANA LUNEDI MARTEDI... (0-6), si parte dalla DOMENICA

```
// data e orario attuale
const date = new Date();
// estraiamo solo il giorno della settimana! (0-6)
// 0 = domenica 1 = lunedì ...
const dayOfWeek = date.getDay();
console.log(dayOfWeek);
```



6

- **.set (metodo di impostazioni)** → tramite questo metodo, possiamo andare a settare dei valori alla date object (*year, month, day, TUTTO...*), andando volendo anche a modificare una new Date()

```
// data e orario attuale  
const date = new Date(2024, 6, 31, 17, 52, 34);  
  
// andiamo a settare l'anno alla costante "date"  
date.setFullYear(2014);  
  
const year = date.getFullYear();  
console.log(year);
```



2014

LEZIONE 10 → dom slide prof

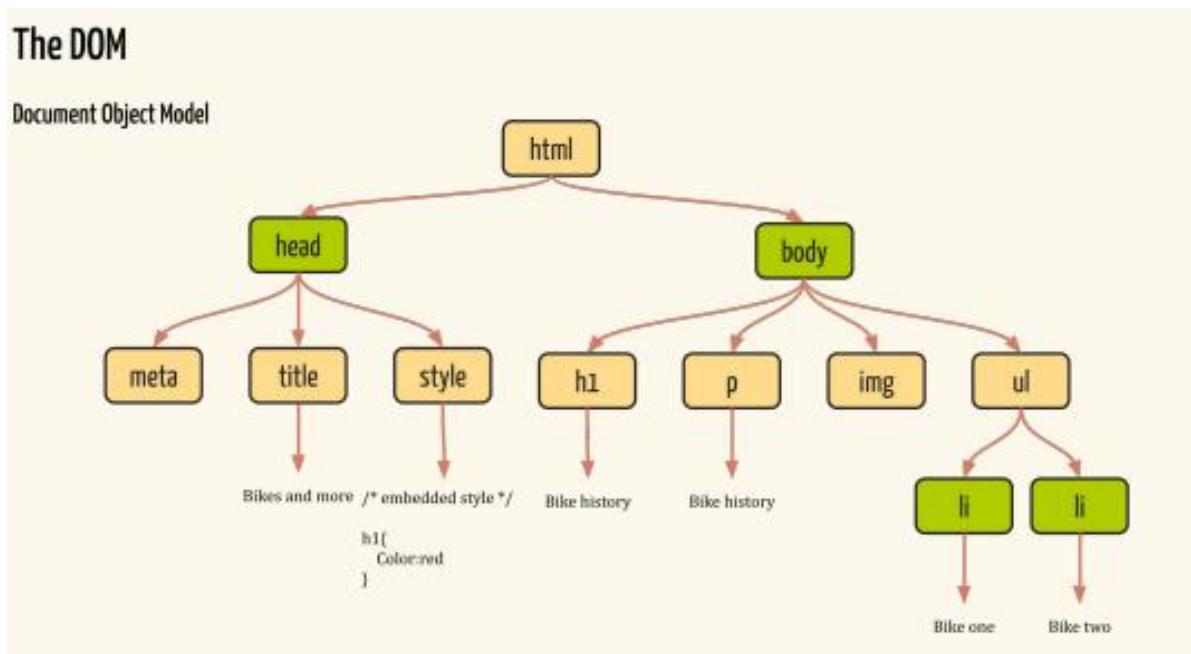
introduzione al DOM → Document Object Model

Fornisce una MAPPA STRUTTURATA del nostro documento:

indica LO STATO DELLA PAGINA IN UN MOMENTO PRECISO, e definisce in che modo i programmi possono accedere e manipolare il suo contenuto, struttura e stile.

Permette ai linguaggi di programmazione come JavaScript di manipolare la struttura, il contenuto e lo stile del documento in modo dinamico.

HA UNA STRUTTURA AD ALBERO:



HTML → È il nodo principale (PADRE) che rappresenta l'intero documento HTML.

