Labforweb

nerd academy

Javascript

Intro a JavaScript

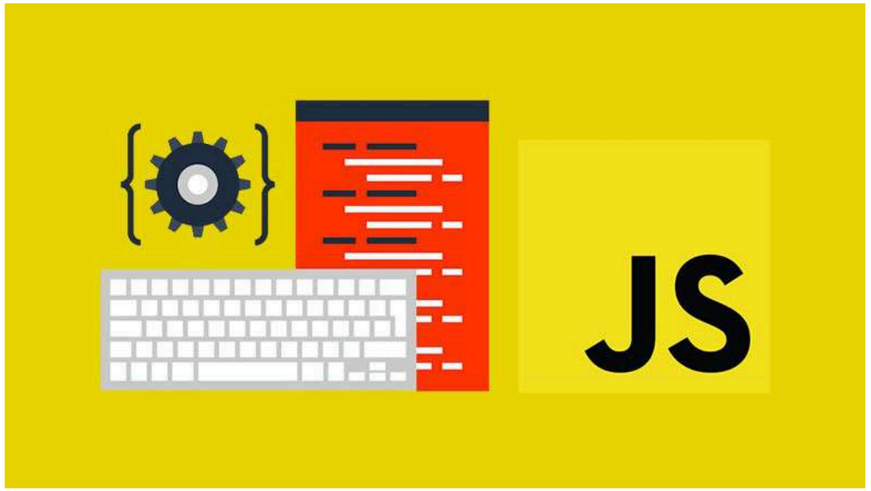
JavaScript è un *linguaggio di programmazione* che consente di aggiungere funzionalità interattive e contenuti dinamici alle pagine web.
Esempi di contenuti JavaScript sono i **moduli compilabili** e **slideshow** di gallerie fotografiche.

Intro a JavaScript

In passato, le pagine Web erano **statiche**, nel senso che non esisteva alcun tipo di interazione lato utente.
JavaScript è nato per rendere dinamiche le applicazioni Web.
Utilizzando JavaScript, il Browser è in grado di rispondere alle interazioni degli utenti e modificare il layout dei contenuti della pagina Web.

Intro a JavaScript

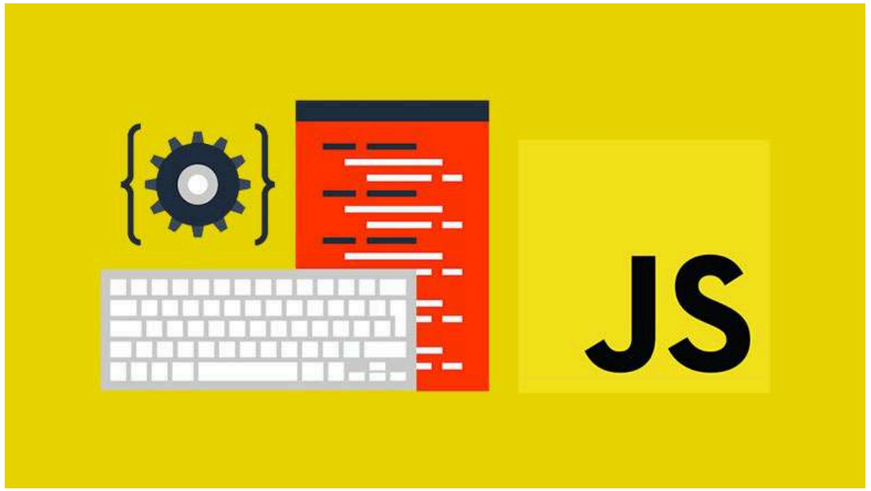
In particolare, è un "**linguaggio di scripting**": il Browser interpreta sequenzialmente ogni riga del programma e lo esegue, quindi passa alla riga successiva.



Linguaggio di scripting = linguaggio interpretato che viene tradotto in linguaggio macchina soltanto nel momento in cui è eseguito.

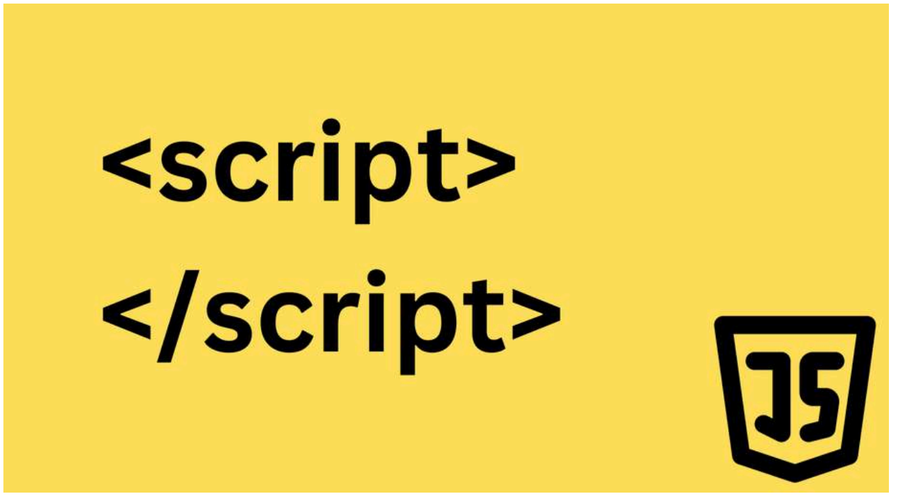
Intro a JavaScript

JavaScript, insieme ad HTML e CSS, è una delle tecnologie principali della programmazione Front-End.
Gestisce il comportamento degli elementi di una pagina web, ovvero il contenuto (HTML) e lo stile (CSS).



Come inserire uno script in una pagina web

Per poter essere eseguito dal Browser, JavaScript deve essere collegato al rispettivo file HTML.
Per inserire codice JavaScript all'interno di un documento si utilizza il tag **<script>**.



Come inserire uno script in una pagina web

Lo *script* può essere inserito sia nella sezione **<head>** che nella sezione **<body>** di un documento HTML.

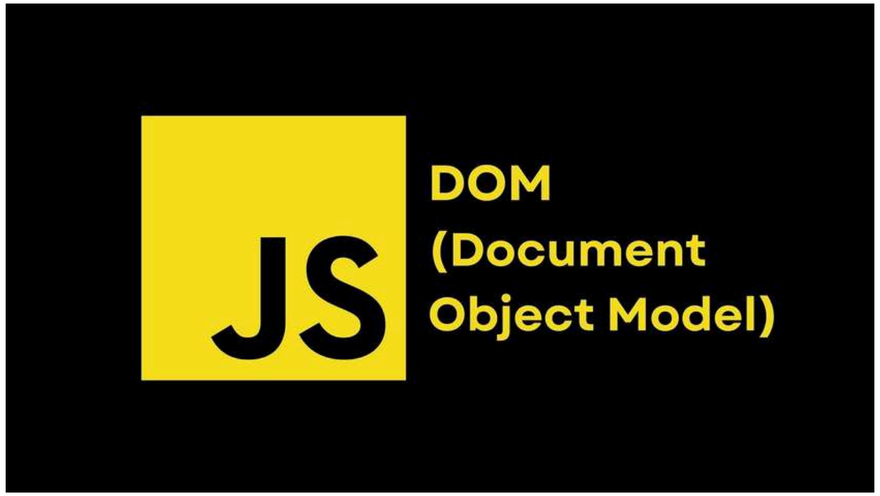
Come inserire uno script in una pagina web

Gli script inseriti nell'<head> vengono caricati ed eseguiti **prima** degli script inseriti nella sezione <body>, i quali vengono eseguiti **sequenzialmente**, secondo l'ordine di caricamento.

Relazione tra JavaScript e HTML

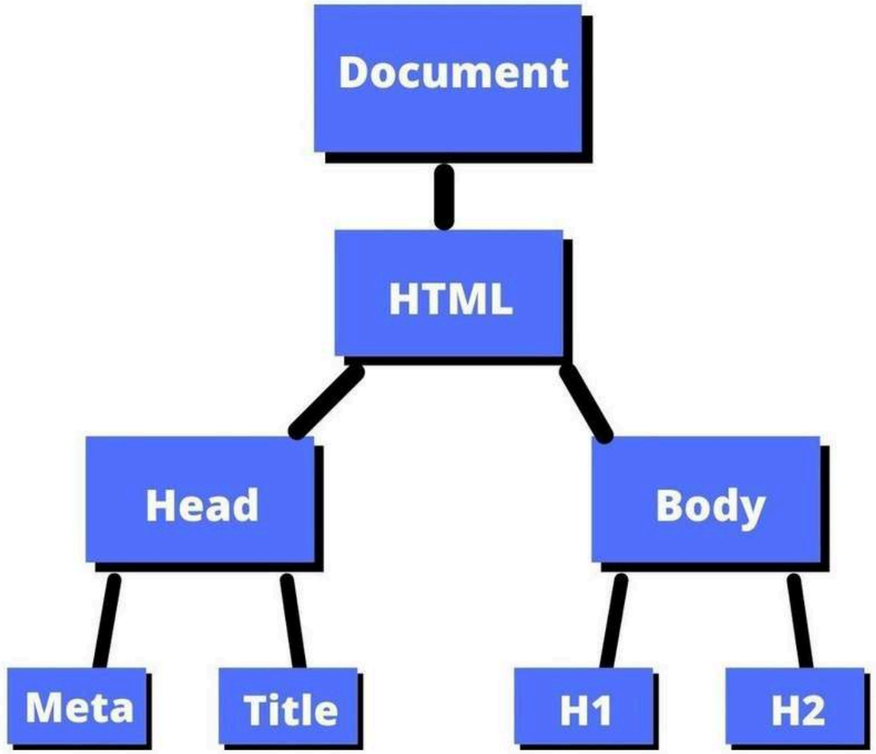
Ma in che modo JavaScript riesce a manipolare l'HTML?

↗ Grazie al **DOM** (**Document Object Model**).



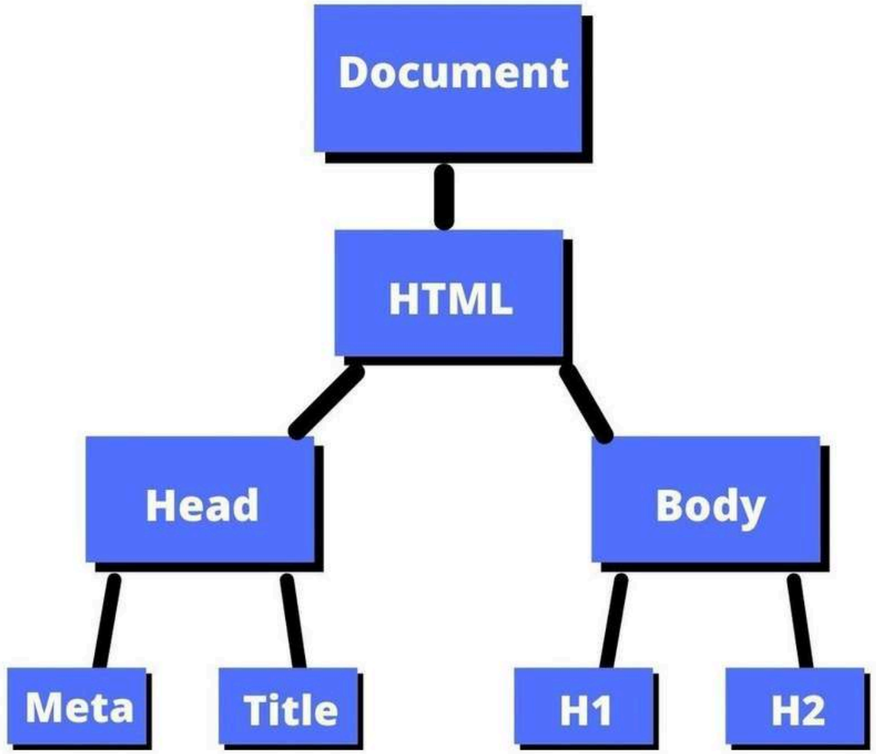
Approccio DOM

Il **DOM** è un modello che rappresenta il ***document***, ovvero la struttura ed il contenuto della pagina web.



Approccio DOM

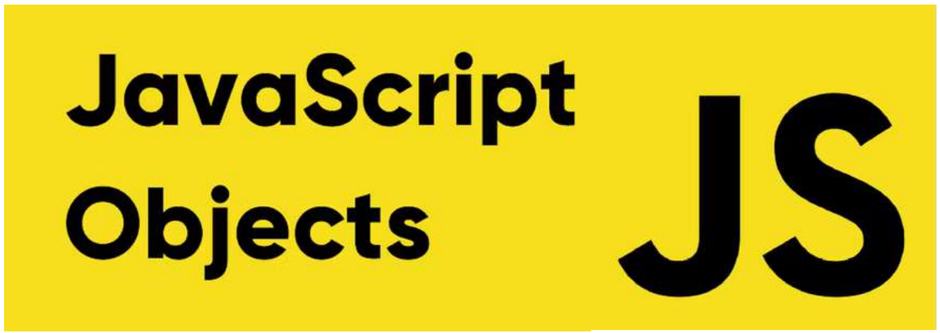
Il **DOM** Definisce una serie di funzionalità per *accedere* e *manipolare* un documento, rappresentandolo come una ***struttura gerarchica ad albero***.
In base a questo approccio, la struttura ed il contenuto di un documento viene creata come una ***gerarchia di oggetti***, ciascuno dei quali rappresenta i diversi elementi della pagina.



Oggetti JavaScript

Ma cos’è un Oggetto?

Gli Oggetti JavaScript sono dei **contenitori di dati e funzionalità** (chiamati rispettivamente "**proprietà**" e “**metodi**”), e sono utilizzati per rappresentare entità specifiche, come ad esempio una persona, un prodotto, etc...



Oggetti Nativi

Oggetto Nativo

È un oggetto che fa parte del linguaggio JavaScript nativo, e che quindi non ha bisogno di essere "*istanziato*" da zero.

Oggetti Nativi

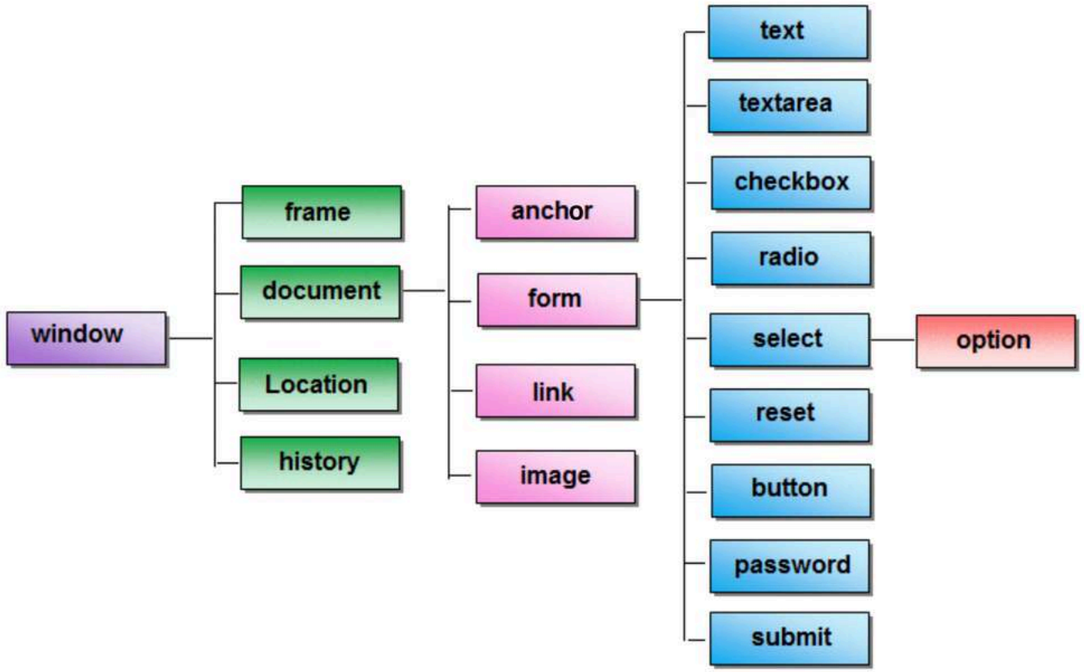
Gli *oggetti nativi* più importanti di Javascript sono:

- **window**
- **document**
- **location**
- **history**
- **console**

Oggetti Nativi

Oggetto Window

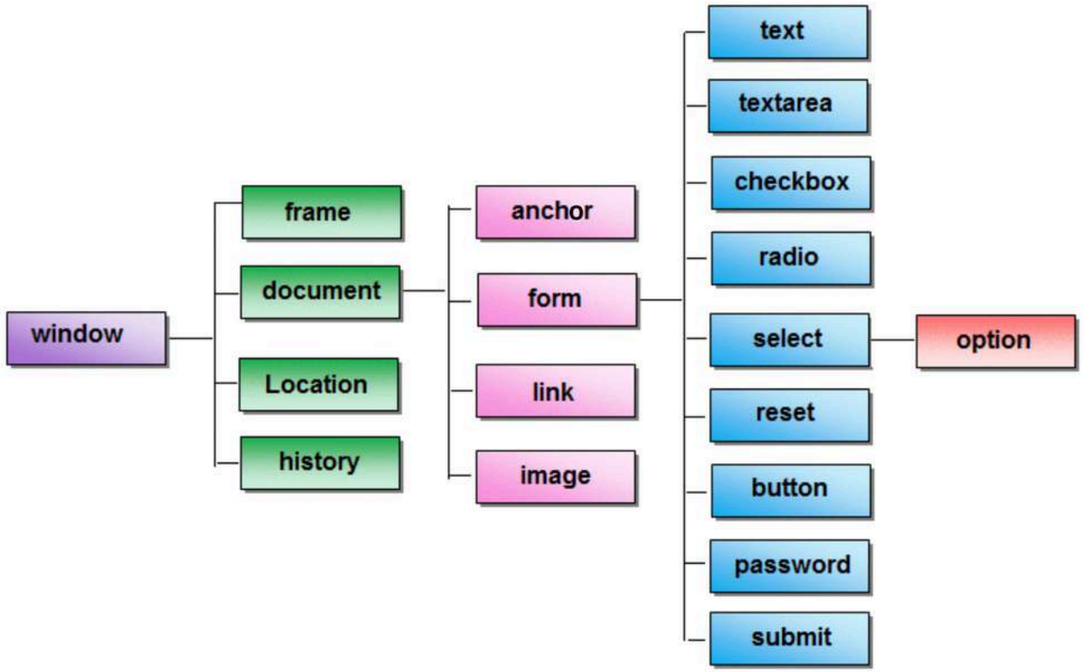
È l'oggetto principale della gerarchia degli oggetti, e descrive la finestra del browser correntemente aperta. Ogni finestra del browser viene rappresentata da un oggetto Window, che definisce proprietà e metodi per la programmazione lato client.



