

Nuova edizione

JavaScript

La guida definitiva, facile, per tutti

*Un libro completo per imparare velocemente partendo dalle
basi fino agli argomenti avanzati per lo sviluppatore esperto*

contiene
180
esercizi

TONY CHAN

JAVASCRIPT

*# la guida **definitiva**, facile, per tutti*

***Un libro completo per imparare a programmare velocemente partendo
dalle basi fino agli argomenti avanzati per lo sviluppatore esperto.***

Di Tony Chan

Il contenuto in questo libro non può essere riprodotto, duplicato o trasmesso senza il diretto permesso scritto dell'autore o dell'editore.

In nessuna circostanza sarà attribuita alcuna colpa o responsabilità legale all'editore, o autore, per eventuali danni, riparazioni o perdite monetarie dovute alle informazioni contenute in questo libro.

Avviso legale:

Questo libro è protetto da copyright, ed è solo per uso personale. Non è possibile modificare, distribuire, vendere, utilizzare, citare o parafrasarne il contenuto senza il consenso diretto dell'autore o dell'editore.

Avviso di esclusione di responsabilità:

Si prega di notare che il contenuto di questo libro è solo a scopo educativo e d'intrattenimento. È stato compiuto ogni sforzo per presentare informazioni accurate, aggiornate, affidabili e complete. Nessuna garanzia di alcun tipo è dichiarata o implicita. I lettori riconoscono che l'autore non s'impegna a fornire consulenza legale, finanziaria, medica o professionale. Il contenuto di questo libro è in parte derivato da varie fonti. Consultare un professionista autorizzato prima di tentare qualsiasi tecnica descritta.

Leggendo questo testo, il lettore accetta che in nessun caso l'autore sarà ritenuto responsabile per eventuali perdite, dirette o indirette, subite a seguito dell'uso delle informazioni contenute in questo documento, incluse omissioni o inesattezze.

L'autore



Tony Chan è nato a South Gate, in California, Stati Uniti, da genitori immigrati cinesi. A 11 anni scrive il suo primo programma software in Basic, a 17 insegna agli amici programmazione e a 20 si laurea in informatica.

Recentemente ha partecipato come consulente nella stesura di alcuni testi sull'intelligenza artificiale e sul linguaggio macchina. Ha poi deciso di pubblicare i suoi libri sulla programmazione sia negli Stati Uniti, dove vive e lavora, ma anche in Italia, che ama e che visita spesso. Un giorno, un pizzaiolo di Napoli gli ha dedicato la "*Pizza Bill Gates*", di forma rettangolare con il condimento che ricordava il desktop di un pc.

Di Tony si dice che dorme sempre assieme o vicino a un pc, afferma che il rumore delle ventole di raffreddamento gli concilia il sonno...

Indice dei contenuti

Introduzione

1 - Conosciamo Javascript

Strumenti di lavoro

Editor di testo

Intelligenza artificiale (IA).

L'ambiente di lavoro

Node.js

Google Apps Script

Three.js

HTML5 e CSS3

Tag HTML da conoscere

Formattazione del testo, tag validi e obsoleti

Check del codice con le asserzioni

Quiz & esercizi / Riassunto

2 - Le prime basi

Impariamo l'ordine

La sequenza di eventi in una pagina web

Codice globale e codice funzione

Esecuzione del codice Javascript

La “coda” degli eventi

In attesa di un evento

Il verificarsi di un evento

Approfondire il ciclo degli eventi

Differenze di grammatica e punteggiatura

Strict Mode (modalità rigorosa), BOM e DOM

La proprietà innerHTML

Modifica dello stile HTML

Utilizzo di eventi

Creazione di nuovi elementi HTML (nodi).

Rimozione di elementi HTML esistenti

Quiz & esercizi / Riassunto

3 - Stringhe e cicli

Maiuscolo e minuscolo

Regole generali

Proprietà metodi e funzioni

I cicli o iterazioni

Quiz & esercizi / Riassunto

4 - Antipasto con oggetti e dati

[Verifica del tipo di dati](#)

[Conversione del tipo di dato](#)

[Altro sugli oggetti](#)

[L'oggetto Math](#)

[Quiz & esercizi / Riassunto](#)

5 - Gli insiemi

[I vettori \(o array\).](#)

[I metodi dei vettori \(o array\).](#)

[Vettori e loop](#)