L'oggetto documento è **disponibile globalmente** nel browser. Ti consente di accedere e manipolare il DOM della pagina web corrente, di standard seguiamo questi passaggi:

1. **INDIVIDUARE IL NODO** che si vuole cambiare usando uno dei **metodi di accesso**

2. **MEMORIZZARE IL NODO** all'interno di una variabile e usarla come esso

3. **MANIPOLARE IL DOM**:

- Cambia i suoi attributi
- Modifica i suoi stili
- Dagli un nuovo innerHTML / textcontent
- Aggiungi nuovi nodi ad esso

accedere al DOM && return values

-Document.getElementById() → Ritorna un **singolo** valore, cioè il **primo elemento** con l'**ID** specificato.

```
let primaLista = document.getElementById("hobby-list");
```

▶ ul#hobby-list

-Document.getElementsByClassName() → Ritorna + valori, un array like (**HTMLCollection**) di tutti gli elementi con **la classe specificata**.

```
let primaLista = document.getElementsByClassName("hobby");
```

```
▼ HTMLCollection(2) [li.hobby, li.hobby]
  ▶ 0: li.hobby
  ▶ 1: li.hobby
  length: 2
  ▶ [[Prototype]]: HTMLCollection
```

-Document.getElementsByTagName() → Ritorna + valori, un array like (**HTMLCollection**) di tutti gli elementi con il nome **del tag specificato**.

```
let primaLista = document.getElementsByTagName("ul");
```

```
▼ HTMLCollection [ul#hobby-list, hobby-list: ul#hobby-list] ⓘ
  ▶ 0: ul#hobby-list
  ▶ hobby-list: ul#hobby-list
  length: 1
  ▶ [[Prototype]]: HTMLCollection
```

-**Document.querySelector()** → Ritorna un **singolo** valore, cioè il primo elemento che corrisponde al selettore specificato. Si possono prendere anche proprietà dal CSS come gli attributi!!

```
let primaLista = document.querySelector("ul li");
```

```
'<li class="hobby">
  ::marker
  "Playing the banjo"
</li>
```

-**Document.querySelectorAll()** → Ritorna + valori, un array like (NodeList) di tutti gli elementi che corrispondono al selettore specificato. Si possono prendere anche proprietà dal CSS come gli attributi!!

```
let primaLista = document.querySelectorAll("ul li");
```

```
▼ NodeList(2) [li.hobby, li.hobby] ⓘ
  ► 0: li.hobby
  ► 1: li.hobby
  length: 2
  ► [[Prototype]]: NodeList
```

array vs array like → [esempi di utilizzi degli array like VS array](#)

Le differenze principale di un array like (HTMLCollection / NodeList) e un array normale:

-**ARRAY** → Un array in JavaScript è un tipo di dato utilizzato per memorizzare una lista ordinata di valori.

Gli array possiedono: **Proprietà e Metodi degli Array**: Gli array in JavaScript hanno accesso a metodi come **push**, **pop**, **shift**, **unshift**, **splice**, **map**, **filter**, **reduce...**

-**ARRAY LIKE** → è un oggetto in cui:

puoi solamente utilizzare la **proprietà .length**, che permettono l'accesso ai suoi elementi come farebbe un array.

NON ha accesso ai metodi degli array perché non eredita da **Array.prototype**, se li si vuole usare, bisogna convertire l'array like in un array vero con → **Array.from()**

HTMLCollection vs NodeList

Entrambi **sono collezioni di nodi DOM**, ma ci sono differenze significative nelle loro proprietà e nei metodi disponibili:

1. HTMLCollection → [HTMLCollection - API Web | MDN \(mozilla.org\)](#)

- **TIPOLOGIA DI ELEMENTI** → Un **HTMLCollection** contiene SOLO elementi HTML. Non può contenere altri tipi di nodi, come testi o commenti.

- **PROPRIETA "length"** → Possiede una proprietà **length** che indica il numero di elementi nella collezione.

- **ACCESSO PER NAME / ID** → Oltre a poter accedere agli elementi tramite **indici numerici**, è possibile accedere agli elementi di un **HTMLCollection** tramite i loro attributi `name` o `id` (se presenti).

- **AGGIORNAMENTO DINAMICO** → Un **HTMLCollection** è **"live"**, il che significa che si aggiorna automaticamente per riflettere qualsiasi modifica al DOM. Ad esempio, se aggiungi o rimuovi un elemento dalla pagina, **HTMLCollection** verrà aggiornata di conseguenza.

HTMLCollection ha metodi limitati rispetto a **NodeList**. Puoi accedere agli elementi per indice, name, o ID, ma non ha metodi di array come `forEach`.

