Corso JavaScript

0 - Intro

JavaScript nasce in un momento storico molto particolare del web: la metà degli anni Novanta, quando i browser stavano rapidamente evolvendo da semplici visualizzatori di pagine HTML a piattaforme in cui era sensato eseguire codice lato client per rendere le pagine interattive. Il linguaggio fu creato da **Brendan Eich** presso Netscape; la prima implementazione uscì nel 1995 e inizialmente si chiamava *LiveScript*, per poi assumere il nome di **JavaScript** nella versione ufficiale rilasciata a dicembre di quell'anno. Il nome — che ha causato e causa ancora confusione — fu scelto in un contesto commerciale in cui "Java" era un marchio molto in voga, e la scelta servì anche a rinforzare l'appeal del nuovo linguaggio.

A livello di standardizzazione, JavaScript è la più nota implementazione di una famiglia di linguaggi basati sullo standard **ECMAScript**. L'adozione di ECMAScript da parte di Ecma International ha reso possibile un'evoluzione ordinata del linguaggio: dal primo standard formale nel 1997 fino alle edizioni più moderne, con **ECMAScript 2015 (ES6)** che ha portato cambiamenti significativi come le dichiarazioni di classe, i moduli, let/const, arrow function eccetera. Questo processo di standardizzazione è ciò che permette oggi l'interoperabilità tra le implementazioni dei vari browser e dei runtime server-side.

Dal punto di vista del modello di esecuzione, JavaScript è tipicamente eseguito in un singolo thread di esecuzione all'interno del contesto principale del browser: il famoso **event loop** è il meccanismo che permette a codice sincrono e asincrono di convivere senza che il programma vada in blocco completo. L'architettura delle API del browser (le cosiddette Web APIs) fornisce la possibilità di operare in modo asincrono; per attività ricche di calcolo o che non devono bloccare l'interfaccia utente si possono altresì utilizzare i **Web Worker**, che permettono di creare thread di lavoro separati e comunicare tramite messaggi. La memoria è gestita automaticamente: i motori JavaScript implementano **garbage collector** che si occupa di liberare la memoria degli oggetti non più riferenziati, sollevando lo sviluppatore dalla gestione manuale.

Le variabili sono dinamiche e il linguaggio impiega una tipizzazione debole: le stesse entità possono contenere numeri, stringhe o altri tipi nel corso dell'esecuzione, e questo porta grande flessibilità ma anche la necessità di attenzione quando si fanno operazioni che coinvolgono coercizioni implicite di tipo. Dal punto di vista sintattico e semantico, la comparsa di **let** e **const** con ES6 ha risolto molti problemi storici legati a **var** (hoisting e scoping non intuitivo), dando strumenti più robusti per controllare la vita delle variabili.

Dentro il browser JavaScript svolge ruoli essenziali: manipola il DOM per aggiornare la UI in risposta ad eventi, gestisce le chiamate di rete (*fetch*, *XMLHttpRequest*) per ottenere o inviare dati, coordina i comportamenti asincroni tramite promise e async/await e costituisce

la base di single-page application e di molte librerie e framework moderni. Fuori dal browser, l'avvento di runtime come Node.js ha esteso JavaScript al server, permettendo di usare lo stesso linguaggio sia per la parte client che per la parte server e favorendo il paradigma del "JavaScript everywhere".

Storicamente, JavaScript ha contribuito a sostituire numerose soluzioni basate su plugin proprietari che fino a un certo punto rendevano possibile contenuti multimediali e interattivi sul web. Il caso più noto è quello di **Adobe Flash**: con il tempo le API standard di HTML5 e le capacità native dei browser hanno raggiunto la maggior parte delle funzionalità per cui Flash veniva utilizzato (video, audio, animazioni e giochi leggeri), e il progressivo abbandono dei plugin per ragioni di sicurezza e portabilità ha portato Adobe e i principali vendor di browser a cessare il supporto e l'esecuzione dei contenuti Flash alla fine del 2020 (30/12/2020).

Per quanto riguarda le strutture dati, JavaScript offre sia tipi primitivi sia strutture complesse; gli **oggetti** sono il nucleo concettuale: in JavaScript gli oggetti sono insiemi di coppie chiave-valore e la loro implementazione si basa su un **modello prototipale** piuttosto che sull'approccio classico tipico di linguaggi come Java. Il prototipo è un meccanismo che permette a un oggetto di ereditare proprietà e metodi da un altro oggetto. In pratica non si definiscono classi come entità fondamentali, ma si creano oggetti che possono essere usati come "prototipi" per altri. A partire da ES6 è stata introdotta anche una sintassi a classi che però funge sostanzialmente da "syntactic sugar" sul sistema prototipale sottostante; insieme alle classi sono arrivate molte altre primitive utili per la gestione di dati complessi, come Map, Set, WeakMap, WeakSet, e le Typed Arrays per lavorare con dati binari.