[Utilizzare il metodo map in un vettore](#)

[Controllare gli elementi di un vettore](#)

[Utilizzare i metodi di ricerca di un vettore](#)

[Utilizzare il metodo sort per sistemare gli elementi in un vettore](#)

[Il metodo reduce](#)

[Le mappe in Javascript](#)

[Iterazione su Map](#)

[Il costruttore set e Operazioni tra set](#)

[I nuovissimi metodi per gli array \(ECMA 2023\).](#)

[Quiz & esercizi / Riassunto](#)

6 - Le funzioni

[Oggetti Javascript](#)

[Definizione dell'oggetto](#)

[Proprietà dell'oggetto](#)

[La parola chiave this](#)

[Accesso ai metodi degli oggetti](#)

[Oggetti e funzioni](#)

[La versatilità delle funzioni](#)

[I callback](#)

[Operatori e conversione](#)

[Raccolta e disposizione di dati](#)

[Sfruttare gli oggetti funzione](#)

[Differenze tra le varie tipologie di funzioni](#)

[La funzione dichiarativa](#)

[La funzione espressione](#)

[Le funzioni immediate](#)

[Le funzioni freccia](#)

[Distinguiamo tra parametri ed argomenti](#)

[Utilizzare l'argomento della funzione](#)

[Usare la sintassi spread con gli argomenti](#)

[Parametri rest, default e destructured](#)

[Evocare le funzioni avvalendosi di this](#)

[Evocare una funzione direttamente](#)

[Evocare una funzione con il costruttore](#)

[Evocare una funzione come metodo di un oggetto](#)

[Evocare una funzione con i metodi apply e call](#)

[Funzioni freccia, callback e this](#)

[Il metodo bind](#)

[Quiz & esercizi / Riassunto](#)

[# 7 - Le Chiusure](#)

[Concetto di scope o contesto di esecuzione](#)

[Utilizzo delle chiusure](#)

[Chiusure e callback](#)

[Approfondiamo le variabili](#)

[Variabili metasintattiche](#)

[Gli ambiti in Javascript](#)

[La variabile var \(obsoleta\)](#)

[Le variabili let e const](#)

[Comprendere le regole degli ambiti](#)

[Eseguire una funzione prima della sua effettiva creazione](#)

[Sovraccarico degli identificatori di funzione](#)

[Sollevamento in Javascript](#)

[Variabili private e chiusure](#)

[Da sapere sulle variabili private](#)

[Chiusure e callback per animare oggetti?](#)

[Quiz & esercizi / Riassunto](#)

[# 8 - Codice Asincrono](#)

[La funzione generatore](#)

[Sfruttare i generatori](#)

[Problemi con i callback](#)

[Abbasso i callback W le promesse](#)

[Promesse concatenate](#)

[In attesa delle promesse](#)

[I metodi delle promesse](#)

[Generatori e promesse uniti per la gloria!](#)

[Async e await](#)

[Quiz & esercizi / Riassunto](#)

[# 9 - Oggetti software](#)

[I prototipi](#)

[Prototipi, oggetti e costruttori](#)

[Differenze di concetto](#)

Prototipi e proprietà hanno le stesse peculiarità

L'ereditarietà delle proprietà

Le proprietà dell'oggetto software

Le applicazioni dell'operatore instanceof

Le classi

Class extend e super per l'ereditarietà

Get e set

Utilizzare get e set

Utilizzo dei proxy e i suoi vantaggi

Utilizzo di proxy per semplificarci la vita

Belle le trappole ma...

Quiz & esercizi / Riassunto

10-Modularizzazione del codice

Modularizzazione del codice in Javascript prima del 2015

I moduli con ECMAScript 6 (ES6).

Export con default

Rinominare importazioni e esportazioni

Utilizzare "import.meta"

Caricamento dei moduli

Analisi di performance

[Quiz & esercizi / Riassunto](#)

[# 11-Json JQuery Ajax e Three.js](#)

[JSON e XML](#)

[JQUERY](#)

[Selettori e pseudo selettori](#)

[Il metodo .find\(\)](#)

[Attraversamento e manipolazione](#)

[Catene](#)

[Gestire il DOM](#)

[Modificare elementi del DOM](#)

[Iterazioni](#)

[Eventi jQuery](#)

[Registrazione ascoltatori di eventi](#)

[Ascoltatori di eventi delegati](#)