Esempi di `HTMLCollection`:

- `document.getElementsByTagName()`
- `document.getElementsByClassName()`

```
let elements = document.getElementsByClassName('example');
console.log(elements.length); // Numero di elementi con la classe 'example'
console.log(elements[0]); // Primo elemento con la classe 'example'
console.log(elements.namedItem('example-id')); // Elemento con id 'example-id'
```

2. NodeList → [NodeList - Web APIs | MDN \(mozilla.org\)](#)

- **TIPOLOGIA DI ELEMENTI** → Una **NodeList** può contenere qualsiasi tipo di nodo DOM, inclusi elementi, nodi di testo, commenti ...

- **PROPRIETA "length"** → Possiede una proprietà **length** che indica il numero di elementi nella collezione.

- **ACCESSO PER INDICE** → Gli elementi di una **NodeList** possono essere accessibili tramite indici numerici. NON è possibile accedere tramite `name` o `id`.

- **AGGIORNAMENTO DINAMICO vs STATICO** → Una **NodeList** può essere sia "**live**" che "**static**":

1. Le **NodeList** restituite da `querySelectorAll()` sono statiche! e non si aggiornano automaticamente quando il **DOM** cambia.

2. Le **NodeList** restituite da `childNodes()` sono live.

Le **NodeList** moderne supportano il metodo `forEach`, che permette di iterare facilmente sugli elementi. Tuttavia, non supportano tutti i metodi degli array a meno che non vengano convertite in array veri e propri tramite **Array.from()**

Esempi di `NodeList`:

- `document.querySelectorAll()`
- `element.childNodes`

```
let nodes = document.querySelectorAll('.example');
console.log(nodes.length); // Numero di elementi con la classe 'example'
nodes.forEach(node => console.log(node)); // Itera su ogni nodo con la classe 'example'

let childNodes = document.body.childNodes;
console.log(childNodes.length); // Numero di nodi figli del body
```

manipolazione del DOM

1. manipolazione degli attributi

vediamo un esempio di manipolazione di un elemento:

1. Il file HTML contiene un'immagine con un **id my-cat** e una sorgente “**src**” che punta a una foto casuale:

```

```

2. andiamo a selezionare il nodo DOM tramite `document.getElementById('my-cat')` per selezionare l'elemento con l'id **my-cat**.

```
// Seleziona il nodo DOM
let catImage = document.getElementById('my-cat');
```

3. accediamo e cambiamo gli attributi `` di un nodo DOM usando la dot notation

catImage.src → accede all'attributo src dell'immagine, che contiene l'URL dell'immagine attualmente visualizzata.

```
// Accedi e cambia gli attributi di un nodo DOM usando la notazione a punti
// Cambia l'attributo src di un'immagine
let oldImageSource = catImage.src;
catImage.src = 'https://picsum.photos/300/200';
```

4. andiamo ad aggiungere una classe CSS “portrait” all' ``

```
// Cambia la className del nodo DOM
catImage.className = 'portrait';
```

2. manipolazione dello style

-bad example ☹

-good way ☺

3. manipolazione del contenuto

creazione DOM Nodes



LEZIONE 11 → events & listeners slide prof

introduzione agli Events

gli **event listeners** vengono utilizzati per eseguire del codice in risposta a eventi specifici che accadono nel browser →(come il clic di un pulsante, il caricamento di una pagina, o il passaggio del mouse su un elemento ...)

PERCHE UTILIZZARLI? Sono utili in vari casistiche come:

- Form validation and processing
- Interactive slideshows
- Games
- Single-page webapps
- Anything that involves user interaction

Aggiungere degli Events Listeners → gioca con event

La sintassi generale per aggiungere un listener di eventi a un nodo del DOM è la seguente:

[EventTarget: addEventListener\(\) method - Web APIs | MDN \(mozilla.org\)](#)

```
domNode.addEventListener(eventType, eventListener, useCapture);
```

- **domNode** → l'elemento del DOM su cui vogliamo ascoltare l'evento.

- **eventType** → il tipo di evento che vogliamo ascoltare (ad esempio, 'click', 'mouseover', ecc...).

[EventTarget: addEventListener\(\) method - Web APIs | MDN \(mozilla.org\)](#)

- **eventListener** → La funzione che verrà chiamata quando l'evento si verifica.

[EventTarget: addEventListener\(\) method - Web APIs | MDN \(mozilla.org\)](#)

- **useCapture** → un parametro opzionale che specifica se l'evento deve essere catturato o meno (per la maggior parte dei casi, viene impostato su **false**) quindi un valore booleano.

