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** 🡪 si occupa della logica che permette di compiere delle azioni dinamiche, può inviare delle richieste e rimandare indietro…

un elemento di JavaScrpit 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** 🡪



COLLEGARE UN FILE JAVASCRIPT A HTML

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



**STRUTTURA** delle cartelle e file:



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.



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 implementatare delle funzionalità complesse nelle nostre pagine, tra le principali abbiamo:

* aggiornare il conenuto del sito con nuovi contenuti.
* disegni, animazioni, gallerie di immagini…
* tenere traccia degli utenti tramite i **COOKIE.**
* Elementi intereattivi (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 programmatori pensare nel linguaggio di programmazione. Anch’essi, richiedono anche la traduzione in linguaggio macchina prima dell'esecuzione.

Traduzione viene eseguita da un c**ompilatore** o da un **interprete**.



LINGUAGGIO COMPILATO 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 COMPILATO** 🡪 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 indipendetemente** **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.
* Modifca 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 COMPILATO E INTERPRETATO 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 (XD) di come funziona il **linguaggio compliato** e il **linguaggio interpreatato**

Importanza di testare e ritestare su vari browser il nostro codice per vedere se funziona correttamente su tutti quanti.

-**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:

***Valori Operatori Espressioni Parole chiave Commenti***

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 succesivamente, 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

-**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 **.lenght**, possiamo vedere da quanti caratteri è composta la nostra stringa, se vogliamo muoverci sul singolo carattere, usiamo gli **array[]**



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



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



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



- **;** 🡪 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

 🡪 //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**

****

**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)](https://www.w3schools.com/js/js_math.asp)

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>



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

**-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!



**-Circular dependencies** 🡪 Viene chiamata **Recursione, 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>

[](https://medium.com/@williambdale/recursion-the-pros-and-cons-76d32d75973a)

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

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



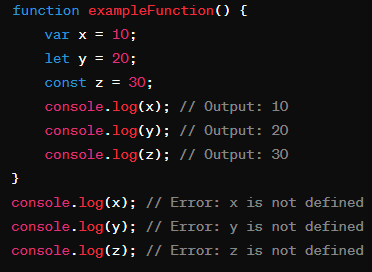
SCOPO DELLE VARIABILI

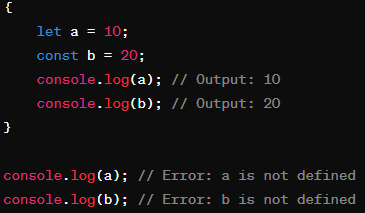
**-Variabili locali** 🡪

**-Variabili globali** 🡪

Le variabili in JS possono avere 2 forme, in base a come le dichiaramo, sono legate alla visibilità delle variabili dichiarate all'interno di una funzione o di un blocco di codice:

**-Function scope** 🡪 dichiarate all'interno di una funzione sono visibili solo all'interno di quella funzione e non sono accessibili al di fuori di essa.



**-Block scoped** 🡪 si riferisce alla visibilità delle variabili all'interno di un blocco di codice delimitato da {}.

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:



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** (&& || !) …



-**SWITCHCASE** 🡪 lo so dai…

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



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



-**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:



CICLI DI RIPETIZIONE

-**WHILE** 🡪 i know bruh

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



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



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 ‘ ‘



METODI STRINGHE 🡪 <https://www.w3schools.com/js/js_string> methods.asp

-**.lenght** 🡪 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)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments/length)

- **.concat()** 🡪**concatena** una stringa ad un altra stringa e restituisce una nuova stringa, e sarà di tipo **STRING:**

[String.prototype.concat() - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/concat)



-**.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), sarà di tipo **STRING**:

[String.prototype.charAt() - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/charAt)



-.**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)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/indexOf)

-**.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)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/split?retiredLocale=it)



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

[String.prototype.slice() - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/slice)



è

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).