[Eventi tastiera](#)

[Eventi sullo schermo](#)

[AJAX](#)

[I metodi AJAX](#)

[I parametri AJAX](#)

[Three.js](#)

[Quiz & esercizi / Riassunto](#)

[# 12 – Le espressioni regolari](#)

[Esempi di espressioni regolari](#)

[I fondamenti delle espressioni regolari](#)

[La sintassi avanzata delle RegExp](#)

[Metacaratteri](#)

[Le espressioni regolari al lavoro!](#)

[RegExp: la cattura](#)

[I metodi matchAll e replaceAll](#)

[Evitare catture doppie con le sottoespressioni passive](#)

[Sostituzione utilizzando le funzioni](#)

[Risoluzione dei problemi con le RegExp](#)

[Test espressioni regolari](#)

[Quiz & esercizi / Riassunto](#)

[**Link download - Bibliografia**](#)

*"JavaScript è l'unico linguaggio di programmazione che ho imparato e iniziato a usare quotidianamente, il quale mi ha reso una persona migliore in ogni aspetto della mia vita professionale." - **Douglas Crockford***

Mi piace sempre iniziare con una precisazione assolutamente d'obbligo: Javascript non è una versione di Java destinata al web. Spesso vengono confusi, ma sono molto diversi: Java è un linguaggio di programmazione orientato agli oggetti e deve essere compilato per essere eseguito, con il vantaggio di funzionare su qualunque computer, pc o mac. Mentre Javascript è un linguaggio di *scripting*, cioè essenzialmente una lista con alcune mansioni da eseguire che poi saranno interpretate dal browser, o dall'ambiente di esecuzione. Tra l'altro hanno anche sintassi e plugin differenti.

Nonostante la premessa apparentemente riduttiva Javascript ha avuto e ha tuttora un grande successo perché: ha una sintassi simile al C ma è relativamente semplice e allo stesso tempo potente, è un linguaggio orientato agli oggetti e, attraverso le numerose librerie (come ad esempio JQuery), permette di fare cose eccellenti e forse impossibili con altri mezzi, inoltre è molto versatile e soprattutto multiplatforma.

Ma come nasce questa meraviglia? Nel 1995 *Brendan Eich*, un geniale programmatore nato a Pittsburgh nel 1961, fu assunto da *Netscape*, azienda che stava cavalcando l'iniziale esplosione del web, con lo scopo di inserire e implementare un linguaggio di nome *Scheme* all'interno del suo browser. Ebbene, in solo dieci giorni creò *Mocha*, un codice orientato agli oggetti destinato a rivoluzionare il web e non solo. Il nome quasi subito cambierà in qualcosa a noi più familiare: **JAVASCRIPT**.

Oggi, dopo quasi trenta anni di sviluppo, tantissimi cambiamenti e nonostante sia ancora incentrato sul browser, questo linguaggio di programmazione è uno dei migliori strumenti a nostra disposizione per creare applicazioni. Perché, come già scritto, è potente, ma tutto sommato abbastanza semplice, è multiplatforma, ha una floridissima community ed è assolutamente al passo con i tempi e proiettato verso un futuro ancora più roseo. Tutte cose che consolidano la leadership di JavaScript anche per gli anni a venire.

Queste affermazioni potrebbero sembrare eccessive o di parte, tuttavia basterà una rapida occhiata sulla rete per capire che non si sta esagerando affatto.

Anzi, i più attenti noteranno anche che JavaScript è presente in moltissimi dei siti visitati, partendo dai più semplici fino a quelli istituzionali.

I numerosi strumenti a cui abbiamo accesso oggi, forniti dagli sviluppatori ufficiali e dalla comunità open source, hanno raggiunto un livello eccellente, impensabile solo pochi anni fa. Esiste una gran quantità di frame work, la possibilità di eseguire test di tutti i tipi, generare report di copertura del codice, verifiche delle prestazioni su dispositivi mobili reali in tutto il mondo, caricare browser virtuali su qualsiasi piattaforma da cui eseguire prove e immense community, come ad esempio *StackOverflow*, *Italia JS* su *Discord*, le discussioni su *Reddit* e anche diversi gruppi su *Telegram* e *Facebook* dove potremmo trovare ispirazione o risposte praticamente su qualsiasi problematica legata al codice.