Può essere impostato anche un oggetto di configurazione

[EventTarget: addEventListener\(\) method - Web APIs | MDN \(mozilla.org\)](#)

Partiamo da un esempio di base in cui aggiungiamo un event listener a un pulsante (**button**) per gestire l'evento '**click**' , quando verrà cliccato apparirà sul browser un alert sopra

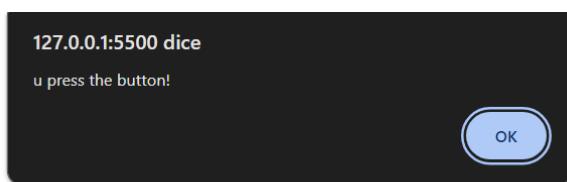
1. Creazione di un button nell'HTML

```
<body>
  <button id="mioBottone">clICCami</button>
```

2. aggiungiamo un event listener in JS al button → per gestire l'evento '**click**'

```
document.getElementById("onClick").addEventListener("click", function () {
  alert("u press the button!");
});
```

3. clicchiamo il bottone e vediamo il messaggio sul browser (alert)



Event Listener VS Event Handler

EVENT LISTENER → (“ascoltatore di eventi”) È il meccanismo utilizzato per ascoltare e gestire eventi su elementi del DOM

è una funzione che rimane in attesa , che succeda un certo evento su un elemento del DOM (come un bottone o un campo di testo)

.quando l'evento accade, l'event listener "scatta" e attiva una funzione (event handler):

EVENT HANDLER → (“gestore di eventi”) è la funzione che viene eseguita quando l'evento si verifica. In altre parole, un event handler è una funzione di callback associata a un evento specifico, Può essere scritto:

1. separatamente e passato come riferimento all'event listener

2. può essere definito direttamente all'interno della chiamata all'event listener

separatamente

```
// Selettore del pulsante
let button = document.getElementById('myButton');

// Definizione dell'event handler
function showMessage(event) {
  console.log('Button clicked!');
}

// Aggiunta dell'event listener
button.addEventListener('click', showMessage);
```

direttamente

```
// Selettore del pulsante
let button = document.getElementById('myButton');

// Aggiunta dell'event listener con l'event handler definito direttamente
button.addEventListener('click', function (event) {
  console.log('Button clicked!');
});
```

Event Types

il **tipo di evento** che vogliamo ascoltare che rappresenta un'occorrenza del DOM, si distinguono in diverse categorie, tra le + importanti abbiamo:

-**MOUSE EVENTS** → tra i +importanti abbiamo: ***mousedown, mouseup, click, dblclick, mousemove, mouseover, mousewheel, mouseout, contextmenu***

[MouseEvent - Web APIs | MDN \(mozilla.org\)](#)

-**TOUCH EVENTS** → tra i +importanti abbiamo: ***touchstart, touchmove, touchend, touchcancel***

[TouchEvent - Web APIs | MDN \(mozilla.org\)](#)

-**KEYBOARD EVENTS** → tra i +importanti abbiamo: ***keydown, keypress, keyup***

[KeyboardEvent - Web APIs | MDN \(mozilla.org\)](#)

-**FORM EVENTS** → tra i +importanti abbiamo: ***focus, blur, change, submit***

-**WINDOW EVENTS** → tra i +importanti abbiamo: ***scroll, resize, hashchange, load, unload***

LEZIONE 12 → json & ajax slide prof + appunti note

Json

ajax

ecco una breve introduzione su cosa e qual è il suo funzionamento:

AJAX e Fetch API: Una Spiegazione Dettagliata

AJAX (Asynchronous JavaScript and XML)

AJAX è una tecnica che permette di aggiornare parti di una pagina web senza ricaricarla completamente. Con AJAX, è possibile inviare e ricevere dati dal server in modo asincrono, ovvero senza interrompere l'interazione con la pagina web.

Funzionamento di una Richiesta AJAX

1. **Un evento accade in una pagina web:** Questo potrebbe essere il caricamento della pagina o il clic di un pulsante.
2. **Un oggetto `XMLHttpRequest` viene creato da JavaScript:** Questo oggetto è fondamentale per effettuare richieste HTTP.
3. **L'oggetto `XMLHttpRequest` invia una richiesta a un server web:** La richiesta può essere di vari tipi, come `GET`, `POST`, `PUT`, `DELETE`, ecc.
4. **Il server elabora la richiesta e risponde:** La risposta del server può contenere dati in vari formati, come JSON, XML, HTML, testo, ecc.
5. **La risposta viene letta da JavaScript:** Una volta ricevuta la risposta, JavaScript può elaborarla e aggiornare dinamicamente il contenuto della pagina web.