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:



-**.substring()** 🡪 è uno dei 2 metodi **molto simile a .slice()**, e fa l’idem lavoro dello slice (anche se oggi è considerato obsoleto)

[String.prototype.substring() - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/substring)

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:**



-**.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)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/substr)

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



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



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

****

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

[String.prototype.toUpperCase() - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/toUpperCase)

[String.prototype.toLowerCase() - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/toLowerCase)

-**.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)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/trim)



Per restituire una nuova stringa con spazi vuoti tagliati da una sola estremità, utilizzare [trimStart()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/trimStart) o [trimEnd().](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/trimEnd)

-**.replace()** 🡪 [String.prototype.replace() - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace)

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:



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 possno supportare queste specifiche su **JS 2015**)



REGULAR EXPRESSION

Le usiamo per cercare qualcosa all’interno di una stringa

**VEDERE SLIDE** [**js-05b-regex**](js-05b-regex.pdf)**, 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](%20%20%20https:/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**:



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



- **accedere all’array** 🡪 per accedere agli array utilizziamo la cosidetta "**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:



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

-**modficare un array** 🡪 per modificare un l’elemento di un array, anche qui utilizziamo anche qui la "**bracket notation”**



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



ma anche tramite il metodo **.push()** 🡪 aggiunge quell’elemento automaticamente nella posizione successiva all’ultima

-**loops con array e stringhe** 🡪 per loopare tutti gli elementi di un’array, usare “SEMPRE” un **ciclo for:**



Ma possiamo anche looppare per una un elemento di tipo **string:**



TEMPLATE LITERALS $ { }

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

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



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**.



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:



- **.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)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice)

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**, verrano inseriti prima dell’indexStart

Esempi utilizzando **indexStart** e **deleteCount**:







Esempi utilizzando **indexStart** e **deleteCount** e **items:**



!! RESTITUISCE GLI ELEMENTI ELEMINATI ANCHE!! VEDERE MEGLIO [SU MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice) !!

- **.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)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/join?retiredLocale=it#try_it)





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



- **.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)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat?retiredLocale=it)



-**.find()** 🡪

[Array.prototype.find() - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find?retiredLocale=it)

-**.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)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/includes?retiredLocale=it)



7 LEZIONE

INTRODUZIONE AGLI OGGETTI (OBJECT)

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



nome 🡪 si chiama **CHIAVE**  **+**  Gabriele 🡪 si chiama **VALORE**



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 ( **,** ) :



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

**PROPRIETA** 🡪 cioè delle variabili: nome, cognome, eta, hobbby, indirizzo.

La domanda è: queste proprietà appartengono alla persona?

**METODI** 🡪 sono le funzioni di particolari oggetti ex: salututi: 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:











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 doppie virgolette** ( **[ “ “ ]** )

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

vediamo che avrà lo stesso effetto rispetto alla dot notation:



E =



E =

QUANDO USARE DOT NOTATION | BRACKET NOTATION

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** !!



**2**.**AGGIUNGERE NUOVE PROPRIETA ALL’OGGETTO**



**3**.**ELIMINARE DELLE PROPRIETA ALL’OGGETTO 🡪 delete**



PAROLA CHIAVE “THIS”

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



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>



Se hai bisogno dei **valori** delle proprietà 🡪 usa [Object.values()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values)

Se sono necessari sia le **chiavi** che i **valori** della proprietà 🡪 utilizzare [Object.entries()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries).

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**:



8 LEZIONE 🡪 [Stack, queue & heap slide prof](file:///C:\Users\ICTS23-25.256\Desktop\JAVA%20SCRIPT\APPUNTI\js-08b-event-loop.pdf)

9 LEZIONE 🡪 [timing and interval slide prof](file:///C:\Users\ICTS23-25.256\Desktop\JAVA%20SCRIPT\APPUNTI\js-08-timing.pdf)

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 sintax





Andiamo ad analizzare la sintassi dei singoli parametri:

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



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

-**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**



2.setInterval sintax





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



clearInterval

Date and Timeàhttps://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()** àcreates  a date object with the **current date and time**:



Forma

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

FormaForma





-**new Date(*year, month, day…*)** à creates a date object with a **specified date and time**:



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

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

Forma

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

**zero time** 🡪 parte **sempre** da 01 January 1970

Forma

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 **.get*X*** è possibile farlo.

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

ANNO (4 CIFRE)

Forma

MESE (0-11)



Forma

GIORNO DEL MESE (1-31)

Forma

ORA (0-23)

Forma



MINUTI (0-59)



Forma

SECONDI (0-59)



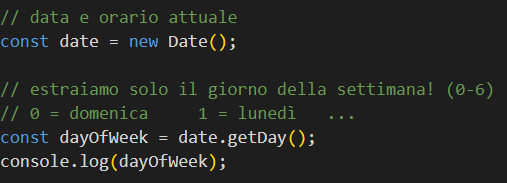
Forma

MILLISECONDI ( 0-999)



Forma

GIORNO DELLA SETTIMANA LUNEDI MARTEDI… (0-6)

Forma

-**.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()

Forma