Forse adesso riusciamo a capire come e perché JavaScript è diventato uno dei fondamenti della rivoluzione tecnologica, che, tra l'altro, è ancora in atto, proprio adesso. Se pensiamo ai primi browser come Netscape e Internet Explorer e alle loro incompatibilità e alle tante problematiche, ci rendiamo davvero conto del balzo che è stato compiuto. Chissà, forse solo chi le ha vissute può capirlo. Comunque, oggi Javascript è diventato uno dei linguaggi di programmazione più utilizzati e significativi. Non è focalizzato solo sui browser, viene adoperato sul server grazie a *Node.js*, *Google Apps Script* e tanti altri, su dispositivi desktop e cellulari attraverso piattaforme come *Apache Cordova*, su dispositivi IoT (*Internet of Things*, Internet delle cose) con *Iot.js*, *Cylon.js*, per citarne solo due, con le API WebVR e WebAR per la realtà virtuale del *Metaverso*, e mi fermo qui per non tediarvi.

Sebbene questo libro sia incentrato principalmente su JavaScript eseguito nel browser, i fondamenti del linguaggio che studieremo sono applicabili in tutti i campi.

Come praticamente tutti i percorsi di studio che si rispettino anche qui abbiamo realizzato un itinerario pensato in modo da progredire gradualmente introducendo i concetti base, fino ad arrivare alle tematiche più complesse, anche per i programmatori esperti.

La parte iniziale anticiperà l'argomento con i fondamentali: difatti nei primi quattro capitoli guarderemo le caratteristiche e gli elementi principali necessari ad iniziare lo sviluppo, daremo una veloce occhiata al codice HTML, al DOM e a come funziona il ciclo degli eventi, proseguiremo con un piccolo ripasso, che ci servirà più avanti, sulle stringhe e sui loop, infine, un'utile anteprima inerente agli oggetti software.

Nella parte centrale ci concentreremo su un argomento cardine di JavaScript: le funzioni. Nel capitolo più corposo del libro, il sesto, studieremo perché sono così importanti in JavaScript, i diversi tipi di funzioni e i dettagli essenziali per la loro invocazione. Dopo, nel settimo capitolo impareremo due concetti strettamente correlati: scopi e chiusure. Argomento chiave nella programmazione funzionale, le chiusure ci consentono di esercitare un controllo dettagliato sull'ambito degli oggetti che dichiariamo e creiamo nei nostri programmi. Concluderemo il nostro studio delle funzioni nel capitolo otto, dando uno sguardo ai generatori e alle promesse che ci aiuteranno anche a gestire i valori asincroni. Vedremo anche come combinarli assieme. Nel capitolo nove, la parte finale del libro, arriveremo ad occuparci finalmente del cuore di tutti i linguaggi di programmazione *OOP*: gli oggetti software, come ci ricorda l'acronimo stesso: *Object Oriented Programming*, ossia orientato agli oggetti. *Oggetto software* è una definizione metaforica la quale serve anche a farci capire la differenza di principio del modello di programmazione incentrato sui dati piuttosto che sulle procedure, il quale di fatto, già dagli anni '80 ha semplificato la progettazione delle applicazioni. Quindi definiremo questi oggetti e li esploreremo a fondo, studieremo come proteggerne l'accesso, gestirne le proprietà e le raccolte. Nel capitolo cinque ci soffermeremo sui vettori (gli insiemi o *array*) e sulle mappe. Dulcis in fundo, nel capitolo dieci e nell'ultimo, il numero dodici, ci dedicheremo anche alle espressioni regolari e alla modularizzazione del codice. Nel capitolo undici effettueremo una digressione sugli argomenti relativi a JSON, alla libreria più utilizzata in Javascript, cioè JQUERY, al linguaggio AJAX e infine alla libreria dedicata alla grafica 3D, Three.JS.

Prima di iniziare mi preme fare un piccolo appunto sulle numerose righe di codice presenti nel libro. Ebbene, esse sono state testate molte volte su diverse piattaforme, quindi, se dovesse presentarsi un errore probabilmente state utilizzando un browser o un programma con limitazioni dovute alla compatibilità. Solitamente la console di Google Chrome funziona relativamente bene, anche solo per un test veloce, pur rimanendo abbastanza limitata, ad esempio, le asserzioni non sono ancora supportate. Quindi, nei prossimi paragrafi suggeriremo alcuni software, sempre gratuiti, più potenti ed aggiornati.