LEZIONE 13 → [in depth pdf prof + appunti note](#)

.map array

Per altri esempi + dettagliati con diversi metodi di comparazione, vedere gli esercizi, un esempio su come si usa map:

```
// .map() = accepts a callback and applies that function
//           to each element of an array, then return a new array

const students = ["Spongebob", "Patrick", "Squidward", "Sandy"];
const studentsUpper = students.map(upperCase);

function upperCase(element){
    return element.toUpperCase();
}
```

SPIEGAZIONE CODICE ASINCRONO CHAT

COSA SIGNIFICA CODICE ASINCRONO

Il codice asincrono consente di eseguire operazioni senza bloccare il flusso principale del programma. Questo è particolarmente utile in attività che richiedono tempo, come:

- Lettura/scrittura di file.
- Chiamate a database.
- Richieste HTTP.
- Operazioni basate su timer.

In JavaScript, il codice asincrono è fondamentale perché il linguaggio utilizza un modello **single-threaded** basato su un **event loop**.

SPIEGAZIONE DELL'EVENT LOOP

L'EVENT LOOP

1. **Thread principale**: gestisce il codice sincrono (istruzioni che vengono eseguite in sequenza).
2. **Coda di eventi**: raccoglie operazioni asincrone completate che sono pronte per essere elaborate.
3. **Event Loop**: controlla se il **thread principale** è libero per eseguire operazioni dalla coda.

Ad esempio, quando fai una richiesta HTTP con codice asincrono:

1. La richiesta parte.
2. Il **thread principale** prosegue l'esecuzione.
3. Quando la richiesta HTTP è completata, il **callback** associato viene messo nella coda.
4. L'**event loop** esegue il **callback** quando il **thread principale** è disponibile.

CALLBACK FUNCTIONS

Cos'è una callback?

Una callback è una funzione che viene passata come argomento a un'altra funzione e viene chiamata quando un'operazione è completata.

Esempio di callback:

javascript

 Copia codice

```
function saluta(nome, callback) {
    console.log(`Ciao, ${nome}!`);
    callback(); // Chiamo la funzione passata come argomento
}

function salutoFinale() {
    console.log("È stato un piacere conoscerti.");
}

saluta("Mario", salutoFinale);
```

Qui, salutoFinale è una **callback** eseguita dopo il messaggio iniziale.

Callback con codice asincrono

Le callback diventano essenziali nel codice asincrono. Per esempio:

javascript

 Copia codice

```
setTimeout(() => {
    console.log("Eseguito dopo 2 secondi!");
}, 2000);
```

CALLBACK HELL

CALLBACK HELL

Quando molte operazioni asincrone vengono annidate una dentro l'altra, il codice diventa difficile da leggere e mantenere: questo è noto come Callback Hell.

Esempio di Callback Hell:

javascript

Copia codice

```
setTimeout(() => {
    console.log("Step 1");
    setTimeout(() => {
        console.log("Step 2");
        setTimeout(() => {
            console.log("Step 3");
        }, 1000);
    }, 1000);
}, 1000);
```

- Problemi principali:
 - **Annidamento profondo:** è difficile capire la logica del codice.
 - **Manutenzione complessa:** modificare o aggiungere nuovi passaggi diventa complicato.

PROMISE

Per affrontare i problemi del Callback Hell, JavaScript ha introdotto le **Promise** (ECMAScript 6). Una **Promise** rappresenta un'operazione che non è ancora completata, ma che lo sarà in futuro.

Stati di una Promise:

1. Pending: in attesa di completamento.
2. Fulfilled: completata con successo.
3. Rejected: fallita con un errore.

Esempio di Promise:

```
javascript Copia codice  
  
const promessa = new Promise((resolve, reject) => {  
    let successo = true;  
  
    if (successo) {  
        resolve("Operazione completata con successo!");  
    } else {  
        reject("Errore durante l'operazione.");  
    }  
});  
  
promessa  
    .then((risultato) => {  
        console.log(risultato); // "Operazione completata con successo!"  
    })  
    .catch((errore) => {  
        console.error(errore);  
    });
```

Vantaggi delle Promise:

1. Migliore leggibilità rispetto alle callback.
2. Permettono di concatenare più operazioni con `.then()`.
3. Gestione centralizzata degli errori con `.catch()`.