JavaScript è attualmente il linguaggio più diffuso sul web, con implementazioni ottimizzate in tutti i browser moderni tramite motori come **V8** (Chrome/Node.js), **SpiderMonkey** (Firefox) e **JavaScriptCore** (Safari) (Wikipedia – JavaScript engines). Il continuo miglioramento delle performance è stato spinto dall'introduzione di tecniche come la compilazione **Just-In-Time (JIT)** e ottimizzazioni specifiche dei motori. L'ecosistema è molto vasto: oltre alle API fornite dal browser e al runtime Node.js, esistono ambienti e strumenti per compilare da e verso JavaScript (**TypeScript**, **WebAssembly** come target per parti prestazionali critiche, transpiler come **Babel**), e la comunità mantiene un ritmo rapido di evoluzione della lingua tramite il comitato **TC39** e il processo di proposta/standardizzazione di ECMAScript (Wikipedia – TC39).

Questo capitolo introduttivo mira a dare una visione d'insieme: la storia che ha portato alla nascita di JavaScript, i principi che spiegano perché si comporta come si comporta (esecuzione single-threaded con event loop, garbage collection, prototipi), il ruolo avuto nello spostare il web da plugin proprietari a standard aperti, e la centralità degli oggetti e delle API moderne. Nei capitoli successivi vedremo in dettaglio ciascuno di questi aspetti, con esempi pratici ed esercizi mirati.

1. I primi passi con JavaScript: console, variabili, operatori

Console e debug

In JavaScript, `console.log()` è il modo più semplice per vedere cosa sta succedendo nel codice e fare debug, lato browser almeno

`Console` fa riferimento allo strumento console del browser
`log` è il tipo di output prodotto, log non genera nessuna formattazione speciale in console (da qua il perché venga usato per debug) ma possiamo usare anche `Console.error` o `Console.warning` in base al contesto

```
console.log("Hello World!");
```

Variabili: `var`, `let`, `const`

- **`var`**: dichiarazione "storica", ha scope di funzione
- **`let`**: dichiarazione moderna, ha scope di blocco (più sicura).
- **`const`**: costante, non può essere riassegnata, ha scope di blocco (ES6)

```
let nome = "Eddy";
const PI = 3.14;
var legacy = "vecchio modo";
```

esempio var

```
 var tester = "hello";

    function newFunction() {
        var hello = "world";
        console.log(hello); //world
    }
    console.log(hello); //errore
    console.log(tester); //hello
```

esempio let

```
  let greeting = "say Hi";
    let times = 4;

    if (times > 3) {
```

```
    console.log(hello) //errore
         let hello = "say Hello instead";
         console.log(hello);// "say Hello instead"
     }
    console.log(hello) // hello is not defined
```

Operatori

- **Aritmetici**: `+`, `-`, `*`, `/`, `%`, `**`
- **Confronto**: `===`, `!==`, `<`, `>`
- **Logici**: `&&`, `||`, `!`

```
console.log(5 + 3);   // 8
console.log(10 % 3);  // 1
console.log(5 === "5"); // false
```

Esercizio: stampa in console il risultato di una somma, un confronto e un operatore logico.

---

2. Gestione dei dati: tipi di dati, array e oggetti

Tipi primitivi

- `string`, `number`, `boolean`, `undefined`, `null`, `symbol`, `bigint`.

```
let testo = "ciao";
let numero = 42;
let attivo = true;
```

Array

Gli array sono liste ordinate di valori.

```
const numeri = [1, 2, 3];
console.log(numeri[0]); // 1
```

Metodi comuni:

- `map`: applica una funzione a ogni elemento
- `filter`: filtra elementi
- `reduce`: riduce un array a un valore

```
const doppi = numeri.map(n => n * 2); // [2,4,6]
const pari = numeri.filter(n => n % 2 === 0); // [2]
```

Oggetti

Gli oggetti sono coppie chiave-valore.

```
const utente = {
  nome: "Anna",
  eta: 25
};
console.log(utente.nome); // "Anna"
```

gli oggetti possono essere dichiarati dinamicamente
per eliminare una chiave, possiamo usare `delete obj.key`
Esercizio: crea un array di oggetti `auto` con `marca` e `anno`, filtra quelle più recenti del 2015.

3. Fondamenti di programmazione in JS

Truthy e falsy

Valori che si comportano come `true` o `false` in condizioni.

```
if ("ciao") console.log("vero"); // stampa
if (0) console.log("mai"); // non stampa
```

Tipizzazione debole

JavaScript converte i tipi automaticamente.

```
console.log(5 + "5"); // "55"
console.log("10" - 2); // 8
```

Esercizio: prova varie operazioni con stringhe e numeri per capire le conversioni implicite.