Oggetti Nativi

Proprietà dell'Oggetto Window

Alcune delle proprietà dell'oggetto window sono a loro volta degli oggetti, con le loro proprietà e metodi.

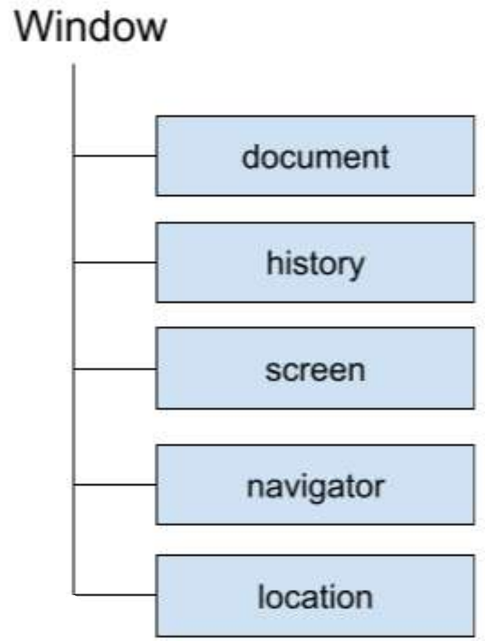


Oggetti Nativi

Proprietà dell'Oggetto Window

Ecco alcune *proprietà-oggetti* di window:

- ***document***
- ***history***
- ***navigator***
- ***screen***
- ***location***



Oggetti Nativi

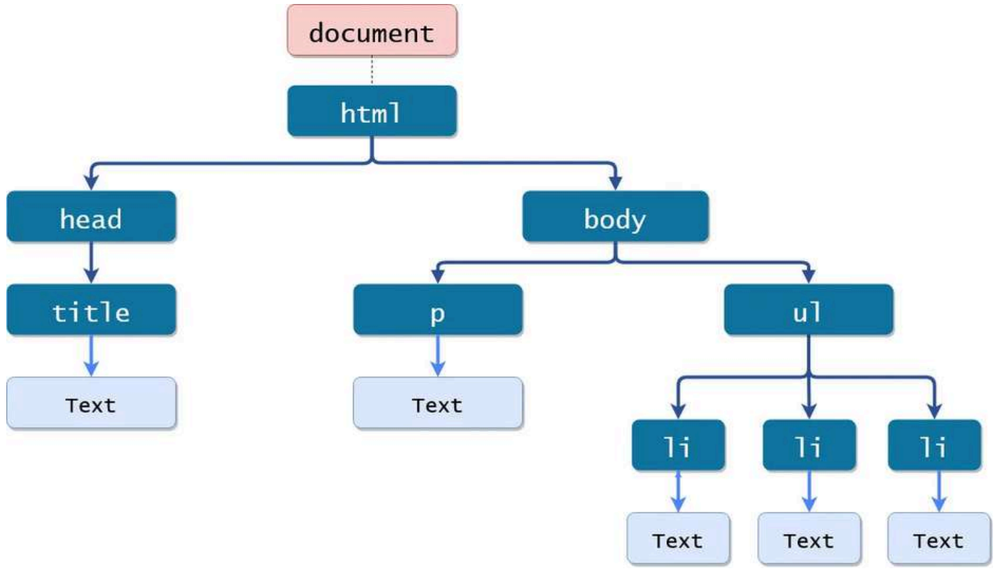
Alcuni Metodi dell'Oggetto Window

- *window*.***open()*** - consente di aprire una nuova finestra del Browser (poco utilizzato oggi per l'utilizzo di AdBlocker da parte degli utenti);
- *window*.***close()*** - chiude la finestra corrente;
- *window*.***setTimeout()*** - permette di eseguire *(solo una volta)* la funzione, dopo un intervallo di tempo specificato;
- *window*.***alert()*** - visualizza una finestra di dialogo nel Browser;
- window.**scroll()** - consente di scorrere la finestra sia orizzontalmente che verticalmente.

Oggetti Nativi

Oggetto Document

Il Document è l'oggetto che rappresenta il documento correntemente visualizzato nella finestra del Browser, ed è caratterizzato da molte *proprietà* e *metodi*.



Oggetti Nativi

Proprietà dell'Oggetto Document

- *document*.***images*** - restituisce un array con le immagini presenti nel documento HTML;
- *document*.***links*** - restituisce un array con tutti i link presenti nel documento;
- *document*.***lastModified*** - indica la data ultima in cui il documento è stato modificato;
- *document*.***forms*** - restituisce un array con i moduli presenti nel documento HTML;

Oggetti Nativi

Metodi dell'Oggetto Document

- *document*.***getElementById()*** - restituisce l'elemento che ha l'ID specificato;
- *document*.***getElementsByClassName()*** - restituisce una *collezione* di tutti gli elementi con la classe specificata;
- *document*.***getElementsByTagName()*** - restituisce una *collezione* di tutti gli elementi con il nome del tag specificato;
- *document*.***querySelector()*** - restituisce *il primo* elemento che corrisponde al selettore CSS specificato;

Oggetti Nativi

Altri Metodi dell'Oggetto Document

- *document*.***querySelectorAll()*** - restituisce *tutti* gli elementi nel documento che corrispondono al selettore CSS specificato;
- *document*.***createElement()*** - crea un nuovo elemento con il nome del tag specificato;
- ***.appendChild()*** - aggiunge un nodo come *ultimo figlio* di un tag;
- ***.removeChild()*** - rimuove un figlio dal tag specificato.

Oggetti Nativi

Oggetto Location

Quest'oggetto contiene tutte le informazioni relative alla *posizione* del documento correntemente visualizzato, come ad esempio l'URL.

Oggetti Nativi

Proprietà dell'Oggetto Location

- *location*.***href*** - restituisce l'URL completo del documento corrente, può anche essere utilizzata per reindirizzare l'utente a un nuovo URL;
- *location*.***protocol*** - restituisce il protocollo web del documento corrente, ad esempio "*http:*" o "*https:*";
- *location*.***pathname*** - restituisce una parte del percorso dell'URL.

Oggetti Nativi

Metodi dell'Oggetto Location

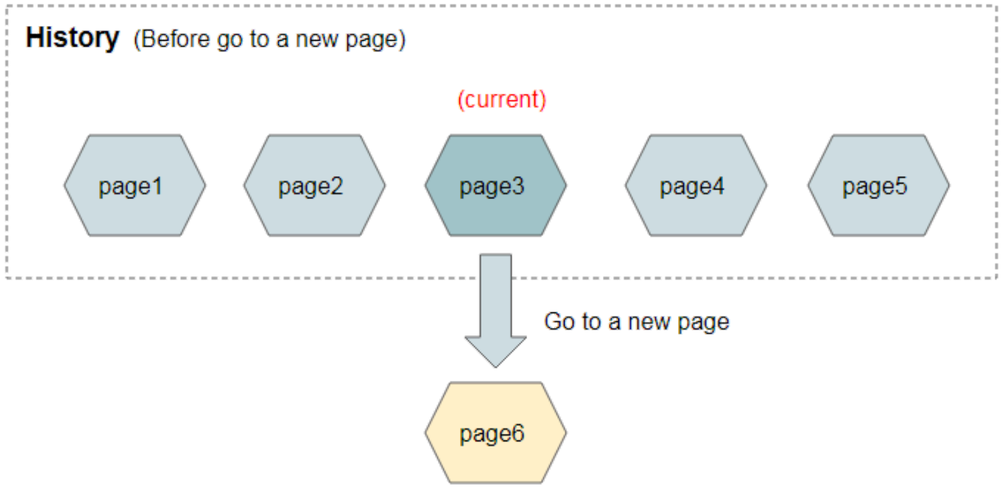
- *location*.***reload()*** - ricarica il documento corrente;
- *location*.***replace(URL)*** - sostituisce il documento corrente con un nuovo documento specificato dall'URL;
- *location*.***assign(URL)*** - carica un nuovo documento specificato dall'URL; a differenza del metodo *replace()*, lascia una traccia del documento originale nella cronologia del Browser.

Oggetti Nativi

Oggetto History

L'oggetto History contiene informazioni relative alla *cronologia* delle pagine visitate sul Browser.

Inoltre, permette di navigare avanti e indietro tra le pagine visitate.

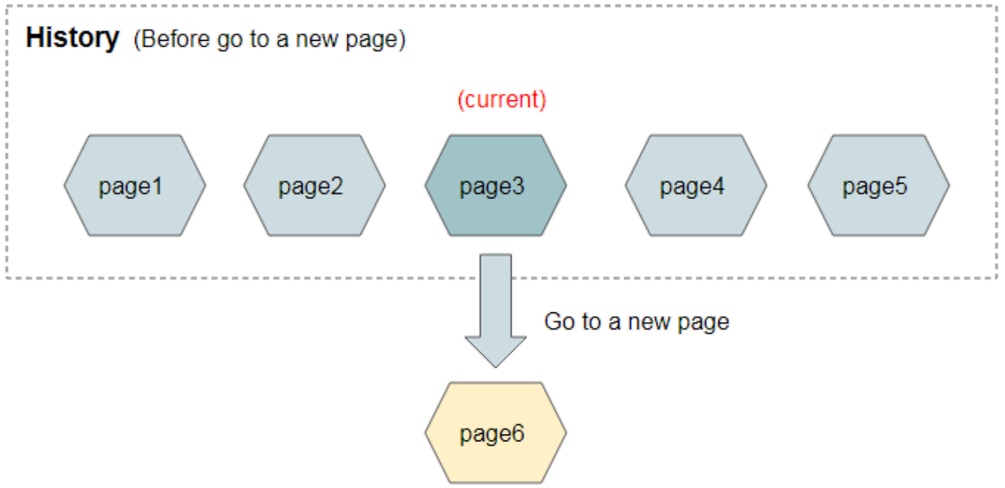


Oggetti Nativi

Proprietà dell'Oggetto History

Questo oggetto dispone della proprietà **length**, che restituisce *il numero* delle URL presenti nella cronologia:

- *history*.***length***



Oggetti Nativi

Metodi dell'Oggetto History

- *history*.***back()*** - manda indietro (alla posizione precedente);
- *history*.***forward()*** - manda avanti (alla posizione successiva);
- *history*.***go()*** - carica una specifica URL tra quelle presenti in cronologia.

Oggetti Nativi

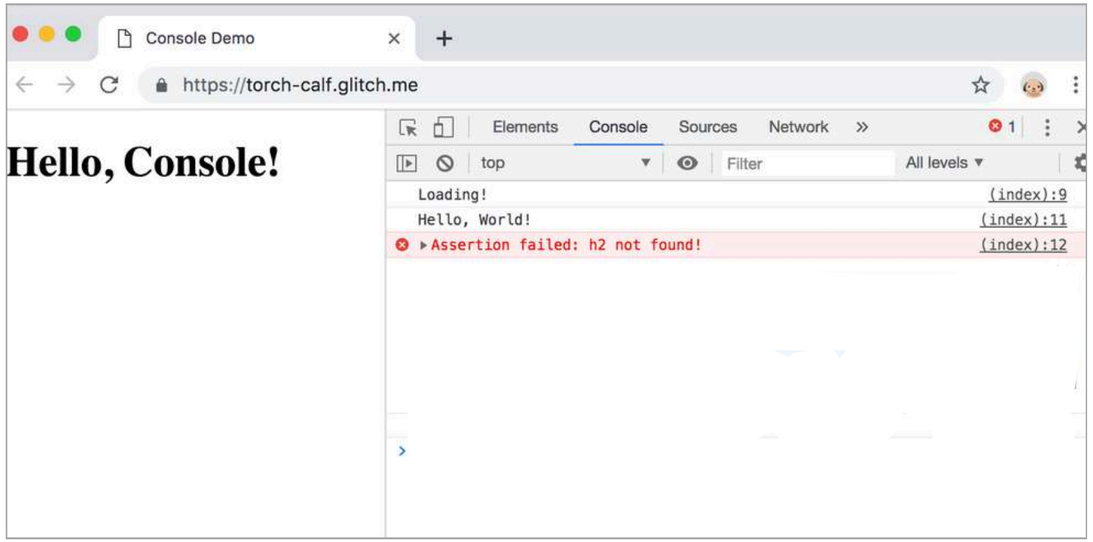
Oggetto Console

L'oggetto Console è uno strumento che permette di interagire con la console del Browser.

Oggetti Nativi

Console del Browser

La console è un ambiente di **debugging**, che consente agli sviluppatori di visualizzare messaggi, errori e varie informazioni durante lo sviluppo del codice. Rappresenta quindi uno strumento fondamentale, poiché consente di monitorare il comportamento del codice.



Sintassi di JavaScript

JavaScript è **Case Sensitive**, ossia un linguaggio in cui c'è distinzione netta tra *caratteri maiuscoli* e *minuscoli* (vengono gestiti come lettere differenti). Questo significa che le parole chiave del linguaggio (*variabili*, *funzioni*, ...) devono sempre essere digitati con una ***capitalizzazione coerente*** delle lettere.

Sintassi di JavaScript

JavaScript è un linguaggio **strutturato**, cioè possiede alcune strutture tipiche dei linguaggi di programmazione:

- *Istruzioni Condizionali* (*es.* ***if-else***)
- *Istruzioni Iterative* (*es.* ***ciclo for***)
- *Funzioni*

Sintassi di JavaScript

Tipi di Dato in JavaScript

In JavaScript i dati possono essere di diverso tipo, in particolare:

- **Stringhe**
- **Numeri**
- **Booleani**
- **undefined**
- **null**

Sintassi di JavaScript

Stringhe

Le stringhe rappresentano dati di tipo *testuale.*

Si può quindi affermare che una stringa è una ***sequenza finita di caratteri alfanumerici***.

Sintassi di JavaScript

Stringhe

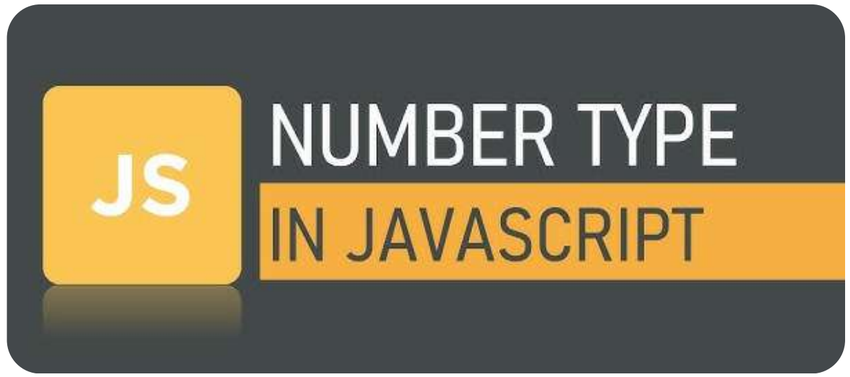
Inoltre, le stringhe vanno sempre inserite all'interno di **apici**, *doppi* o *singoli*.

Sintassi di JavaScript

Numeri

Il tipo **number** viene usato per rappresentare sia per i *numeri interi* che per quelli *in virgola mobile*.

Per questi valori si hanno a disposizione diverse operazioni, dalle classiche 4 (*moltiplicazione*, *addizione*, *sottrazione* e *divisione*) a quelle più complesse.