Infine, dato che hai acquistato la versione cartacea avrai diritto a scaricare l'ebook gratuitamente anche per motivi di praticità, infatti potrai facilmente

utilizzare e gestire tutti i listati presenti sul libro in maniera molto più semplice avendo la possibilità di selezionarli e copiarli all'interno del software utilizzato per studiare.

È presente anche un link per scaricare le soluzioni a tutti gli esercizi presenti. I QR code per i download si trovano alla fine del libro.

1 - Conosciamo Meglio Javascript

Le caratteristiche e gli elementi principali

Iniziare lo sviluppo in JavaScript

Sprazzi di HTML e CSS

Come abbiamo già scritto in precedenza, JavaScript è letteralmente dappertutto, anche grazie a motori più potenti e all'introduzione di framework come *Node*, che lo hanno portato ad un livello decisamente superiore rispetto alle prime versioni viste con Netscape. E, come l'HTML, il linguaggio stesso sta ricevendo aggiornamenti attesi da tempo intesi a rendere JavaScript ancora più adatto per lo sviluppo di applicazioni moderne. Si pensi, ad esempio, che i primi anni il browser interpretava il codice, cioè sostanzialmente lo traduceva riga per riga in linguaggio macchina, con evidenti problemi di lentezza con applicazioni più grandi e complesse. I browser attuali hanno dei motori i quali in parte compilano e in parte interpretano il codice chiamati JIT (Just In Time, appena in tempo) effettuando una *traduzione dinamica* praticamente identica alla classica compilazione. Oltre a questo sussistono moltissimi esempi della sua evoluzione e del suo sfruttamento. In sostanza Javascript, dopo tanti anni oramai, è ampiamente conosciuto e utilizzato. Tuttavia, molti programmatori, nei listati JavaScript ricorrono a una sintassi analoga al C, e ad altri simili come C# e Java, e questo ovviamente comporta dei grossi problemi di bug. Rispetto ad altri linguaggi tradizionali, JavaScript è molto più orientato alla funzionalità. Alcuni concetti differiscono fondamentalmente, ad esempio: in JavaScript, le funzioni coesistono e possono essere trattate come qualsiasi altro oggetto. Possono essere create tramite valori letterali, referenziate da variabili, passate come argomenti della funzione stessa, persino ottenute come valore restituito dalla funzione, e inoltre, possono essere “chiuse”. Vedremo cosa significa e impareremo il giusto approccio nei confronti di questo linguaggio di programmazione nel corso del nostro studio.

Agli albori JavaScript non aveva variabili con *scope* (ambito) locale come in C, o in C#. Si doveva fare affidamento solo su quelle globali e lo stesso valeva per le funzioni. Studieremo come adesso l'ambito di validità delle variabili sia un concetto utile e importante.

JavaScript utilizza i *prototipi*, a differenza di altri linguaggi di programmazione tradizionali come C#, Java o Ruby, i quali utilizzano l'orientamento agli oggetti basato sulla classe, e spesso, quando gli sviluppatori provenienti da quel background arrivano a JavaScript, lo sfruttano scrivendo il codice basato su classi serevendosi della sintassi di JavaScript. Poi, rimangono anche stupiti quando i risultati differiscono da quelli che si aspettavano. Questo è uno dei motivi per cui nel nostro percorso di studio approfondiremo i prototipi, l'orientamento agli oggetti e come vengono implementati.

In JavaScript vi è una stretta correlazione tra oggetti e prototipi, funzioni e chiusure. Comprendere i solidi nessi tra questi concetti può migliorare notevolmente la nostra capacità di programmazione, offrendoci una solida base per qualsiasi tipo di sviluppo di applicazioni, indipendentemente dal fatto che il nostro codice JavaScript sia eseguito in una pagina web, in un'app desktop, in un'app mobile o sul server.

Oltre a questi concetti fondamentali, altre funzionalità JavaScript possono aiutarci a scrivere un codice schematico, pulito ed efficiente. Alcune di queste sono funzionalità che gli sviluppatori esperti riconosceranno da altri linguaggi, come Java e C ++, ad esempio:

I generatori, sono funzioni che possono creare più valori e possono sospendere la loro esecuzione tra le richieste per offrirci un migliore controllo sul codice asincrono assieme alle *promesse*.

Le Mappe, usate per creare raccolte di dizionari e set, con cui gestiremo collezioni di oggetti unici.