---

4. Controllo del flusso: condizionali e cicli

Condizionali

```
let x = 10;
if (x > 5) {
  console.log("maggiore di 5");
} else {
  console.log("minore o uguale a 5");
}
```

Switch

```
let colore = "rosso";
switch (colore) {
  case "rosso":
    console.log("Stop!");
    break;
  case "verde":
    console.log("Vai!");
    break;
}
```

Cicli

```
for (let i = 0; i < 3; i++) console.log(i);

for (const lettera of "JS") console.log(lettera);
```

Esercizio: stampa tutti i numeri pari da 1 a 20.

---

5. Funzioni: definizione, parametri, valori di ritorno

Definizione classica

```
function somma(a, b) {
  return a + b;
}
```

Arrow function

```
const moltiplica = (a, b) => a * b;
```

Parametri default e rest

```
function saluta(nome = "ospite") {
  console.log("Ciao " + nome);
}
```

Esercizio: scrivi una funzione che calcoli la media di un array di numeri.
parte 2: falla ricorsiva

---

6. Scope e Closure

Scope

- `let` / `const` hanno scope di blocco
- `var` ha scope di funzione

```
if (true) {
  let a = 10;
  var b = 20;
}
console.log(b); // 20
```

Closure

Una funzione "ricorda" le variabili del contesto in cui è stata creata.

```
function creaContatore() {
  let count = 0;
  return () => ++count;
}

const c = creaContatore();
console.log(c()); // 1
console.log(c()); // 2
```

Esercizio: crea un closure che memorizzi una lista di stringhe e permetta di aggiungerne una alla volta.

Event Loop e Gestione della Memoria in JavaScript

1. Architettura generale

JavaScript è **single-threaded** e gestisce la concorrenza tramite l'**Event Loop**, un meccanismo che coordina esecuzione sincrona e asincrona tra:

- **Call Stack** → esegue il codice (funzioni in corso)
- **Heap** → area di memoria dinamica (oggetti, array, closure)
- **Queues** → contengono i task in attesa di essere eseguiti
- **Event Loop** → orchestration engine che gestisce i flussi

2. Componenti principali

Call Stack

- Struttura LIFO (Last In, First Out)
- Quando lo stack è vuoto, l'Event Loop può processare nuove task

Heap

- Area di memoria gestita automaticamente
- Oggetti e funzioni vengono allocati qui
- Garbage Collector libera memoria di oggetti non raggiungibili (algoritmo *mark-and-sweep*)

Code (Queues)

- **Macro-task queue** → `setTimeout`, `setInterval`, I/O callbacks
- **Micro-task queue** → `Promise.then`, `queueMicrotask`, `MutationObserver`
- L'Event Loop **svuota completamente le microtask** prima di processare una nuova macro-task

Event Loop

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

setTimeout(() => console.log("Timeout"), 0);

Promise.resolve().then(() => console.log("Promise"));

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

Allocazione

let x = 42; // Stack
let obj = { a: 10 }; // Heap

- Primitivi (Number, String, Boolean, ecc.) → Stack
- Oggetti, Array, Funzioni → Heap

Garbage Collection

- Gestita dal motore JS (es. V8)
- Algoritmo **mark-and-sweep**

- Strategie moderne: **generational**, **incremental**, **concurrent**

Potenziali memory leak

- Closure che mantengono riferimenti
- Listener non rimossi (`addEventListener`)
- Reference circolari tra oggetti

Componenti principali

- **Call Stack**:
  Contiene i frame di esecuzione delle funzioni attive. Quando una funzione termina, viene rimossa dallo stack.
- **Heap**:
  Area di memoria dinamica in cui vengono allocati oggetti, array, e funzioni.
  La gestione è automatica (garbage collection).
- **Queues (code di messaggi)**:
  - **Macro-task queue**: contiene task come `setTimeout` , `setInterval` , `I/O callbacks` , ecc.
  - **Micro-task queue**: contiene promesse (`Promise.then` , `queueMicrotask` , `MutationObserver`).
    → Ha **priorità più alta**: viene svuotata completamente prima di processare il prossimo macro-task.
- **Web APIs** (nell'ambiente browser) o **libuv** (in Node.js):
  Sono moduli esterni che gestiscono I/O, networking e timer in background, notificando l'event loop quando le operazioni sono completate.

Fasi di esecuzione semplificate

1. **Esecuzione sincrona:**
   Tutto il codice nel file principale viene eseguito nel call stack.
2. **Registrazione task asincroni:**
   Le chiamate come `setTimeout` o `fetch` passano a Web APIs o libuv.
3. **Event Loop ciclo:**
   - Controlla se il call stack è vuoto.
   - Se sì, estrae un task dalla **queue** e lo mette nello stack.
   - Dopo ogni macro-task, svuota completamente la **micro-task queue**.
   - Continua all'infinito finché il programma è vivo.