Sintassi di JavaScript

Numeri

Per questa tipologia di dati esistono dei valori speciali, ad esempio **Infinity** e **NaN**:

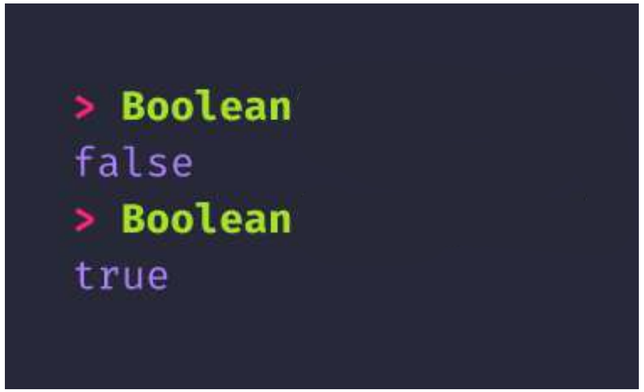
- *Infinity* - rappresenta il concetto matematico di infinito;
- *NaN (Not a Number)* - rappresenta un errore di calcolo, e può essere il risultato di un'operazione non corretta o indefinita.

```
NaN
+Infinity
-Infinity
```

Sintassi di JavaScript

Boolean

I dati booleani prevedono soltanto 2 valori:
true o **false**.
Sono spesso utilizzati nelle variabili, per memorizzare valori del tipo *Sì*/*No*.



Sintassi di JavaScript

Boolean

I valori booleani si ottengono anche come risultato di operazioni *di confronto*:

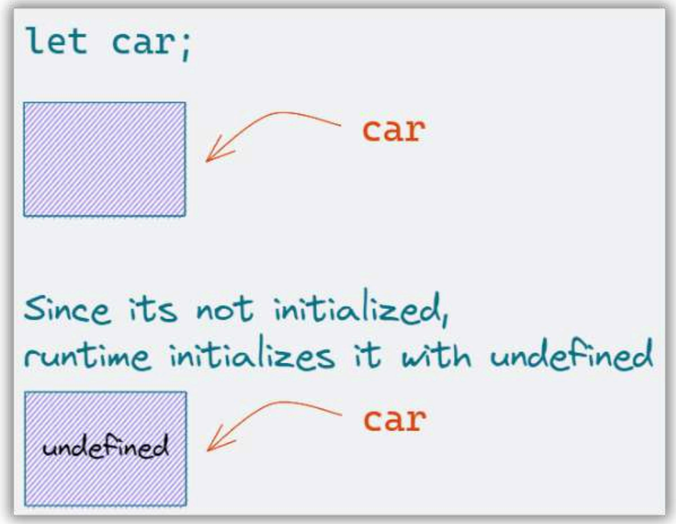
```
> Boolean(43<556);
<· true
> Boolean('hello'=="hello");
<· true
> Boolean(9%4==2);
<· false
```

Sintassi di JavaScript

undefined

Il valore speciale *undefined* indica l'**assenza di valore**.

Ad esempio, se una variabile viene *dichiarata* ma non *assegnata*, il suo valore sarà esattamente *undefined*.

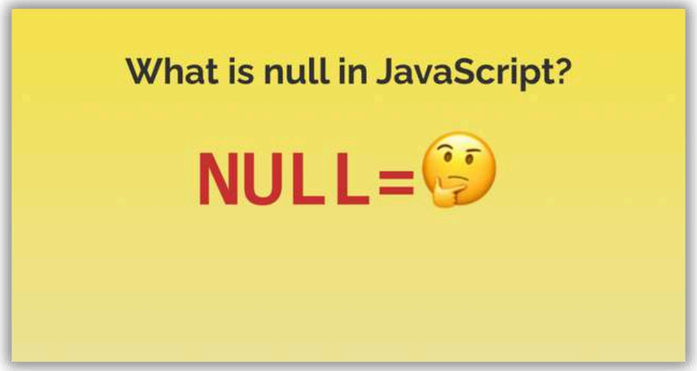


Sintassi di JavaScript

null

Il *null* è un tipo di dato speciale utilizzato per indicare il *valore nullo*, *vuoto*.

Spesso il *null* viene interpretato come un errore, ma non è così: rappresenta un valore a tutti gli effetti.

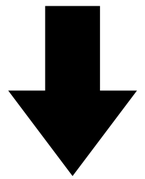


Sintassi di JavaScript

null

Molto spesso, si ottiene questo valore quando si fa riferimento ad un tag HTML *inesistente*:
in questo caso, la variabile "*tagH1*" risulterà *null*, poichè nel documento non esiste alcun elemento con *id="title"* (il tag <h1> infatti ha ***class***="title").

```
<body>

    <h1 class="title">Titolo</h1>

</body>
```

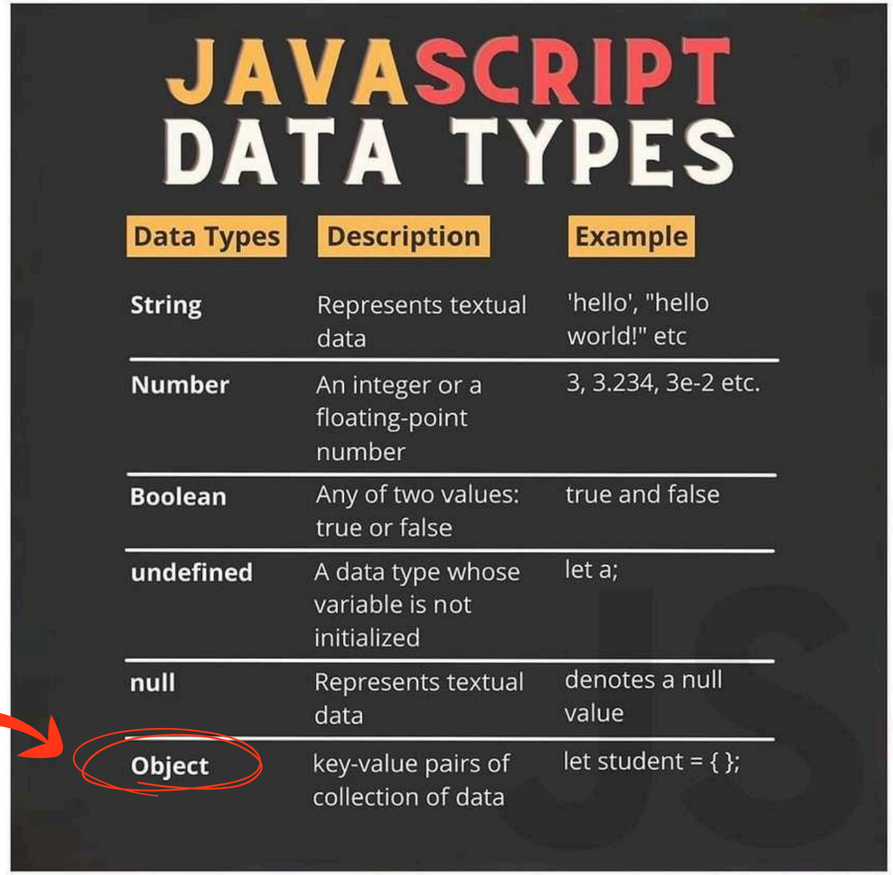


```
let tagH1 = document.getElementById('title');

console.log(tagH1);  // null
```

Sintassi di JavaScript

Tipi di Dato in JavaScript

Tutto ciò che non è un *valore primitivo*, è un oggetto.



undefined VS null

undefined

"*undefined*" significa che una variabile è stata dichiarata ma non le è stato assegnato alcun valore. In sintesi, rappresenta la **mancanza di un valore assegnato**.

```
let variabileTest;
console.log(variabileTest); // undefined
```

null

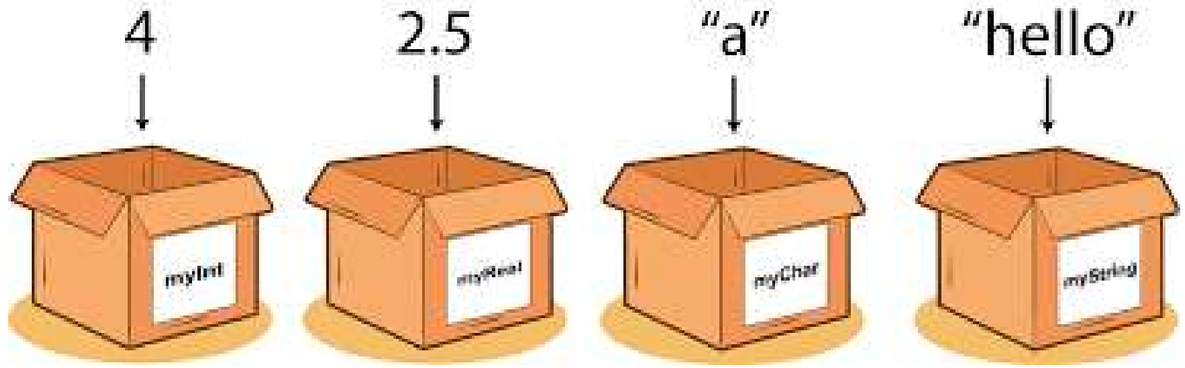
"*null*" è un valore di assegnazione, e si utilizza per indicare "**nessun valore**". Può essere assegnato a una variabile come rappresentazione di *valore vuoto*.

```
let variabileTest = null;
console.log(variabileTest); // null
```

Sintassi di JavaScript
Variabili

Una variabile *è uno "**spazio di memoria**"* utilizzato per salvare dati.

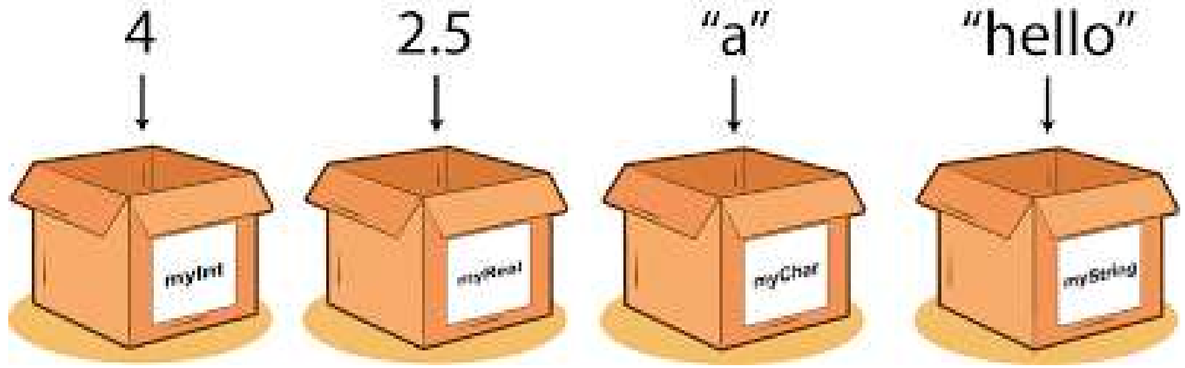
Ogni variabile ha un *nome* e può contenere *diversi tipi di dato*.



Sintassi di JavaScript
Variabili

Per dichiarare una variabile si utilizza una tra le seguenti *parole chiave*:

- **var**
- **let**
- **const** (per le ***costanti***)



Variabili
Differenza tra *var*, *let* e *const*

Prima di analizzare le differenze tra queste istruzioni, occorre introdurre il concetto di **ambito di visibilità** (**scope**).

In breve, lo *scope di una variabile* rappresenta la porzione di codice in cui una variabile è visibile e di conseguenza può agire.

Variabili
Differenza tra *var*, *let* e *const*

L'ambito di visibilità può essere *globale* o *locale*:

- con **scope globale** (global scope) si intende tutta l'area codice;

- con **scope locale** (local scope) si indica una porzione di codice.

Variabili
Differenza tra *var*, *let* e *const*

A sua volta, lo **scope locale** si divide in:

- **scope funzionale** (functional scope), porzione di codice delimitata da una *funzione;*
- **scope di blocco** (block scope), porzione di codice delimitata dalle *parentesi graffe* (non solo quelle delle funzioni).

Variabili
Differenza tra *var*, *let* e *const*

L'ambito di visibilità è uno dei principali aspetti che differenzia **var**, **let** e **const**.

var

Le variabili *var* hanno ambito ***globale*** o di *funzione.*

Inoltre possono essere sia **aggiornate** che **ri-dichiarate**.

```
var hey = "hey";  // Ambito globale

   function newFunction() {
       var hello = "hello";  // Ambito funzionale
   }

console.log(hello);  // error: hello is not defined
```

Variabili
Differenza tra *var*, *let* e *const*

Ri-dichiarare var

```
// Ri-dichiarazione

var greeter = "hey hi";
var greeter = "say Hello instead";
```

Aggiornare var

```
// Aggiornamento

var greeter = "hey hi";
greeter = "say Hello instead";
```

Variabili
Differenza tra *var*, *let* e *const*

Limiti delle variabili var ⚠️

Il fatto che queste variabili possano essere *ri-dichiarate*, può portare a diversi problemi in fase di sviluppo.

Vediamo un esempio...

```
var saluto = "hey";
var times = 4;

if (times > 3) {
   var saluto = "ciao!";
}

console.log(saluto);  // "ciao!"
```

Variabili
Differenza tra *var*, *let* e *const*

In questo caso, visto che (times > 3) restituisce *true*, la variabile *saluto* è **ri-definita** a "*ciao!*".

```
var saluto = "hey";
var times = 4;

if (times > 3) {
   var saluto = "ciao!";
}

console.log(saluto);  // "ciao!"
```

Variabili
Differenza tra *var*, *let* e *const*

Ciò non rappresenta un problema, se si desidera che "*saluto*" venga effettivamente ri-definita, ma nel caso in cui non ci si accorga che una variabile con lo stesso nome è già stata utilizzata, potrebbero esserci sorprese sull'output ottenuto, e questo causerà probabilmente molti bug.

```
var saluto = "hey";
var times = 4;

if (times > 3) {
   var saluto = "ciao!";
}

console.log(saluto);  // "ciao!"
```

Variabili
Differenza tra *var*, *let* e *const*

let

Le variabili *let* hanno **ambito di *blocco***
(vedi esempio a fianco).
Possono essere **aggiornate** ma ***non***
ri-dichiarate.

```
let hey = "hey";
let times = 4;

 if (times > 3) {
     let hello = "hello";
     console.log(hello);  // "hello"
 }

console.log(hello)  // hello is not defined
```

Variabili
Differenza tra *var*, *let* e *const*

Ri-dichiarare let

```
// Ri-dichiarazione

let greeting = "say Hi";
let greeting = "say Hello instead";  // error: Identifier 'greeting'
                                        has already been declared
```

Aggiornare let

```
// Aggiornamento

let greeting = "say Hi";
greeting = "say Hello instead";
```

Variabili
Differenza tra *var*, *let* e *const*

let

Il fatto di **non** poter *ri-dichiarare* le variabili *let*, risolve il problema che si aveva con *var*:

```
let hello = "hey";

    if (true) {
        let hello = "say hello";
        console.log(hello);  // "say hello"
    }

    console.log(hello);  // "hey"
```

Variabili
Differenza tra *var*, *let* e *const*

Perche non c'è errore?

Perché entrambe le istanze sono trattate come *variabili differenti*, visto che hanno **ambito diverso**.

Inoltre, visto che una variabile *let* non può essere dichiarata più di una volta dentro un ambito, il problema discusso prima inerente a *var*, non si presenta.

```
let hello = "hey";

    if (true) {
        let hello = "say hello";
        console.log(hello);  // "say hello"
    }

    console.log(hello);  // "hey"
```

Variabili
Differenza tra *var*, *let* e *const*

const

Le variabili dichiarate con *const* mantengono **valori costanti**.

Le dichiarazioni *const* hanno delle similitudini con le dichiarazioni *let*.

Variabili
Differenza tra *var*, *let* e *const*

const

Le *costanti*, esattamente come le variabili *let*, hanno ambito di *blocco*, **non** possono essere *ri-dichiarate*, ma **nemmeno** *aggiornate* (le variabili let invece sì).

