# JAVASCRIPT ES6

* Backward-compatibile: tutto ciò che funziona, funzionerà in futuro
* Non è forward compatibile: ciò che viene creato, non è detto che funzionerà
* **Modalità Strict-Mode**: disabilita alcune funzionalità molto vecchie.

File parsificato e poi eseguito riga per riga dall’inizio alla fine.

*API per web non funzionano per node. Node può modificare file sul disco mentre browser no.*

Alcune note:

* È **Unicode** quindi posso mettere qualsiasi cosa nelle variabili.
* Devo mettere il punto e virgola alla fine. **;**
* **Nomi** devono iniziare con lettere o $ o \_
* Per usare strict mode **“use strict”**

## TIPI

In javascript le variabili non hanno tipi ma i valori sì.

* Immagine che contiene testo, schermata, Rettangolo, Carattere

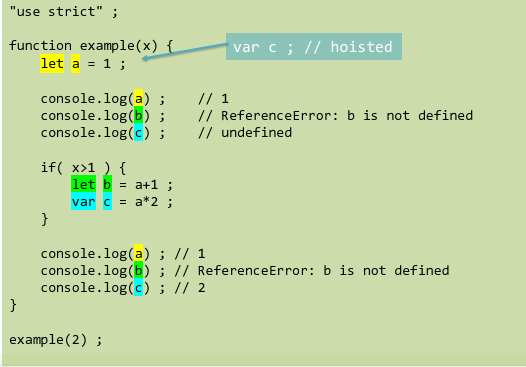
  Descrizione generata automaticamente**String**:
  + “ ” ‘ ’ normali
  + `` stringLiteral permette di andare a capo liberamente all’interno del testo
* **Number**:
  + no differenza tra interi e floating
  + NaN nel caso in cui il risultato di un’operazione numerica non sia un numero

NOTA: NaN non è uguale a NaN

* **Boolean**: bisogna far attenzione alle cose
  + falsi: 0, -0, NaN, undefined, null, ‘’
  + veri: qualsiasi cosa non sia sopra
  + = = : converte tipi e compara risultato
  + = = = : compara tipo e risultato
* **Null**: valore vuoto per una certa variabile
* **Undefined**: valore dichiarato ma non ancora inizializzato.

## VARIABILI

* Immagine che contiene testo, schermata, Carattere, linea

  Descrizione generata automaticamente**let**: variabile di qualunque tipo che si può riassegnare ma non posso ridichiarare
  + *let a=5;*
* **const**: variabile di qualunque tipo che non posso ridichiarare/riassegnare
  + *const b=6;*
* **var**: variabile di qualunque tipo che si può ridichiarare e riassegnare
  + *var c=2;*
  + ***HOSTING****: con var posso usare una variabile prima di dichiararla*

**scope**: {}/funzioni

*const e let esistono e valgono solo nello scope in cui loro esistono.*

## ESPRESSIONI

### Assegnazione

* *Assegnazione* x=y *Addizione* x+=y *Sottrazione* x-=y
* *Moltiplicazione* x\*=y *Divisione* x/=y *Resto* x%=y
* *Esponenziale* x\*\*=y
* *Left shift* x≪=y *Right shift* x>>=y *Unsigned Right Shift* x>>>=y
* *Bitwise AND*  x&=y *Bitwise XOR* x^=y *Bitwise OR* x|=y

Confrontodue tipologie di confronto:

* convertono l’oggetto in modo tale da confrontare solo valore:
  + = =

**Conversioni**

Immagine che contiene testo, schermata, Carattere, diagramma

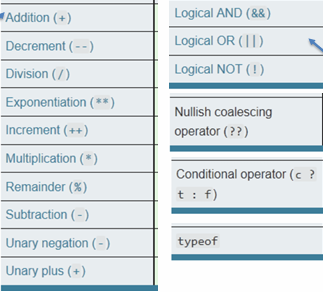
Descrizione generata automaticamente

* + ! =
  + >
  + >=
  + <
  + < =
* non convertono tipo:
  + = = =
  + ! = =

### OPERATORI LOGICI:

* expr1 **&&** expr2
  + ritorna expr1 se può essere convertito in falso, altrimenti expr2
  + se usata con booleani, true se entrambi sono true
* expr1 **||** expr2
  + ritorna expr1 se può essere convertito in true, altrimenti expr2
  + se usata con booleani, true se almeno uno true
* **!**expr
  + Ritorna falso se expr può essere convertito in true

### OPERATORI MATEMATICI:

* + : addizione o concatenazione di stringhe
* / divisione
* \*\* esponenziale
* \* moltiplicazione
* % resto
* – sottrazione
* c ? t: f
* a || b : se a è vero, vado avanti con a; altrimenti vado avanti con b