I moduli, suddividono il codice in parti più piccole e comprensibili.

Le RegEx, ci consentono di condensare ulteriormente i listati.

Grazie all'enorme successo di JavaScript come linguaggio client-side, Microsoft ne sviluppò una sua versione diversa e compatibile del linguaggio chiamandolo *JScript* e includendolo nella versione 3 di Internet Explorer, creando parecchi problemi di compatibilità.

Quindi, nel 1996, su spinta di Netscape, *Ecma International*, un'associazione che dal 1961 si occupa di standardizzare il settore informatico e anche i linguaggi, tra cui C# e C++, pubblica il documento *ECMA-262*,

volto a mediare tra Netscape e Microsoft. Il risultato fu *ECMAScript* (o ES), il quale definisce la specifica tecnica di un linguaggio di scripting, standardizzato e mantenuto da *ECMA International*. I più famosi sono JScript e ActionScript, ma soprattutto *JavaScript*. Tuttavia ciò non impedì a Netscape e Microsoft di fornire ulteriori funzioni al di fuori degli standard. Ad oggi sono tredici le edizioni dell'ECMA-262. Dal 2015 la pubblicazione avviene annualmente:

Giugno 1997, prima edizione.

Giugno 1998, seconda edizione.

Dicembre 1999, terza edizione (aggiunta *espressioni regolari*).

Quarta Edizione, (abbandonata).

Dicembre 2009, quinta edizione (aggiunti *getters* e *setters*).

Giugno 2015, sesta edizione o ECMAScript 2015 (aggiunte nuove dichiarazioni delle *classi* (`class Foo { ... }`) e i moduli con *import* e *export*. le funzioni a freccia (`() => {...}`), i generatori, molto simili a quelli Python).

Giugno 2016, settima edizione, ECMAScript 2016 (aggiunto il *block-scoping* per variabili e funzioni, e la nuova gestione del codice asincrono).

Giugno 2017, ottava edizione, ECMAScript 2017 (aggiunto *concurrency* ed *atomics*, *zero copy binary transfer*, e molto altro).

Giugno 2018, nona edizione, ECMAScript 2018 (aggiunte le iterazioni asincrone (*for await*(i of var)), la *spread syntax* per i parametri delle funzioni (`function pippo(foo, ...bar)`).

Giugno 2019, decima edizione, ECMAScript 2019 (aggiunti metodi *Array.flatMap()* e *Object.fromEntries()*).

Giugno 2020, undicesima edizione, ECMAScript 2020 (aggiunti metodo *matchAll()*, la keyword *GlobalThis*).

Giugno 2021, dodicesima edizione, ECMAScript 2021 (aggiunto metodi *replaceAll()*, *Promise.any* e l'oggetto *WeakRef*).

Giugno 2022, tredicesima edizione, ECMAScript 2022 (aggiunti i metodi *.at()*, *Object.hasOwn()*, nuovi membri per le classi e funzionalità sui moduli *await*, metodo *#* per rendere velocemente le classi private).

Giugno 2023, quattordicesima edizione, ECMAScript 2023 (nuovo set di istruzioni per lavorare con gli array, metodi *toSpliced()* e *toSorted()*).

Giugno 2024, quindicesima edizione, ECMAScript 2024 (*Atomics.waitAsync*, *RegExp* v flag con notazione impostata + proprietà delle stringhe, *ArrayBuffer* ridimensionabili e espandibili, da confermare)

Fino a qualche anno fa questi continui aggiornamenti da una parte erano un'ottima notizia, ma dall'altra fomentavano tutta una serie di problematiche e preoccupazioni per le eventuali incompatibilità tra i vari browser, che tuttavia persiste, ma in maniera molto inferiore rispetto solo a quattro o cinque anni fa, soprattutto perché oramai il livello tecnico dei browser è altissimo, dato che la struttura del linguaggio Javascript è del tutto assodata. Le prossime edizioni apporteranno solo aggiustamenti, e, in caso contrario gli aggiornamenti saranno molto veloci. Inoltre esistono strumenti che ci consentono di controllare le problematiche eventuali di compatibilità, una su tutte l'utilissimo: <https://developer.mozilla.org/>.

In questo libro, ci concentriamo in particolare sull'esecuzione del codice JavaScript nel browser. Iniziamo vedendo quali saranno i mezzi con cui lavoreremo, studieremo e che ci faranno compagnia in questo percorso.