Variabili
Differenza tra *var*, *let* e *const*

Ri-dichiarare const ❌

```
//Ri-dichiarazione

  const greeting = "say Hi";
  const greeting = "say Hello instead";  // error: Identifier 'greeting'
                                            has already been declared
```

Aggiornare const ❌

```
//Aggiornamento

const greeting = "say Hi";
greeting = "say Hello instead";  // error: Assignment to constant
                                    variable.
```

Variabili
Differenza tra *var*, *let* e *const*

const

Ogni dichiarazione *const* deve quindi essere *inizializzata* **al momento della dichiarazione**.

```
const hello;  // SyntaxError: Missing initializer in const declaration
hello = "hey!";
```

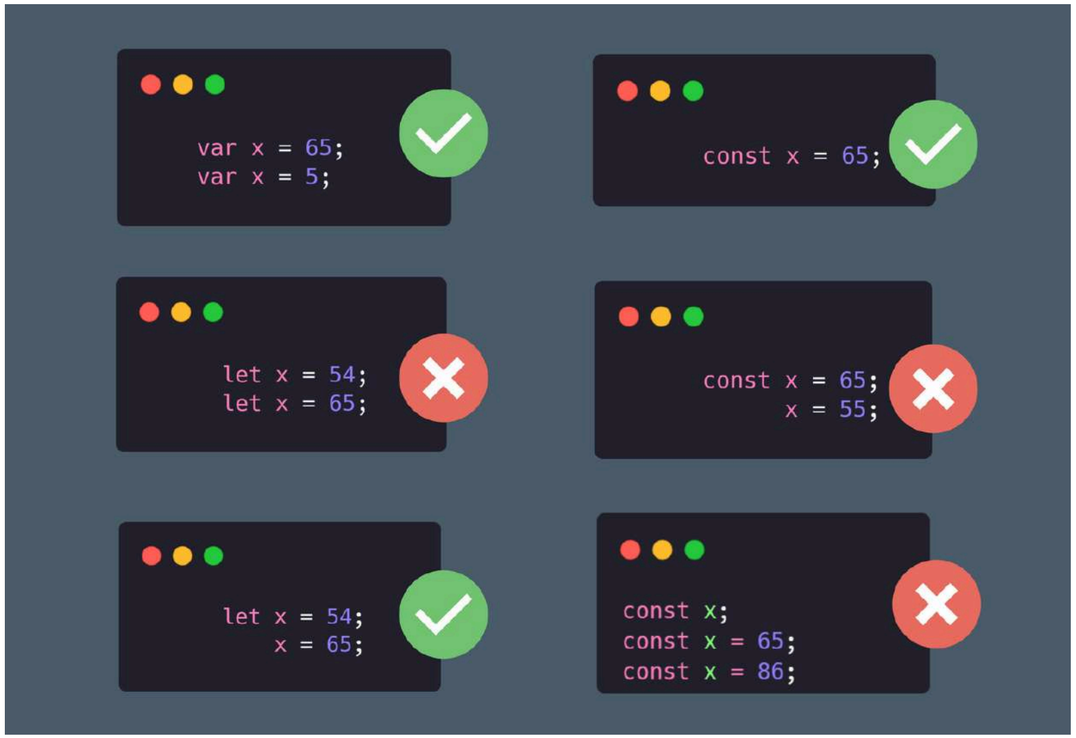
Differenza tra *var*, *let* e *const*

...Ricapitolando...

	global scoped	function scoped	block scoped	reassignable	redeclarable
var	+	+	-	+	+
let	-	+	+	+	-
const	-	+	+	-	-

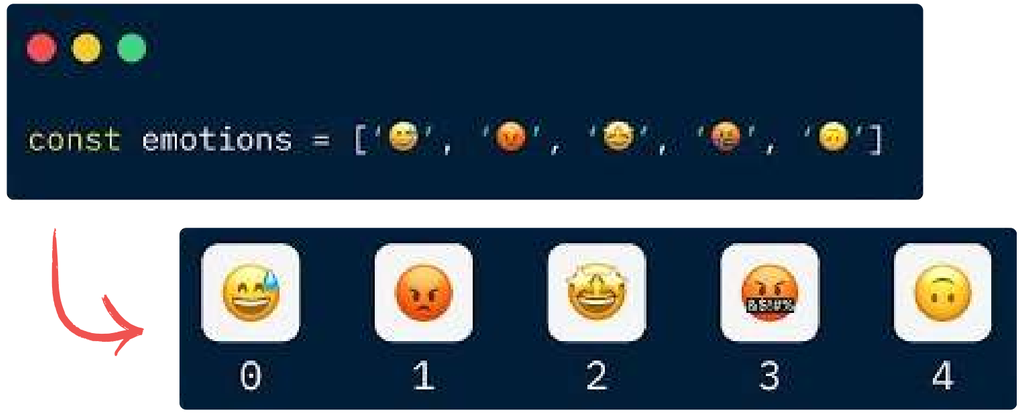
Differenza tra *var*, *let* e *const*

...Ricapitolando...



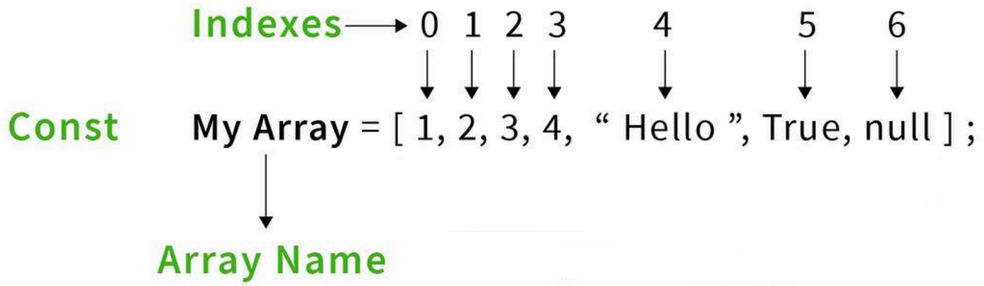
Array

Un Array è un *contenitore* che, a differenza di una variabile, può contenere **più valori**, accessibili tramite un *indice numerico*.



Array

Gli elementi di uno stesso Array possono essere **di tipo diverso** (*stringhe*, *numeri*, *booleani*, ...) e possono essere a loro volta Array.



Come definire un Array

Come si vede in questo esempio, gli elementi di un Array sono delimitati da parentesi quadre e separati da virgole:

Come definire un Array

Una volta definito un Array è possibile accedere ai singoli elementi facendo riferimento al nome dell'Array e all'indice corrispondente all'elemento, ricordandoci che la numerazione degli indici **parte da zero**.

```
let primoGiorno = giorniDellaSettimana[0];

//la variabile 'primoGiorno' contiene il primo elemento
dell'Array 'giorniDellaSettimana'
```

Array.length

È possibile determinare *quanti elementi* sono contenuti in un Array attraverso la proprietà **length**.

```
let giorniDellaSettimana = [
    "lunedì",
    "martedì",
    "mercoledì",
    "giovedì",
    "venerdì",
    "sabato",
    "domenica"];

console.log(giorniDellaSettimana.length)  // 7
```

Array.length

Questa *proprietà* viene utilizzata anche per determinare la lunghezza di una **stringa**:

Da notare che gli "*spazi vuoti*" sono considerati dei caratteri.

Operatori

- Operatori aritmetici: **+ - * / ^ %**
- Operatori logici **&& || ! (AND, OR, NOT)**
- Operatori di incremento e di decremento **++ --**
- Operatori relazionali **< <= > >=**
- Operatore di confronto **==**
- Operatore di assegnazione **=**
- Diverso da **!=**

Operatori

Doppia valenza dell'operatore +

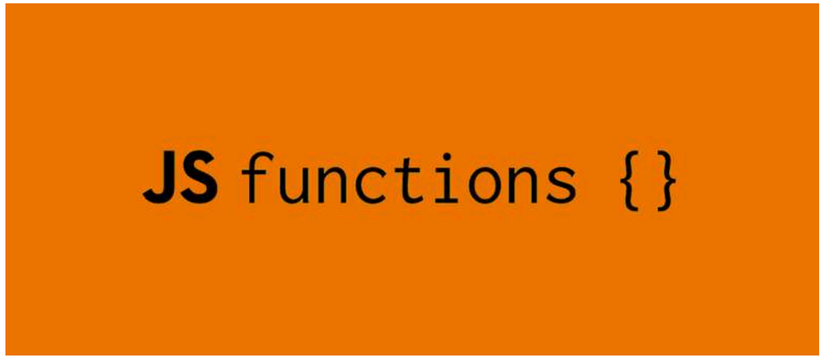
Oltre ad effettuare la somma algebrica tra 2 numeri, il "**+**" è utilizzato anche per **concatenare le stringhe**.

La *concatenazione* è il processo di aggiunta di una stringa alla fine di un'altra stringa.

```
let stringa1 = "Lab";

let stringa2 = "for";

let stringa3 = "Web";

let risultato = stringa1 + stringa2 + stringa3;

console.log(risultato);  // LabforWeb
```

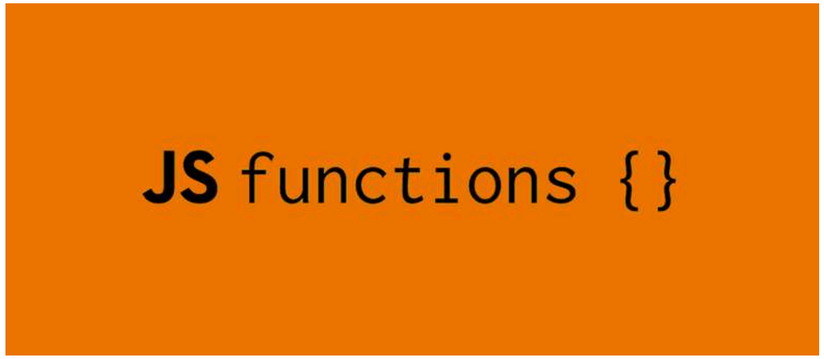
Funzioni Javascript

Una funzione è un ***insieme di istruzioni*** racchiuse in un ***blocco di codice***, che può essere contraddistinto da un nome, può accettare parametri di ingresso e restituire valori.



Funzioni Javascript

Se la funzione ha un ***nome***, esso servirà come riferimento per richiamare ed eseguire la funzione stessa, in qualunque punto del programma.

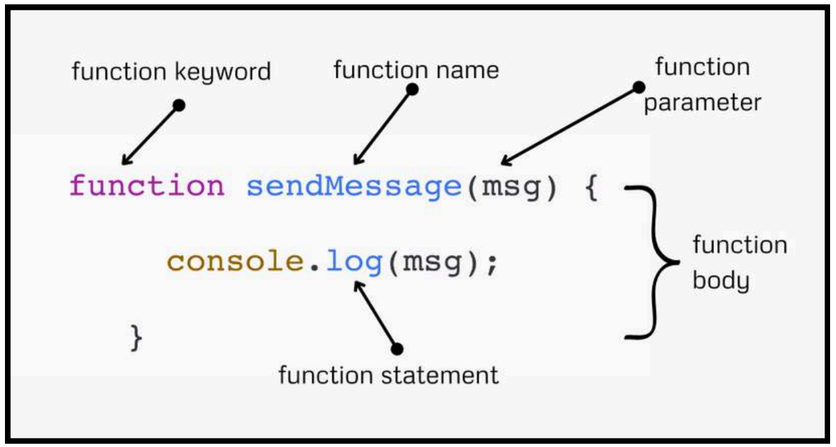


Funzioni Javascript

Di conseguenza, l'utilizzo di una funzione all'interno di uno script prevede due fasi:

- una fase di **definizione** o **dichiarazione** della funzione, in cui si assegna un nome al blocco di codice;

- una fase di **invocazione** o **chiamata**, in cui il blocco di codice viene eseguito.



Definire una funzione JavaScript

Per definire una funzione si utilizza la parola chiave **function**, seguita dal nome della funzione.

Gli **argomenti** sono una lista opzionale di variabili, separati da virgole, che verranno utilizzate all'interno del corpo della funzione.

Definire una funzione JavaScript

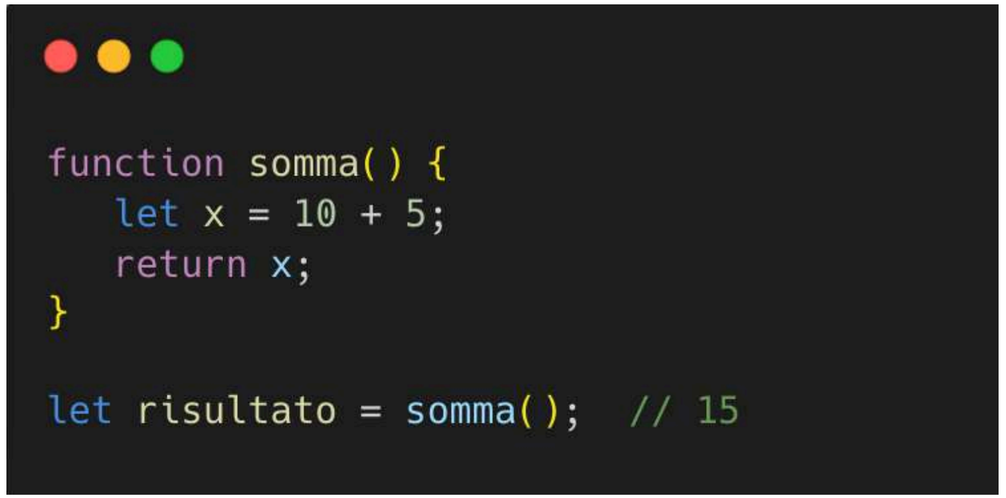
Una volta dichiarata, una funzione non viene eseguita subito.
L'esecuzione vera e propria avviene con l'**invocazione** o **chiamata**, la cui sintassi è la seguente:

```
function nome(argomenti) {
// istruzioni
}


nome(valori)
```

Il return delle funzioni

Nel corpo di una funzione può essere presente l'istruzione **return**, che consente di **terminare** e **restituire un valore** al codice che l'ha chiamata.

Questo consente di assegnare ad una variabile il valore restituito da una funzione.



```
function somma() {
   let x = 10 + 5;
   return x;
}

let risultato = somma();  // 15
```

Il return delle funzioni

In questo esempio, è stata definita una funzione senza argomenti che somma due interi e restituisce il risultato.

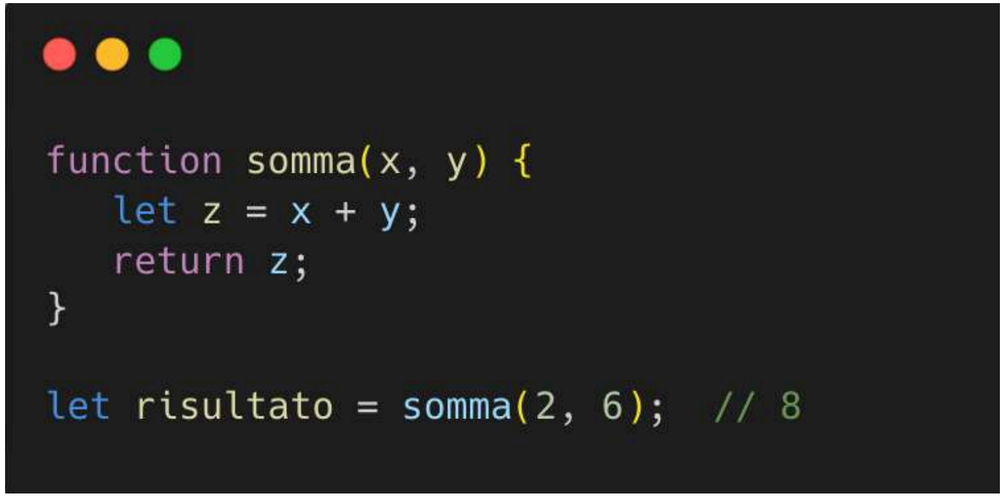
L'*invocazione* della funzione fa sì che venga eseguita la somma ed il risultato venga assegnato alla variabile "*risultato*".

```
function somma() {
   let x = 10 + 5;
   return x;
}

let risultato = somma();  // 15
```

Parametri di una funzione

Nell'esempio precedente, la funzione *somma()* è in grado di sommare soltanto i due numeri fissati nel blocco di istruzioni.

Per renderla *più generale* è opportuno introdurre due **parametri**, che rappresenteranno i numeri da sommare:



```
function somma(x, y) {
   let z = x + y;
   return z;
}

let risultato = somma(2, 6);  // 8
```

Parametri di una funzione

In questo caso, i valori da sommare verranno passati alla funzione *somma()* al momento dell'**invocazione**:

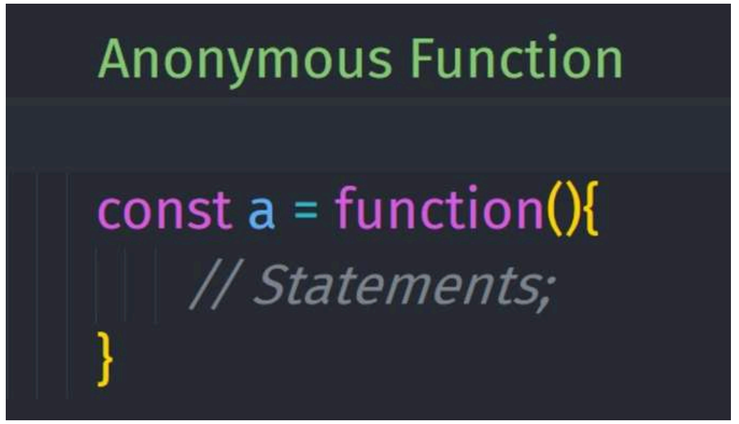
```
function somma(x, y) {
   let z = x + y;
   return z;
}

let risultato = somma(2, 6);  // 8
```

Funzioni anonime

In JavaScript è possibile definire anche **funzioni anonime**, ossia funzioni a cui non viene associato alcun nome nel momento della dichiarazione.

Funzioni anonime

Nella definizione classica di una funzione viene specificato un *nome* per poter accedere ad essa in un'altra parte dello script.
Nel caso in cui una funzione venga assegnata **ad una variabile** o **ad un evento**, invece, l'utilità del nome viene meno, dal momento che possiamo accedere ad essa tramite la *variabile* o attraverso l'*evento*.