## esercizio Strange.js

/\* Strange JS behaviors and where to find (some of) them \*/

'use strict';

const type = typeof NaN;

console.log('NaN is a ' + type); // number

console.log(`NaN === NaN? ${NaN === NaN}\n`); // FALSE

console.log(`NaN == NaN? ${NaN == NaN}`); // FALSE

console.log(`null == undefined? ${null == undefined}\n`); // TRUE perchè entrambi sono valori nulli

console.log(`null == false? ${null == false}`); // FALSE perchè è come se stessimo paragonando null a 0

console.log(`'' == false? ${'' == false}`); // TRUE

console.log(`3 == true? ${3 == true}`); //FALSE

console.log(`0 == -0? ${0 == -0}\n`); // TRUE

console.log(`true + true = ${true + true}`); // 2

console.log(`true !== 1? ${true !== 1}\n`); // TRUE

console.log(`5 + '10' = ${5 + '10'}`); // 510

console.log(`'5' - 1 = ${'5' - 1}\n`); // 4

console.log(`1 < 2 < 3? ${1 < 2 < 3}`); // TRUE

console.log(`3 > 2 > 1? ${3 > 2 > 1}\n`); // FALSE

console.log(`0.2 + 0.1 === 0.3? ${0.2 + 0.1 === 0.3}\n`); // false perché in floating point le somme non sono esatte

console.log('b' + 'a' + (+ 'a') + 'a'); // baNaNa perché sto usando un operatore unario

### STRUTTURE DI CONTROLLO:

* if, else, else if
* switch -case -default
* for
* do while
* Immagine che contiene testo, Carattere, schermata, linea

  Descrizione generata automaticamentewhile
* for (variable in object){}
* for(variable of iterable){} (per array)

### ALTRI METODI DI ITERAZIONE

Metodi per iterare su una collezione

* a.forEach()
* a.map()

### GESTIONE ECCEZIONI:

**Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente**

## ARRAY:

* dichiaro con [] *let v = []; let v=[1,”hi”,3.1, true]; let v=* ***Array.of****(1,”hi”,3);*
* proprietà **.lenght**
* possono avere elementi di tipo a piacere
* metodi di due tipi:
  + **creano copia** e cambiano copia
  + **modificano array**:
    - v.push(“a”); v.push(8); aggiunge alla fine
    - v.unshift(8); aggiunge ad inizio
    - v.pop() elimina ultimo
    - v.pop() elimina primo
* copiare array:
  + let arrayCopia = **Array.from**(v); //copia debole/shallow: *se dentro l’array che sto copiando, ho un indirizzo di un altro array, di questo secondo non viene fatta una copia*
* metodi:
  + .**concat**() : *unisce due array e ne ritorna uno nuovo (Senza modificare)*
  + .**join**(delimiter = ‘,’) *: unisce tutti gli elementi di un array in una stringa con virgole*
  + .**slice**(start\_ind, upto\_ind) : *estrae una sezione di un array, ritornandone uno nuovo*
  + .**splice**(index, cntToRemove, addEl1, addEl2, …): rimuove elementi da un array ed eventualmente li rimpiazza
    - *In place*
  + .**reverse**() : *in place*
  + .**sort**() : ordina *in place ma non riconosce i numeri come valore numerico*
  + .**indexOf**( searchElement *[,fromIndex]* )
  + .**lastIndexOf**( searchElement *[,fromIndex]* )
  + .**includes**( valueToFind [, fromIndex])
* Metodi matematici
  + Math.min() 🡪 vuole come parametri i valori
* destrutturazione assignment:

let [x,y]=[1,2]

* + **[x, y]=[y,x]** 🡪 swap

var foo =[‘1’, ‘2’, ‘3’]  
**var** [one, two, three] =foo 🡪 one=’1’; two=’2’; three=’3’;

let [x**, ..y**] =[1,2,3,4]; 🡪 x=1; y=[2,3,4]

const parts [ ‘shoulder’, ‘knee’]

* const l=[‘heas’**, …parts**, ‘and’] 🡪 [“head”, “shoulder”, “knee”, “and”]

NOTA: volendo esiste la funzione toSorted che mi crea una copia dell’array ordinata

Ex1BetterScore.js

'use strict';

const scores=[20, -5, -1, 100, -3, 30, 50];

const betterScores= [];

let NN=0;

for(let s of scores){  //indifferente se const o let

    if(s>=0)

        betterScores.push(s);

}

Altro metodo usando funzione sort

betterScores.sort((a,b) => a-b);

for(let i=0; i<2; i++){

    betterScores.shift();

}

NN= scores.length-betterScores.length;

for(let i=0; i<2; i++){

    let minScore=Math.min(...betterScores)

    let ind=betterScores.indexOf(minScore);

    betterScores.splice(ind, 1);

}

let avg=0;

for(let s of betterScores){

    avg+=s;

}

avg/=betterScores.length;

avg=Math.round(avg);

for(let i=0; i<NN+2; i++){

    betterScores.push(avg);

}

console.log(betterScores);

## STRINGHE

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamentesequenza immutabile di caratteri unicode da 16 bit.

tutte le operazioni sulle stringhe restituiscono una nuova stringa in quanto le singole stringhe sono immutabili

**+** 🡪 concatenazione

.**length** 🡪 numero di caratteri

**NOTA**: le stringhe possono includere emoticons e cose varie ma i metodi potrebbero avere problemi a gestire quindi se bisogna usare i metodi è meglio evitarle.

### TEMPLATE LITERALS

Stringhe incluse nei backticks possono contenere espressioni delimitate da ${}

* il valore delle espressioni è interpolato nella stringa
  + let name=”bill”;

let free = `Hello ${ name }.`;