Promises in JS

Ciao! Le *Promises* in JavaScript sono uno strumento fondamentale per gestire operazioni asincrone, cioè operazioni che non avvengono immediatamente ma che richiedono del tempo per completarsi, come per esempio una richiesta HTTP a un server, il caricamento di file, o un'interazione con un database.

A cosa servono le Promises?

Le Promises sono utili per evitare di usare *callback* annidate, che possono rendere il codice difficile da leggere e mantenere (fenomeno noto come *callback hell*). Con le Promises, puoi scrivere codice asincrono in modo più lineare e leggibile, gestendo facilmente i casi di successo e di errore.

Come funzionano le Promises?

Una Promise è come una "promessa" che qualcosa verrà completato in futuro. Ha tre stati principali:

1. **Pending** (In attesa) – La Promise è in corso e non è ancora completata.
2. **Fulfilled** (Completata con successo) – L'operazione asincrona è riuscita, e la Promise restituisce un risultato.
3. **Rejected** (Rifiutata) – L'operazione asincrona è fallita, e la Promise restituisce un errore.

Una volta creata una Promise, puoi usare i metodi `.then()` e `.catch()` per gestire i risultati:

- `.then()` viene eseguito quando la Promise viene risolta con successo.
- `.catch()` viene eseguito quando la Promise viene rifiutata con un errore.

PROMISE PER RISOLVERE IL CALLBACK HELL

Promise per risolvere il Callback Hell

Esempio riscritto usando Promise:

javascript

Copia codice

```
const step1 = () => {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log("Step 1");
            resolve();
        }, 1000);
    });
};

const step2 = () => {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log("Step 2");
            resolve();
        }, 1000);
    });
};

const step3 = () => {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log("Step 3");
            resolve();
        }, 1000);
    });
};

// Concatenazione
step1()
    .then(step2)
    .then(step3)
    .catch((errore) => {
        console.error("Errore:", errore);
    });
}
```

ASYNC E AWAIT

ASYNC/AWAIT

Le Promise migliorano la leggibilità, ma possono diventare verbose in caso di molte operazioni concatenate. **Async/Await**, introdotti in ECMAScript 2017, rendono il codice asincrono ancora più simile a quello sincrono.

Come funziona `async/await`?

1. Una funzione dichiarata con `async` restituisce automaticamente una Promise.
2. Il comando `await` può essere usato **solo dentro funzioni `async`** per aspettare che una Promise venga risolta.

Esempio semplice di `async/await`:

javascript

 Copia codice

```
async function esegui() {
  try {
    const risultato = await new Promise((resolve) => {
      setTimeout(() => resolve("Completato!"), 2000);
    });
    console.log(risultato); // "Completato!"
  } catch (errore) {
    console.error("Errore:", errore);
  }
}

esegui();
```

Call back function asincrona esempio in classe

```
//me
Complexity is 4 Everything is cool! | Codeium: Refactor | Explain | X
function sumTwoValues(x, y, callback){ █
  if(typeof callback !== "function"){
    return;
  }
  setTimeout(()=> {
    let sum = x + y;
    callback(sum);
  }, 3000);
}

//client
sumTwoValues(2, 2, result => {
  //do something with result
  console.log(result);
});

// se sumtwovalues fosse sincrono avrei fatto cosi'
let result = sumTwoValues(3, 8);
console.log(result);
```

Esempio problema del codice asincrono

Mettiamo che stiamo facendo una richiesta fetch api per ricevere il meteo del tempo (è una operazione asincrona!).

se successivamente mettiamo un console.log() che fa parte del percorso SINCRONO, risulterà undefined.

perché non abbiamo ancora ottenuto il meteo, dato che la funzione che prende il meteo è asincrona, NON BLOCCERA il codice e la sua esecuzione e andrà avanti con essa! Ignorando questo enorme problema!



```
let weather = getWeather()
console.log(weather) //undefined

function getWeather() {
  callWeatherAPI(() => {
    return 'Sunny' //this is ignored
  })
  return undefined
}
```

Se il codice fosse sincrono...



```
let weather = getWeather()
console.log(weather) //'Sunny'

function getWeather() {
  return 'Sunny'
}
```