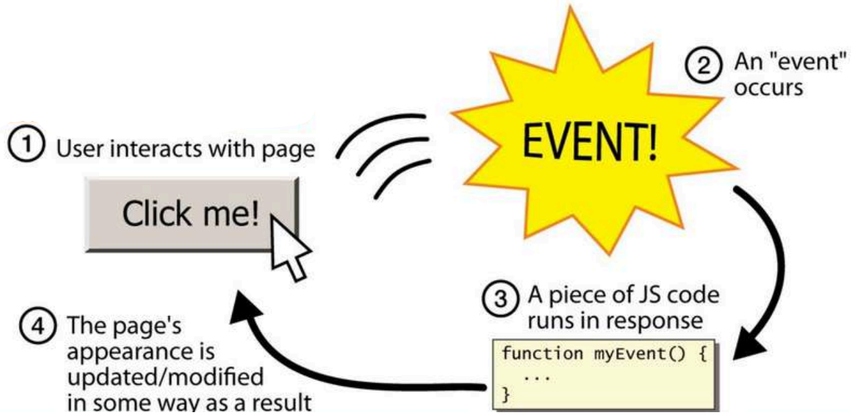


Eventi JavaScript

Un **evento** è, molto semplicemente, qualcosa che accade nella pagina.
Gli eventi possono essere "*scatenati*" dall'utente (***es.***: Click su un elemento del documento), oppure dalla situazione contingente (***es.***: La pagina è stata caricata).

Eventi JavaScript

Gestendo gli eventi, è possibile eseguire determinate istruzioni solamente quando l'utente esegue una certa azione:
quando clicca sul bottone di un <form> si controlla che i dati siano nel formato giusto, oppure quando scrolla la pagina, si gestisce il comportamento di contenuti multimediali, ad esempio il play automatico di un video.

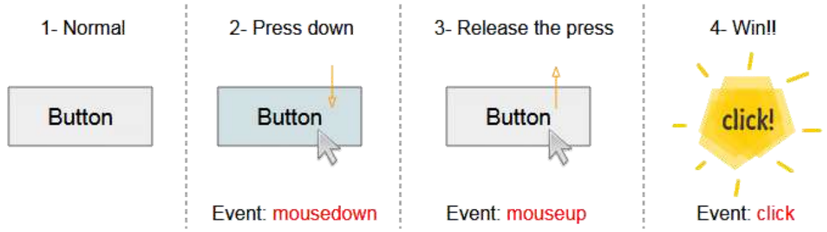


Eventi JavaScript

Ecco una lista degli eventi più utili:

Eventi del mouse

- **click** – quando si clicca col mouse su un elemento;
- **mouseover** / **mouseout** – quando il cursore passa sopra/abbandona un elemento;
- **mousedown** / **mouseup** – quando viene premuto/rilasciato il pulsante del mouse;
- **mousemove** – quando si sposta il mouse.



Eventi JavaScript

Eventi da tastiera

- **keydown** / **keyup** – quando viene premuto e rilasciato un tasto della tastiera.

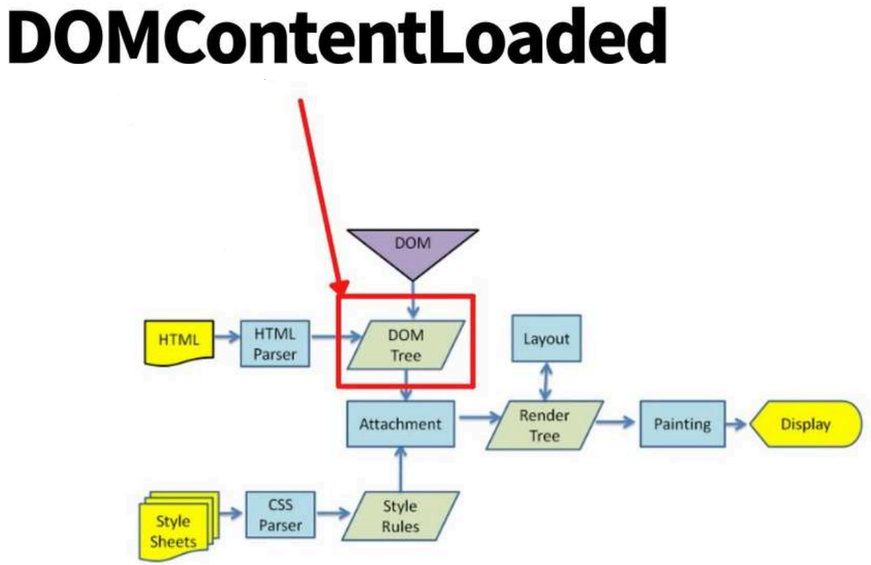
Eventi degli elementi del <form>

- **submit** – quando l'utente invia un <form>;
- **focus** – quando l'utente attiva il focus su un elemento, ad esempio su un <input>.

Eventi JavaScript

Eventi del document

- **DOMContentLoaded** – quando l'HTML viene caricato e processato, e la costruzione del DOM è stata completata.



Come gestire un evento

In JavaScript è possibile gestire gli eventi associando ad essi una *funzione*.

Come gestire un evento

Come associare una funzione ad un evento?

Ci sono 3 modi:

- tramite **markup**;
- tramite **event handler**;
- tramite **event listener**.

Come gestire un evento
Markup

In questo esempio, all'attributo **onclick** del tag <input> viene associata una funzione.
La gestione dell'evento in questo caso avviene direttamente nell'HTML.

```
<input type="button" value="Hello" onclick="alert('Hello World!');">
```

Come gestire un evento
Markup

Oggi, questa modalità non è più molto diffusa, poichè non permette di separare adeguatamente gli ambiti di lavoro.

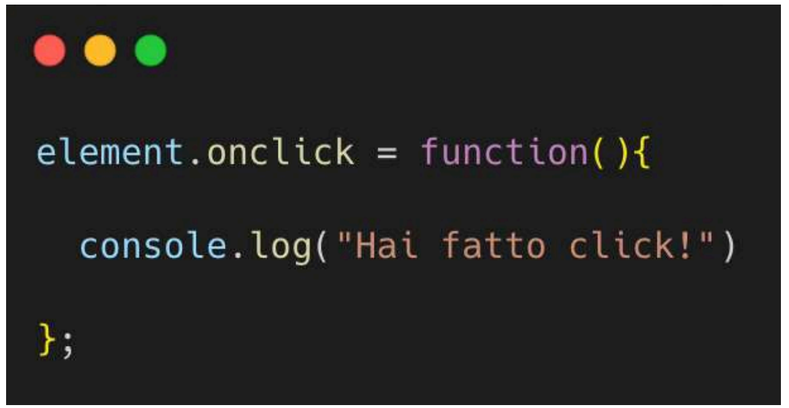
Come gestire un evento
Event Handler

Gli **event handler** (*gestori di evento*), permettono di eseguire una *funzione* contestualmente alla generazione di un evento.
Rappresentano quindi un modo per eseguire codice JavaScript al verificarsi delle azioni dell'utente.


```
element.onclick = function(){

  console.log("Hai fatto click!")

};
```

Come gestire un evento
Event Handler

In genere, queste istruzioni sono composte dalla parola "**on**", seguita dall'evento che gestiscono.
Ad esempio, l'*evento click* ha il gestore di evento **onclick**, che si attiva quando l'utente clicca su un elemento specifico.



Come gestire un evento
Event Listener

Oltre agli *event handler*, esistono anche gli **event listener** (*ascoltatori di evento*), ovvero *istruzioni* che si mettono appunto ***in ascolto*** del relativo evento, e, una volta che si è verificato, permettono di eseguire una determinata funzione.


```
document.addEventListener("DOMContentLoaded", function(){

  document.getElementById("box").innerHTML = "Hello World";

});
```

Come gestire un evento
Event Listener

Per aggiungere un *ascoltatore di evento*, è necessario utilizzare il **metodo** (*funzione*) **addEventListener()**, che accetta 2 parametri:

- il **tipo di evento** da ascoltare;
- la **funzione** da eseguire quando si verifica l'evento.

```
document.addEventListener("DOMContentLoaded", sayHello);

function sayHello() {
  document.getElementById("box").innerHTML = "Hello World!";
}
```

Come gestire un evento
Event Listener

Esistono eventi che non possono essere gestiti tramite un *event handler*, ma solo con *addEventListener()*.
Un esempio è l'evento **DOMContentLoaded**, innescato quando viene caricato il documento e costruita tutta la struttura del DOM.

```
// non viene mai eseguito
document.onDOMContentLoaded = function() {
  alert("DOM costruito");
};
```

```
// in questo modo funziona
document.addEventListener("DOMContentLoaded", function() {
  alert("DOM costruito");
});
```

onclick VS addEventListener()

onclick

L'istruzione *onclick* si attiva quando un utente clicca su un elemento HTML, come ad esempio un <button>. Può essere utilizzato per eseguire una funzione specifica o un pezzo di codice in risposta all'azione di click dell'utente.

onclick VS addEventListener()

onclick

Il limite di questa istruzione risiede nel fatto che, qualora venissero allegati più *onclick* ad uno stesso elemento, verrebbe eseguito **solo l'ultimo**, e gli altri verrebbero sovrascritti.

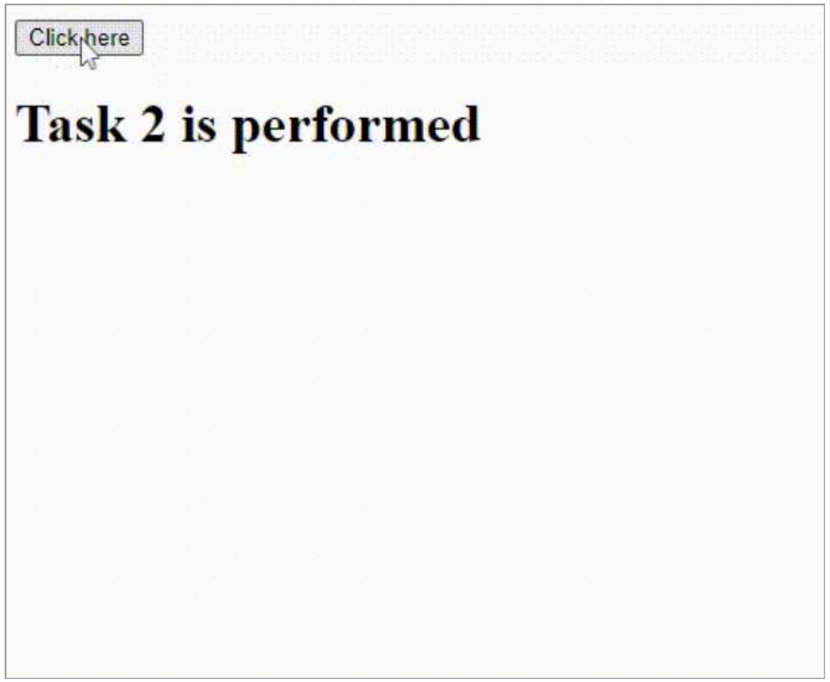
onclick VS addEventListener()

In questo esempio, vengono allegati 2 *handler di eventi* onclick allo stesso pulsante. Tuttavia, solo il secondo evento verrà eseguito, poiché il primo viene sovrascritto, quindi verrà aggiornato solo il secondo elemento <h1>.

```
<body>
    <button id="btn">Click here</button>
    <h1 id="text1"></h1>
    <h1 id="text2"></h1>

    <script>
        let btn_element = document.getElementById("btn");

        btn_element.onclick = function() {
            document.getElementById("text1").innerHTML = "Task 1 is performed";
        };

        btn_element.onclick = function() {
            document.getElementById("text2").innerHTML = "Task 2 is performed";
        };
    </script>
</body>
```

onclick VS addEventListener()

```html
<body>
    <button id="btn">Click here</button>
    <h1 id="text1"></h1>
    <h1 id="text2"></h1>

    <script>
        let btn_element = document.getElementById("btn");

        btn_element.onclick = function() {
            document.getElementById("text1").innerHTML = "Task 1 is performed";
        };

        btn_element.onclick = function() {
            document.getElementById("text2").innerHTML = "Task 2 is performed";
        };
    </script>
</body>
```



onclick VS addEventListener()

addEventListener()

La funzione *addEventListener()* invece, consente di assegnare un numero qualsiasi di eventi ad uno stesso elemento, senza sovrascrivere quelli già esistenti.

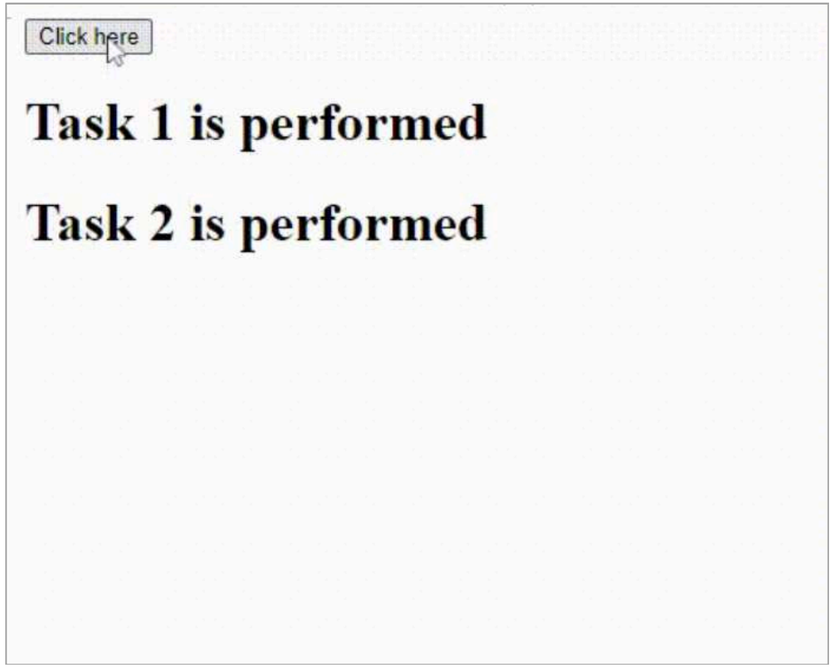
onclick VS addEventListener()

In questo esempio, vengono aggiunti 2 *listener di eventi* click allo stesso pulsante. Quando viene cliccato, **entrambe** le funzioni verranno eseguite, e quindi verrà modificato il contenuto di entrambi i tag <h1>.

```
<body>
    <button id="btn">Click here</button>
    <h1 id="text1"></h1>
    <h1 id="text2"></h1>

    <script>
        let btn_element = document.getElementById("btn");

        btn_element.addEventListener('click', function() {
            document.getElementById("text1").innerHTML = "Task 1 is performed";
        });

         btn_element.addEventListener('click', function() {
            document.getElementById("text2").innerHTML = "Task 2 is performed";
        });
    </script>
</body>
```

onclick VS addEventListener()

```html
<body>
    <button id="btn">Click here</button>
    <h1 id="text1"></h1>
    <h1 id="text2"></h1>

    <script>
        let btn_element = document.getElementById("btn");

        btn_element.addEventListener('click', function() {
            document.getElementById("text1").innerHTML = "Task 1 is performed";
        });

         btn_element.addEventListener('click', function() {
            document.getElementById("text2").innerHTML = "Task 2 is performed";
        });
    </script>
</body>
```



onclick VS addEventListener()
...Ricapitolando...

onclick	addEventListener()
Può aggiungere solo un singolo evento ad un elemento	Può aggiungere più eventi ad un elemento
Non è in grado di controllare la propagazione degli eventi	Permette di controllare la propagazione degli eventi
Può essere aggiunto anche come attributo HTML	Può essere aggiunto solo all'interno di tag <script>, o in file JavaScript esterni
È supportato da tutti i Browser	Non è supportato dalle vecchie versioni di Internet Explorer

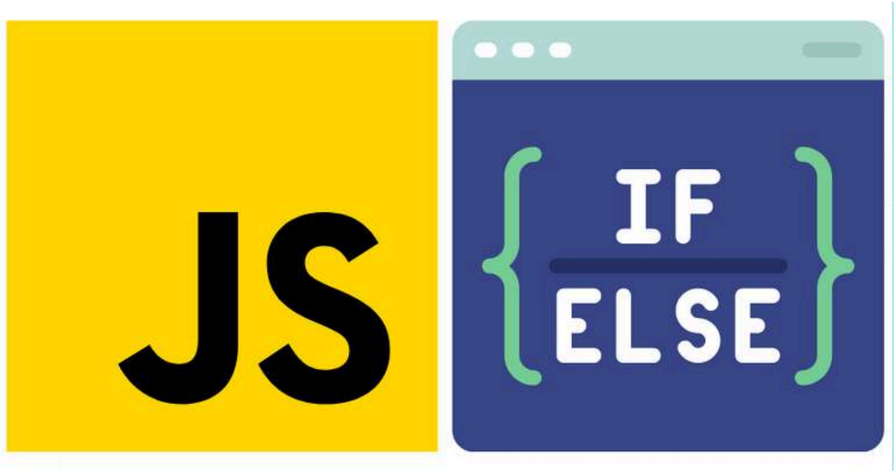
Gestori di eventi dell'Oggetto Window

Ecco alcuni degli eventi associati all'oggetto window:

- *window**.onload*** - consente di aprire una nuova finestra nel Browser;
- *window**.onfocus*** - si verifica quando la finestra è posta in primo piano;
- *window**.onblur*** - si verifica quando la finestra è posta in secondo piano;
- *window**.onerror*** - *si verifica se il documento non viene caricato nel modo corretto nella finestra .*

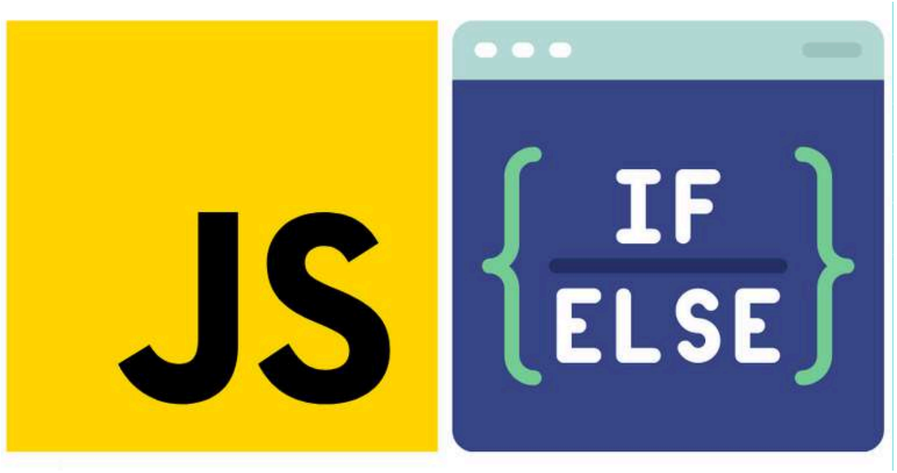
Controlli Condizionali - If Else

Talvolta occorre eseguire determinate istruzioni solo nel caso in cui si verifichino certe condizioni.
Quindi, per capire se tali condizioni siano o meno verificate, è necessario implementare un **controllo**.



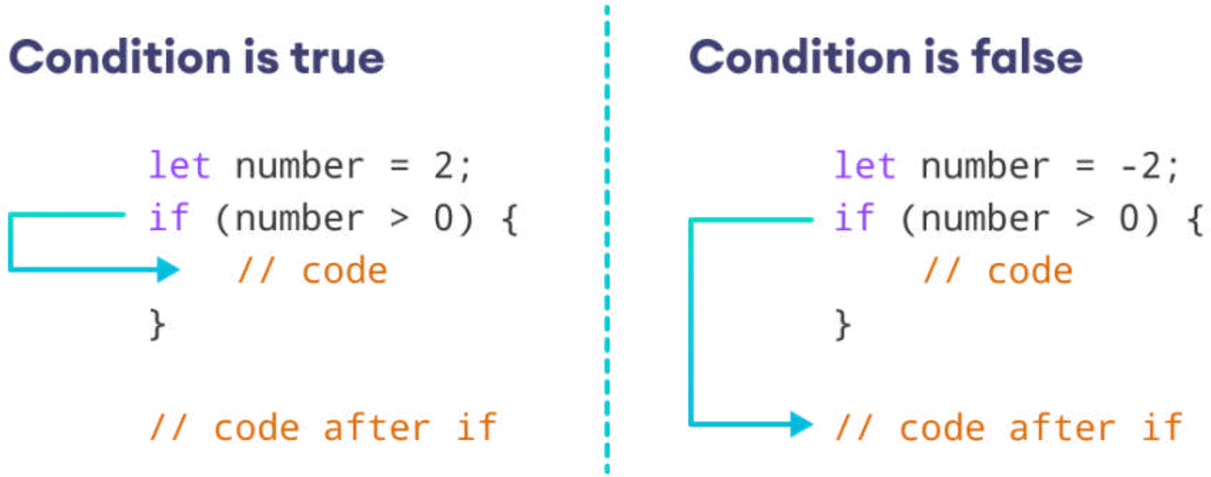
Controlli Condizionali - If Else

In JavaScript è possibile implementare dei controlli tramite l'istruzione **if()**.



Controlli Condizionali - If Else

L'istruzione *if()* valuta la condizione specificata tra le *parentesi tonde*:
se il risultato è **true**, esegue le istruzioni relative all'*if()*, altrimenti no.



Controlli Condizionali - If Else

Inoltre, l'istruzione *if()* può essere seguita da un blocco opzionale **else**, che verrà eseguito quando la condizione è **false**.

Condition is true

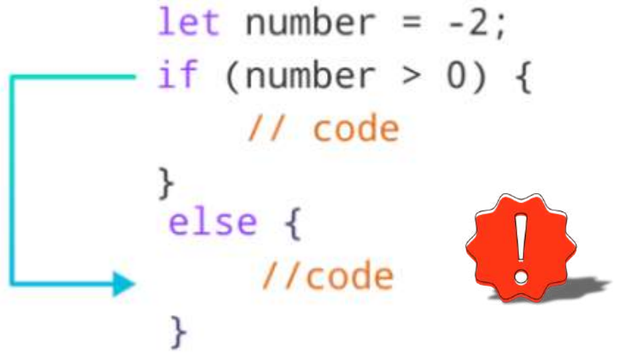
```
let number = 2;
if (number > 0) {
    // code
}
else {
    //code
}
```

Condition is false

```
let number = -2;
if (number > 0) {
    // code
}
else {
    //code
}
```

Controlli Condizionali - If Else

Utilizzando **else** non c'è bisogno di specificare tra parentesi tonde la condizione, dato che verranno valutate tutte le altre situazioni possibili, esclusa quella specificata nell'*if()*.
Quindi, se la condizione **non** si verifica, il programma eseguirà quanto specificato nel blocco *else*.



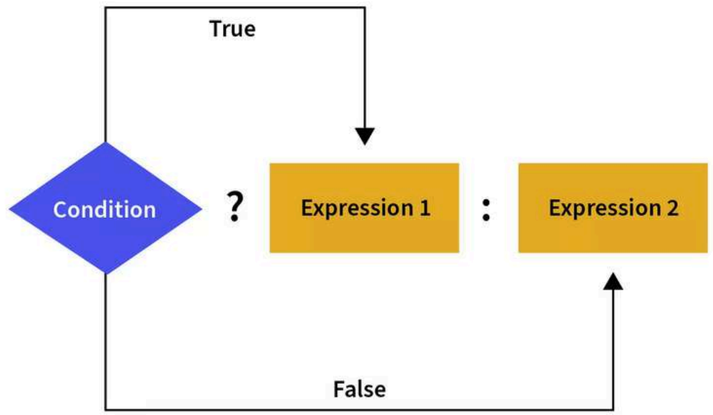
Controlli Condizionali - If Else

In questo esempio, viene intercettata la larghezza dello schermo utente e, attraverso una *struttura condizionale*, viene mostrato in console un messaggio differente, a seconda che la risoluzione lasci intendere l'utilizzo di un tablet o di un computer.

```
let screenWidth = screen.width;

if (screenWidth < 800) {
  console.log('Stai usando un tablet');
}else{
  console.log('Stai usando un computer');
}
```

Controlli Condizionali - If Else

L'istruzione *if() else{}* offre solo 2 possibilità di esecuzione: una per l'*if* ed una per l'*else*.

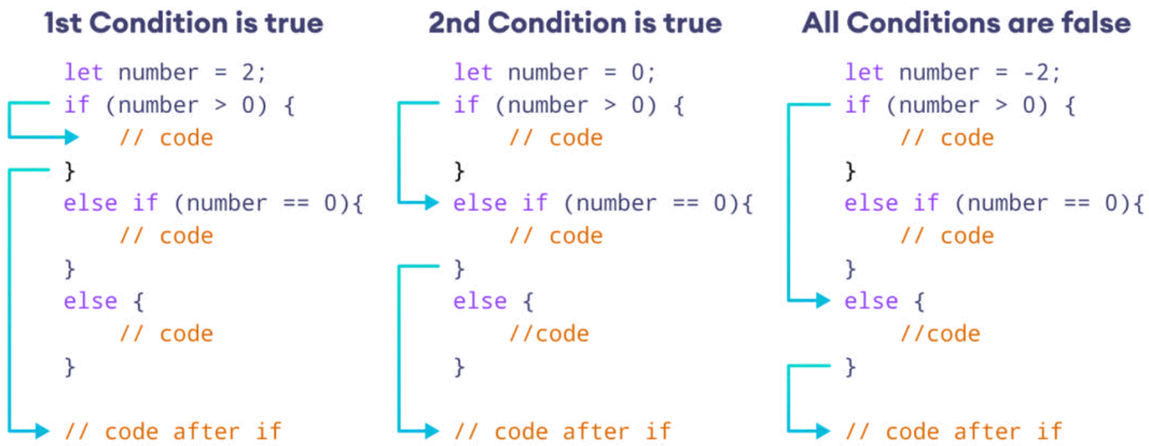


Ma qualora vi fosse necessità di valutare più di 2 condizioni?

Controlli Condizionali - If Else

In questo caso si utilizza l'istruzione **else if()**.

Sostanzialmente, se la condizione *if()* risulta **falsa**, si passa alla valutazione della condizione dell'*else if()* successiva. Questo processo continua fino a quando viene trovata una condizione **vera**, o fino a quando non si raggiunge un'istruzione *else* finale.



Controlli Condizionali - If Else

Come si può notare, il costrutto *else if()* permette di creare controlli più complessi.

```
let punteggio = 80;

if (punteggio >= 90) {
  alert('Hai ottenuto un voto eccellente!')
}else if (punteggio >= 80) {
  alert('Hai ottenuto un voto molto buono!')
}else if (punteggio >= 70) {
  alert('Hai ottenuto un voto sufficiente.')
}else {
  alert('Hai bisogno di migliorare il tuo punteggio.')
};
```

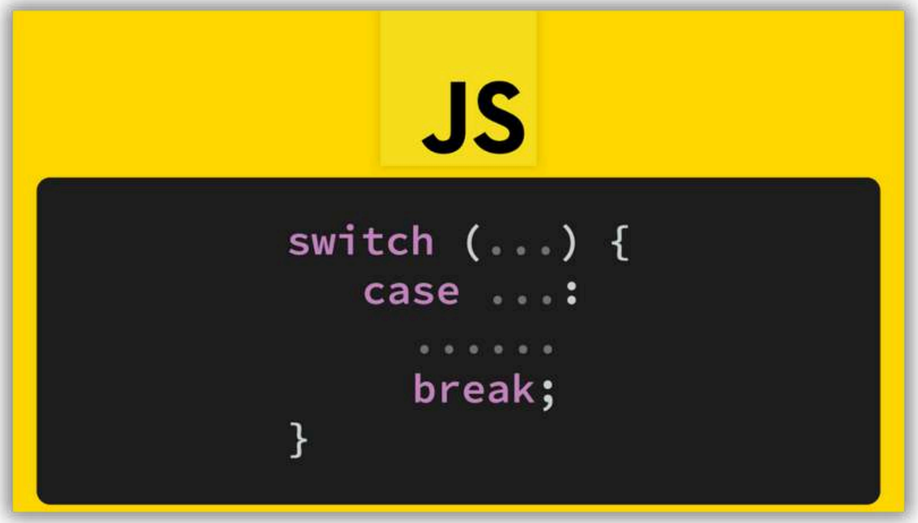
Controlli Condizionali - If Else

In altre parole, quando la condizione dell'*if()* non viene soddisfatta, il programma può passare a valutare la condizione dell'*else if()*.
Se quest'ultima è *true*, il blocco di codice associato viene eseguito, altrimenti è possibile definire ulteriori blocchi *else if()* e, se necessario, un blocco else finale per catturare tutti gli altri casi non gestiti.

```
let punteggio = 80;

if (punteggio >= 90) {
  alert('Hai ottenuto un voto eccellente!')
}else if (punteggio >= 80) {
  alert('Hai ottenuto un voto molto buono!')
}else if (punteggio >= 70) {
  alert('Hai ottenuto un voto sufficiente.')
}else {
  alert('Hai bisogno di migliorare il tuo punteggio.')
};
```

Switch Case – Un’altra struttura di controllo

In JavaScript esiste anche un altro tipo di controllo, lo **switch**, che sostanzialmente si comporta come un *if()else{}*, ma risulta più *descrittivo,* ed è particolarmente comodo quando si ha una serie di condizioni da valutare, in cui ognuna di esse corrisponde **a un valore specifico**.



Switch Case – Un’altra struttura di controllo

La sintassi è la seguente: quando viene valutata un’espressione, JavaScript cerca corrispondenze tra il valore dell’espressione e i vari ***case***.

```
switch (espressione) {
    case valore1:
        //blocco di codice da eseguire se 'espressione' corrisponde a valore1
    break;
    case valore2:
        //blocco di codice da eseguire se 'espressione' corrisponde a valore2
    break;
    case valore3:
        //blocco di codice da eseguire se 'espressione' corrisponde a valore3
    break;
    default:
        //blocco di codice da eseguire se nessuna delle precedenti
        condizioni è verificata
}
```

Switch Case – Un’altra struttura di controllo

Se viene trovata una corrispondenza, il blocco di codice associato a quel *case* viene eseguito.

```
switch (espressione) {
    case valore1:
        //blocco di codice da eseguire se 'espressione' corrisponde a valore1
    break;
    case valore2:
        //blocco di codice da eseguire se 'espressione' corrisponde a valore2
    break;
    case valore3:
        //blocco di codice da eseguire se 'espressione' corrisponde a valore3
    break;
    default:
        //blocco di codice da eseguire se nessuna delle precedenti
        condizioni è verificata
}
```

Switch Case – Un’altra struttura di controllo

Dopo l’esecuzione di un *case*, è importante utilizzare l’istruzione **break** per uscire dallo *switch* ed evitare che vengano eseguiti anche i blocchi dei *casi* successivi.

```
switch (espressione) {
    case valore1:
        //blocco di codice da eseguire se 'espressione' corrisponde a valore1
    break;
    case valore2:
        //blocco di codice da eseguire se 'espressione' corrisponde a valore2
    break;
    case valore3:
        //blocco di codice da eseguire se 'espressione' corrisponde a valore3
    break;
    default:
        //blocco di codice da eseguire se nessuna delle precedenti
        condizioni è verificata
}
```

Switch Case – Un’altra struttura di controllo

Se nessun caso corrisponde al valore dell’espressione, viene eseguito il blocco di codice definito all’interno dell’istruzione **default** (*opzionale*).

```
switch (espressione) {
    case valore1:
        //blocco di codice da eseguire se 'espressione' corrisponde a valore1
    break;
    case valore2:
        //blocco di codice da eseguire se 'espressione' corrisponde a valore2
    break;
    case valore3:
        //blocco di codice da eseguire se 'espressione' corrisponde a valore3
    break;
    default:
        //blocco di codice da eseguire se nessuna delle precedenti
        condizioni è verificata
}
```

Switch Case – Un’altra struttura di controllo

L’istruzione **default** viene utilizzata per gestire tutti i *casi* non contemplati precedentemente.

```
switch (espressione) {
    case valore1:
        //blocco di codice da eseguire se 'espressione' corrisponde a valore1
    break;
    case valore2:
        //blocco di codice da eseguire se 'espressione' corrisponde a valore2
    break;
    case valore3:
        //blocco di codice da eseguire se 'espressione' corrisponde a valore3
    break;
    default:
        //blocco di codice da eseguire se nessuna delle precedenti
        condizioni è verificata
}
```

Switch Case – Un’altra struttura di controllo

In questo esempio, la variabile *“giorno”* ha valore "*Lunedì*".
L'istruzione *switch* controlla questo valore ed esegue il blocco di codice che corrisponde al *case*: in questo caso stamperà "*È Lunedì!*" in console.

```
let giorno = "Lunedì";

switch(giorno) {
  case "Lunedì":
    console.log("È Lunedì!");
    break;
  case "Martedì":
    console.log("È Martedì!");
    break;
  case "Mercoledì":
    console.log("È Mercoledì!");
    break;
  default:
    console.log("È un altro giorno della settimana.");
    break;
}
```

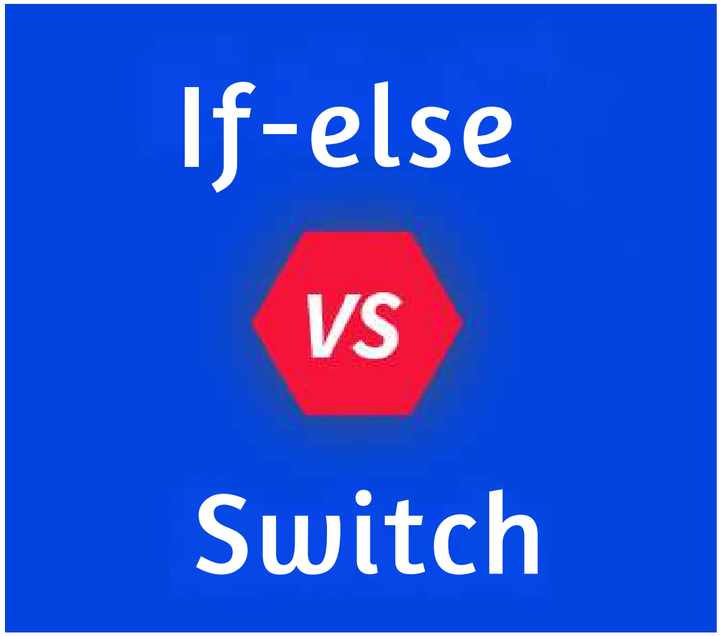
Switch Case - Un'altra struttura di controllo

Per implementare questo controllo si può utilizzare anche un classico **if() else{}**.
Il risultato è il medesimo, ma a livello di codice probabilmente lo **switch-case** è più schematico ed ordinato.

```
let giorno = "Lunedì";

if (giorno == "Lunedì") {
    console.log("È Lunedì!");
}else if (giorno == "Martedì") {
    console.log("È Martedì!");
}else if (giorno == "Mercoledì") {
    console.log("È Mercoledì");
}else{
    console.log("È un altro giorno della settimana");
}
```

Switch Case - Un'altra struttura di controllo

Un'altra differenza tra queste due strutture è che con l'if() è possibile valutare condizioni complesse, mentre con lo switch è possibile valutare solamente valori costanti.



I Cicli

Cos’è un Ciclo?

È una struttura che permette di eseguire delle istruzioni **un certo numero di volte**. Il numero di “*giri*” che verranno effettuati dal Ciclo è determinato da una **condizione**.
Le condizioni in genere restituiscono **true** o **false**: il ciclo continuerà ad iterare fino a quando la condizione ritorna *false*.

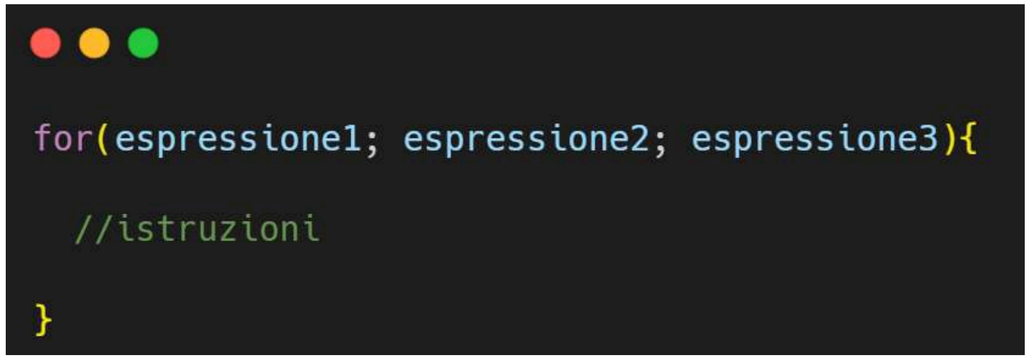
Ciclo For

Ciclo For

È la struttura *ciclica* principale di JavaScript, e la sua sintassi è raffigurata in quest'immagine: come si può notare, è costituito da **3 espressioni**.

```
for(espressione1; espressione2; espressione3){

  //istruzioni

}
```

Ciclo For



- **INIZIALIZZAZIONE** - viene eseguita ***prima*** dell'esecuzione del ciclo ed è usata per creare un "*contatore*".
- **CONDIZIONE** - viene controllata ***ogni volta prima*** dell'esecuzione del ciclo. Se restituisce *true*, le istruzioni del ciclo vengono eseguite, se restituisce *false*, il ciclo si interrompe.
- **INCREMENTO** - viene eseguita ***dopo*** ogni iterazione del ciclo ed è usata per incrementare (o decrementare) il contatore.

Ciclo For Infinito

Se la condizione (*espressione2*), è **sempre vera**, allora le istruzioni all'interno delle parentesi graffe saranno eseguite **all'infinito**. In questo caso si parla di *Ciclo Infinito*.

```
for(espressione1; espressione2; espressione3){

  //istruzioni

}
```

Ciclo For Infinito

Per “*infinito*” si intende un Ciclo la cui *condizione di fine* non potrà mai verificarsi. In tal caso il Ciclo andrà avanti all'infinito e l'unico modo di fermarlo sarà quello di chiudere forzatamente il programma.

```
for(espressione1; espressione2; espressione3){

  //istruzioni

}
```

Ciclo For Infinito

Un esempio di Ciclo Infinito può essere questo in figura:
mancando infatti l'espressione di ***incremento*** della *variabile contatore*, quest'ultima rimane sempre uguale a 5, e non potrà mai soddisfare la *condizione di fine ciclo*, che richiede invece che essa diventi uguale a 20.

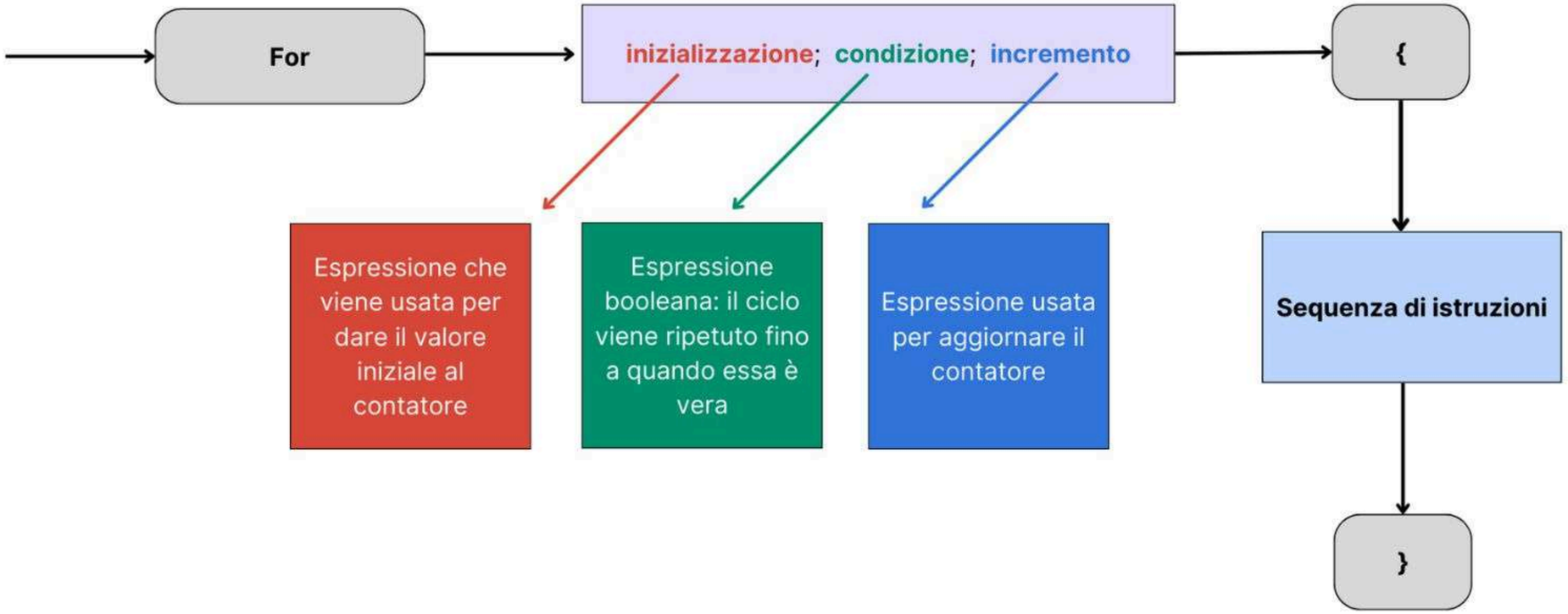
```
for(let i=5; i <= 20;){
    alert("Ciao!");
}
```

Ciclo For Infinito

Un Ciclo del genere continuerebbe a mostrare all'infinito una finestra di dialogo (alert()) con il messaggio “Ciao!”.

```
for(let i=5; i <= 20;){
    alert("Ciao!");
}
```

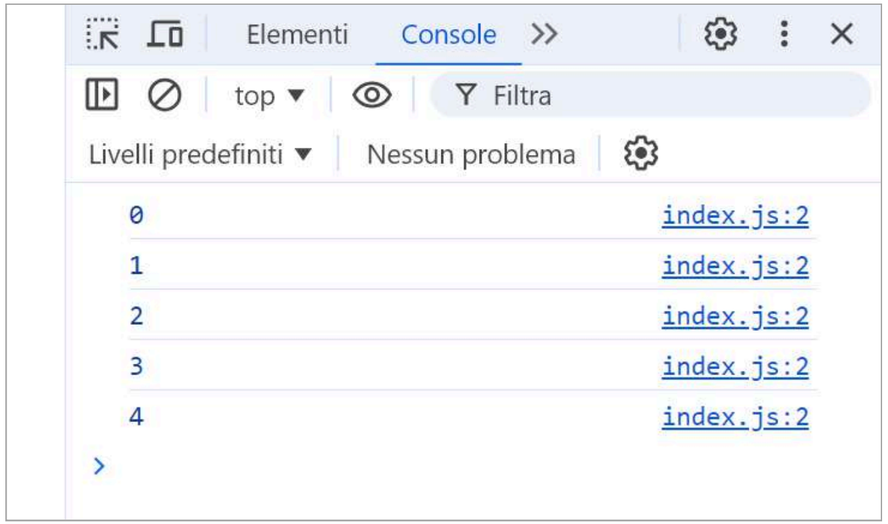
Ciclo For


For
inizializzazione; condizione; incremento
{
Espressione che viene usata per dare il valore iniziale al contatore
Espressione booleana: il ciclo viene ripetuto fino a quando essa è vera
Espressione usata per aggiornare il contatore
Sequenza di istruzioni
}

Ciclo For

In questo esempio, come prima cosa viene inizializzata a 0 la *variabile contatore* **i**; dopodichè viene valutata la condizione **i < 5**, e, finchè torna *true*, viene eseguito il *console.log()*.
Dopo ogni esecuzione, la variabile contatore viene incrementata di un'unità, in questo modo il Ciclo può continuare ad iterare.

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



Ciclo For

Il Ciclo For è molto utile anche quando si vuole iterare su una *sequenza di elementi*, come ad esempio gli **elementi di un Array**.
In parole povere, si possono eseguire le istruzioni contenute nel Ciclo su *ogni singolo elemento* di un'Array.

Ciclo For

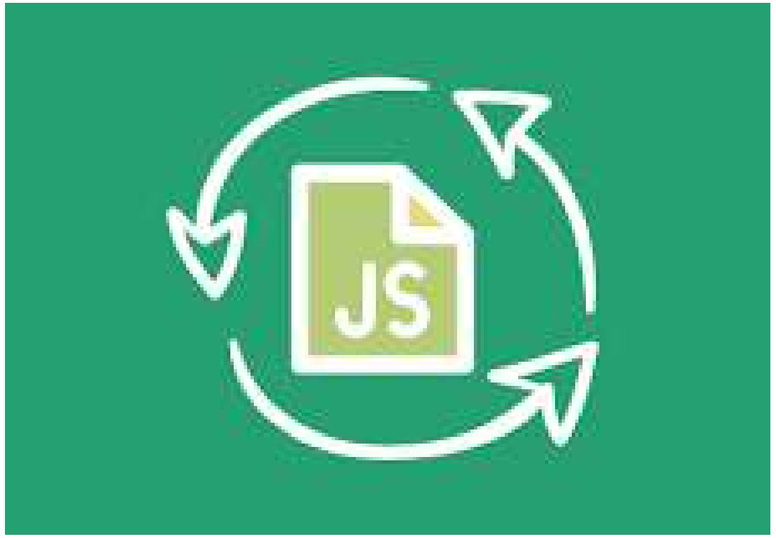
Per farlo si utilizza la proprietà ***length*** dell'Array e la ***posizione*** che ogni elemento occupa al suo interno.

Nell'esempio a fianco, l'istruzione ***console.log()*** verrà eseguita *4 volte* ed in console verranno stampate tutte e 4 le stringhe contenute nell'Array "*stringhe*".

```
const stringhe = ["abc", "def", "ghi", "test"];

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

...Ricapitolando...

Il Ciclo For in JavaScript è utilizzato quando è necessario eseguire delle istruzioni un numero specifico di volte, o quando si vuole iterare su una sequenza di elementi, come ad esempio gli elementi di un Array.



...Ricapitolando...

I Cicli hanno una grande importanza in programmazione: grazie ad essi è possibile eseguire, per mezzo **di poche righe di codice**, un numero potenzialmente infinito di istruzioni, che altrimenti richiederebbero, a seconda dei casi, decine, centinaia o addirittura migliaia di righe di codice.
Sostanzialmente si tratta di "dire" al Browser di eseguire determinate istruzioni fino a che non si verifica una certa condizione. In questo modo è possibile scrivere ciascuna istruzione **una sola volta**, indicando poi per quante volte o fino a quando tali istruzioni dovranno essere eseguite.

Ciclo While

Un’altra struttura ciclica è rappresentata dal **while**, la cui sintassi è la seguente:

```
while (condizione) {
    // istruzioni
}
```

Ciclo While

Finché “*condizione*” tornerà **true,** verranno eseguite le istruzioni contenute nel blocco di codice. Requisito fondamentale nell'uso del **while** è che le istruzioni contenute nel blocco di codice modifichino la condizione, altrimenti si rischia di incorrere in un ciclo infinito.

```
while (condizione) {
    // istruzioni
}
```

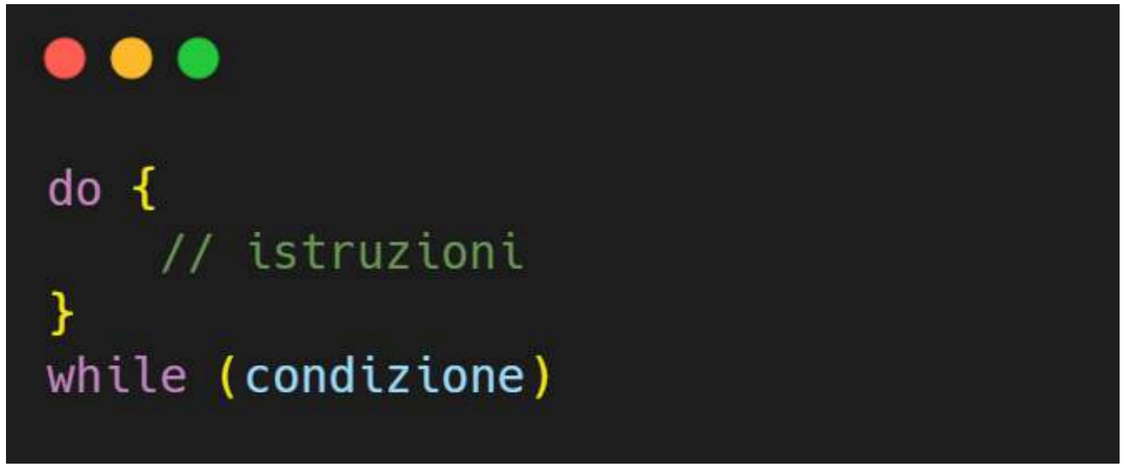
Ciclo While

Questo è un esempio di utilizzo dell'istruzione **while**: da notare come l'incremento della variabile **x** garantisca l'uscita dal ciclo, dal momento che al superamento del valore 9 renderà la condizione **false**.

```
let x = 0;

while (x < 10){
    x = x + 1;
    console.log("variabile x");
}

console.log("fine");
```

Ciclo Do While

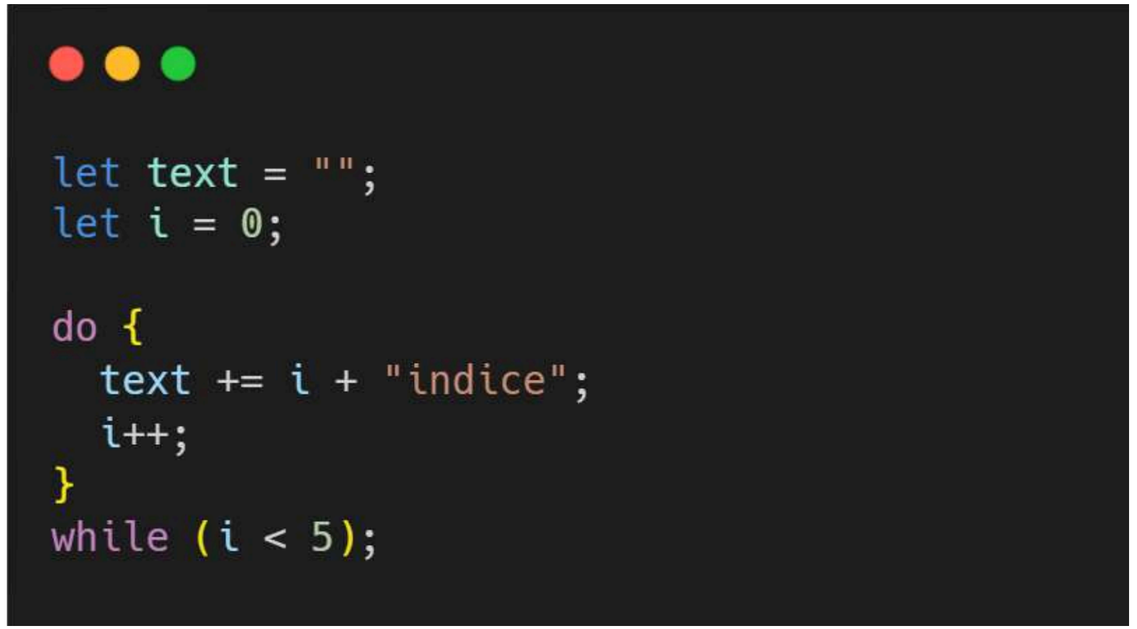
Una variante del *while* è il **do...while**: la differenza sostanziale tra queste 2 strutture consiste nel fatto che la *condizione di fine* ciclo viene valutata **dopo** aver eseguito le istruzioni.



```
do {
    // istruzioni
}
while (condizione)
```

Ciclo Do While

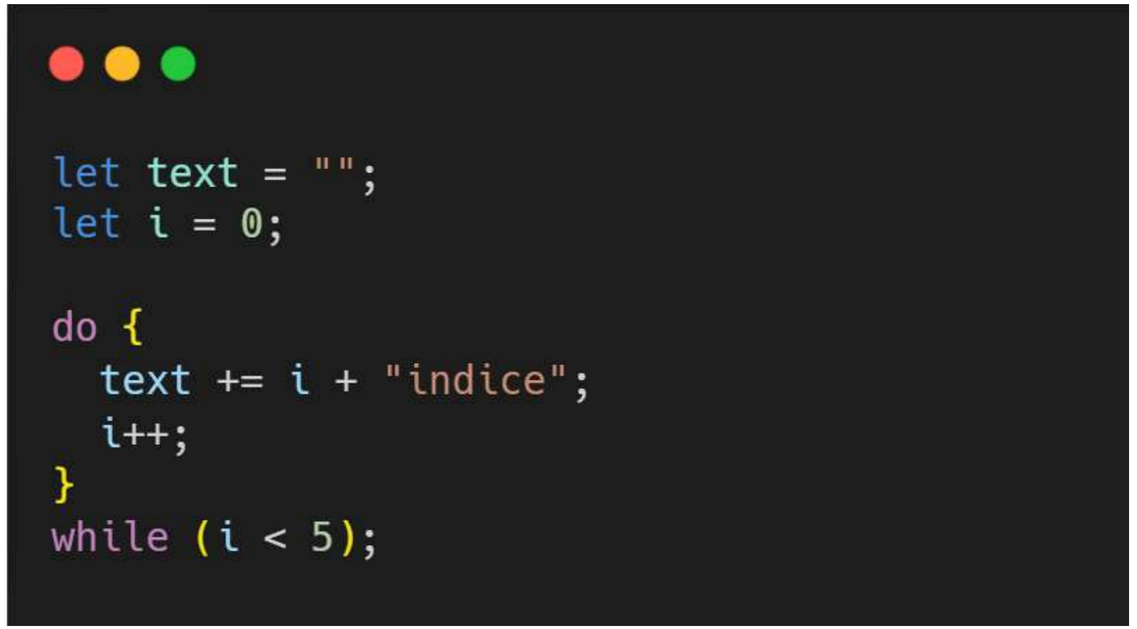
Questo è un esempio di utilizzo dell'istruzione **do...while**: da notare come l'incremento della variabile *i* garantisca l'uscita dal ciclo, dal momento che al superamento del valore 4 renderà la condizione **false**.



```
let text = "";
let i = 0;

do {
  text += i + "indice";
  i++;
}
while (i < 5);
```

Ciclo Do While

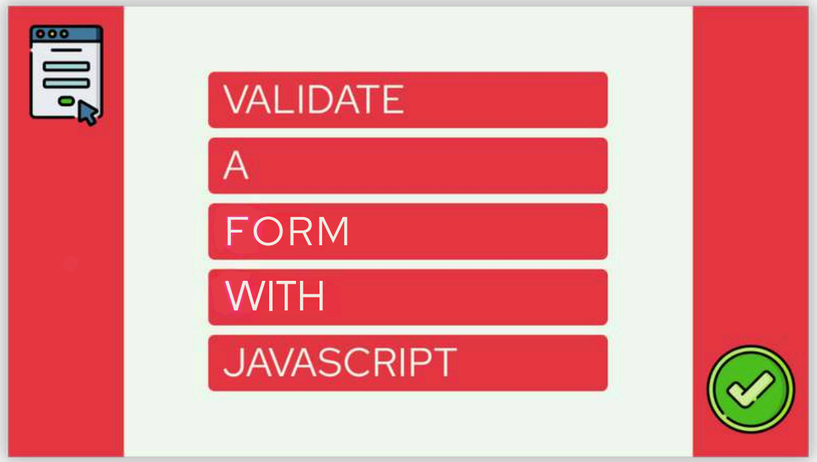
Questo è un esempio di utilizzo dell'istruzione **do...while**: da notare come l'incremento della variabile *i* garantisca l'uscita dal ciclo, dal momento che al superamento del valore 4 renderà la condizione **false**.



```
let text = "";
let i = 0;

do {
  text += i + "indice";
  i++;
}
while (i < 5);
```

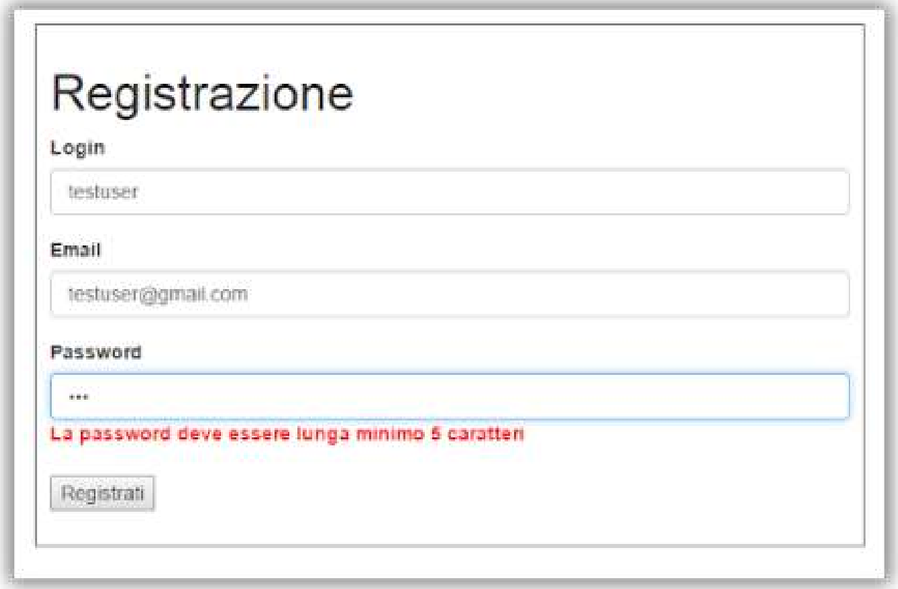
Validazione Dati

Uno degli usi classici di JavaScript è la **validazione dei dati** inseriti dall'utente in un <form> prima dell'invio al server, o comunque prima della loro elaborazione.



Validazione Dati

Verificare che un valore *obbligatorio* sia stato effettivamente fornito dall'utente, e che sia correttamente *formattato*, consente di evitare errori talvolta critici in fase di elaborazione dei dati.



Validazione Dati
Come implementare il controllo di un <form>

Quando l'utente compila un <form>, i dati che inserisce vengono spediti dopo il *click* su un ***<button type="submit">***.

Quest'azione scatena l'evento **submit**.

Validazione Dati
Come implementare il controllo di un <form>

Dal momento che i dati, prima di poter essere inviati al server, devono essere verificati, occorre gestire questo evento ed implementare tutti i controlli necessari proprio all'interno della funzione ad esso associata.

```html
<form id="form">
  <label for="firstname"> First Name* </label>
  <input type="text" name="firstname" id="firstname">
  <button type="submit">Submit</button>

  <span role="alert" id="nameError">
    Please enter First Name
  <span>
</form>
```

```js
const form = document.getElementById("submit");

form.addEventListener("submit", function(){

  //istruzioni di controllo

});
```

Validazione Dati
Come implementare il controllo di un <form>

In questo esempio, viene controllato se il campo con *id="firstname"* sia stato compilato o meno (nella pratica si verifica se il suo ***value*** è una stringa vuota).

```
const form = document.getElementById("submit");

form.addEventListener("submit", function(e){
    e.preventDefault();

    const firstNameField = document.getElementById("firstname");

    if (firstNameField.value == "") {
        const nameError = document.getElementById("nameError");
        nameError.style.display = "block";
        firstNameField.classList.add("invalid");
    }

});
```

Validazione Dati
Come implementare il controllo di un <form>

Nella funzione associata all'evento *submit* è stato aggiunto il **parametro "e"**, che rappresenta l'evento in questione, e che quindi contiene tutte le informazioni ed i dati ad esso associati.

```
const form = document.getElementById("submit");

form.addEventListener("submit", function(e){
    e.preventDefault();

    const firstNameField = document.getElementById("firstname");

    if (firstNameField.value == "") {
        const nameError = document.getElementById("nameError");
        nameError.style.display = "block";
        firstNameField.classList.add("invalid");
    }

});
```

Validazione Dati
Come implementare il controllo di un <form>

In particolare è stata definita l'istruzione ***e.preventDefault()***, che permette di bloccare il comportamento di default del tag <form> (infatti, non appena l'utente clicca sul bottone di *submit*, i dati vengono spediti).

```
const form = document.getElementById("submit");

form.addEventListener("submit", function(e){
    e.preventDefault();

    const firstNameField = document.getElementById("firstname");

    if (firstNameField.value == "") {
        const nameError = document.getElementById("nameError");
        nameError.style.display = "block";
        firstNameField.classList.add("invalid");
    }

});
```

Validazione Dati
Come implementare il controllo di un <form>

Dopodichè si procede con un controllo **if()**, che verifica se il *value* dell'<input *id="firstname"*> è una stringa vuota, e se così fosse, verrà mostrato il relativo messaggio di errore (*nameError*).

```
const form = document.getElementById("submit");

form.addEventListener("submit", function(e){
    e.preventDefault();

    const firstNameField = document.getElementById("firstname");

    if (firstNameField.value == "") {
        const nameError = document.getElementById("nameError");
        nameError.style.display = "block";
        firstNameField.classList.add("invalid");
    }

});
```

Validazione Dati
Eventi legati al controllo dati

Oltre all'evento *submit*, esistono altri eventi legati alla compilazione di dati utente.

Tra i più utilizzati troviamo:

- evento **focus** - un elemento riceve il *focus* sia quando l'utente ci clicca sopra, sia quando usa il tasto "*Tab*" della tastiera;
- evento **blur** - si verifica quando un controllo perde il *focus*;
- evento **change**- si verifica quando l'utente modifica il valore di un controllo.

Validazione Dati
Regular Expressions

Cosa sono le Regular Expressions?

Le **espressioni regolari** (**regex**) sono strumenti di *pattern matching*, ovvero sequenze di caratteri usate per identificare e manipolare stringhe complesse.
Permettono quindi di analizzare stringhe di testo.

Validazione Dati
Regular Expressions

Cosa sono le Regular Expressions?

Riassumendo, una possibile definizione di **regex** potrebbe essere la seguente: *"una regex è una stringa che identifica un insieme di stringhe"*.

JavaScript Regular expression validation

Username validation Regex

```
/^[a-z][a-z0-9]{4,20}$/;
```

URL Validation Regex

```
/^((https?:\/\/(www\.)?|\.)[a-zA-Z0-9]
[\w+\d+\-\/%?:,;+1)5/gi;
```

Email Address Validation Regex

```
/^([a-28-91#$%&*+\-/-(1)~]+(?:\.[a-zA-Z0-91
#$%&*+\-/?^(1)~]+)*)&((?:\[a-z0-9]+(?:[8-2-8-
9-]*)\.)+[a-z]{2,})5/g1;
```

Validazione Dati
Regular Expressions

Per questo motivo, le *regex* vengono spesso impiegate per *trovare e sostituire del testo*, *verificare che un input di dati corrisponda al formato richiesto* e altre cose simili.

JavaScript Regular expression validation

Username validation Regex

```
/^[a-z][a-z0-9]{4,20}$/;
```

URL Validation Regex

```
/^((https?:\/\/(www\.)?|\.)[a-zA-Z0-9]
[\w+\d+\-\/%?:,;+1)5/gi;
```

Email Address Validation Regex

```
/^([a-28-91#$%&*+\-/-(1)~]+(?:\.[a-zA-Z0-91
#$%&*+\-/?^(1)~]+)*)&((?:\[a-z0-9]+(?:[8-2-8-
9-]*)\.)+[a-z]{2,})5/g1;
```

Regular Expressions
Sintassi delle RegEx

Sicuramente le *Regular Expressions* hanno una forma che al primo sguardo appare complessa e criptica, ma una volta acquisita familiarità con i loro componenti di base, diventano strumenti estremamente potenti e versatili.

JavaScript Regular expression validation

Username validation Regex

```
/^[a-z][a-z0-9]{4,20}$/;
```

URL Validation Regex

```
/^((https?:\/\/(www\.)?|\.)[a-zA-Z0-9]
[\w+\d+\-\/%?:,;+1)5/gi;
```

Email Address Validation Regex

```
/^([a-28-91#$%&*+\-/-(1)~]+(?:\.[a-zA-Z0-91
#$%&*+\-/?^(1)~]+)*)&((?:\[a-z0-9]+(?:[8-2-8-
9-]*)\.)+[a-z]{2,})5/g1;
```

Regular Expressions
Sintassi delle RegEx

Una regex include una **combinazione di testo** ed un insieme di **operatori** che agiscono come caratteri “jolly” per ricercare una corrispondenza del pattern.

JavaScript Regular expression validation

Username validation Regex

```
/^[a-z][a-z0-9]{4,20}$/;
```

URL Validation Regex

```
/^((https?:\/\/(www\.)?|\.)[a-zA-Z0-9]
[\w+\d+\-\/%?:,;+1)5/gi;
```

Email Address Validation Regex

```
/^([a-28-91#$%&*+\-/-(1)~]+(?:\.[a-zA-Z0-91
#$%&*+\-/?^(1)~]+)*)&((?:\[a-z0-9]+(?:[8-2-8-
9-]*)\.)+[a-z]{2,})5/g1;
```

Regular Expressions
Sintassi delle RegEx

In termini pratici, una regex può includere un *singolo carattere*, *una corrispondenza per uno o più caratteri*, una *corrispondenza per zero o più caratteri*, *caratteri facoltativi*, ...
In questo modo, è possibile costruire un'espressione complessa che permette di ottenere risultati di vasta portata, ma molto specifici.

JavaScript Regular expression validation

Username validation Regex
```
/^[a-z][a-z0-9]{4,20}$/;
```

URL Validation Regex
```
/^((https?:\/\/(www\.)?|\.)[a-zA-Z0-9]
[\w+\d+\-\/%?:,;+1)5/gi;
```

Email Address Validation Regex
```
/^([a-28-91#$%&*+\-/-(1)~]+(?:\.[a-zA-Z0-91
#$%&*+\-/?^(1)~]+)*)&((?:\[a-z0-9]+(?:[8-2-8-
9-]*)\.)+[a-z]{2,})5/g1;
```

Regular Expressions
Sintassi delle RegEx

Gli operatori comuni delle RegEx

Ogni elemento delle regex ha una funzione specifica;

i **metacaratteri** sono i componenti fondamentali, e includono simboli come:

. (*punto*) ***** (*asterisco*) **+** (*più*)
? (*punto interrogativo*) **** (*backslash*)

Regular Expressions
Sintassi delle RegEx

Gli operatori comuni delle espressioni regolari

- punto (**.**) - è un carattere jolly, che può rappresentare ogni singolo carattere;
- L'asterisco (*****) - seleziona una corrispondenza per zero o più elementi;
- Il più (**+**) - seleziona una corrispondenza **per uno o più** elementi;
- Il punto interrogativo (**?**) - rende il carattere precedente una **parte facoltativa** dell'espressione;

Regular Expressions
Sintassi delle RegEx

Gli operatori comuni delle espressioni regolari

- **digit** o **d** - imposta una corrispondenza per ogni cifra numerica (0-9);
- Pipe (**|**) - indica una funzione di opposizione (**OR**);
- Accento circonflesso (**^**) - si usa per denotare **l'inizio** di una stringa;
- Dollaro (**$**) - serve a denotare **la fine** di una stringa.;
- Il backslash (****) - se inserito prima di un operatore/di un carattere vale come "**escape**", ovvero consente di escludere un carattere speciale.

Regular Expressions
Sintassi delle RegEx

Supponiamo di voler controllare la *formattazione* di un indirizzo email, che di norma ha una forma del tipo:

username@domain.com

Regular Expressions
Sintassi delle RegEx

La regex per identificare un *indirizzo email* potrebbe essere qualcosa di questo genere:


```
\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b
```

Regular Expressions
Sintassi delle RegEx

\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b

- **\b** - indica un confine di parola, assicurando che l'indirizzo email sia una parola completa;

- **[A-Za-z0-9._%+-]+** - corrisponde alla parte ***username****;* è una *classe di caratteri* che include lettere maiuscole e minuscole, numeri, e alcuni caratteri speciali. Il **+** indica che ci deve essere **almeno un carattere**;

- **@** - è il simbolo della chiocciola, che deve essere presente in ogni indirizzo email;

Regular Expressions
Sintassi delle RegEx

\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b

- **[A-Za-z0-9.-]+** - corrisponde alla parte ***domain***, ed è simile alla parte *username*. Questa *classe di caratteri* include lettere, numeri, punti e trattini;
- **\.** - corrisponde a un punto letterale, necessario per separare il dominio dall'estensione;
- **[A-Z|a-z]{2,}** - corrisponde all'estensione del dominio, che deve essere composta da **almeno 2**lettere;
- **\b** - indica un altro confine di parola.

Labforweb
nerd academy

GRAZIE