Strumenti (Tools) di lavoro

Sebbene, in linea teorica, per scrivere codice JavaScript e HTML sia sufficiente disporre di un editor di testi (come il *blocco note*) e di un browser (la console su Google Chrome: tasto ctrl+shift+j), quando si comincia a scrivere codice in modo un po' più "serio", gli strumenti giusti possono fare davvero la differenza.

Senza la pretesa di essere esaustivi, nelle pagine che seguono proporremo una panoramica degli strumenti (gratuiti) più diffusi:

Editor di testo

Una prima alternativa al Blocco Note è costituita dagli editor di testo "più evoluti". Ne ricordiamo tre: Notepad ++, Atom e Visual Studio.

Il primo di questi, **Notepad ++** è un editor gratuito che può essere scaricato dal sito <https://notepad-plus-plus.org/>.

La sua interfaccia è semplice e di facile utilizzo. Dispone di una serie di caratteristiche davvero utili: evidenziazione della sintassi, raggruppamento di porzioni omogenee di codice (Syntax Folding) in modo da poter nascondere o visualizzare porzioni di un documento lungo, evidenziazione della sintassi e Syntax Folding personalizzato dall'utente, evidenziazione delle parentesi, ricerca/sostituisci mediante espressioni regolari (Perl Compatible Regular Expression), autocompletamento della sintassi, segnalibri, visualizzazione a schede, visualizzazione di documenti affiancati per il confronto.

Atom è un editor gratuito scaricabile dal sito <https://atom.io/> disponibile per più piattaforme (OS X, Windows e Linux). Può essere completato con diversi pacchetti open source e dispone di supporto al sistema di controllo di versione Git.

Fra i punti di forza di Atom ci sono: autocompletamento, evidenziazione della sintassi, funzionalità di ricerca e sostituzione fra più file, possibilità di aprirne diversi in pannelli affiancati per poterli confrontare.

Visual Studio Code è l'editor che abbiamo usato per scrivere gli esempi di questo libro. È un editor sviluppato da Microsoft per più piattaforme (OS X, Windows e Linux). Si tratta di uno strumento gratuito scaricabile dalla pagina <https://code.visualstudio.com/>.

Dispone già di Git ed è integrabile con ulteriori pacchetti. Fra i suoi punti di forza ci sono: autocompletamento, evidenziazione della sintassi, funzionalità di ricerca e sostituzione, possibilità di impostare breakpoint, lavorare direttamente con file e cartelle senza la necessità di creare progetti.

Ci servirà anche un *linter*, il quale è un programma che in genere si integra con un editor di codice e permette di evidenziare gli errori di sintassi o in generale di scrittura del codice. Attualmente, uno dei *linter* per JavaScript più diffusi è ESLint (<https://eslint.org/>).

AppStore: oltre ai software elencati, in tutti i dispositivi aggiornati con Windows, MacOS, iOS e Android, negli *appstore* dedicati esistono numerosi editor gratuiti anche in prova, più o meno validi, comodi e veloci, da utilizzare per esercitarsi e studiare. Vale la pena darci un'occhiata per trovare quello che fa per noi.

Intelligenza Artificiale (IA)

Il 2022 è stato l'anno del consacramento di questa tecnologia ad uso e consumo del grande pubblico. Ve ne sono una gran quantità disponibili gratuitamente (almeno finora), alcune specializzate solo su determinati ambiti, come ad esempio la creazione di immagini o nella diagnosi medica. La maggior parte di noi ha sicuramente sperimentato il fantastico giocattolo di OpenAi, cioè Chat GPT (*Generative Pretrained Transformer*, in pratica un sistema di apprendimento detto "pre-addestramento generativo"), il quale si serve di una rete neurale e di complessi algoritmi di *deep learnig* (apprendimento profondo, essenzialmente immense quantità di dati a disposizione per imparare a svolgere vari compiti) per comprendere e generare del testo di risposta ad un input di un utente.

A prescindere dall'interesse che possa suscitare in noi l'argomento, questa tecnologia è anche utilissima nel nostro ambito. Infatti proprio con Chat GPT è possibile avere un dialogo costruttivo su argomenti legati alla programmazione e anche ottenere del codice come esempio. Praticamente, in alcuni casi potrebbe sostituirsi ad un tutor, o sbloccare una situazione in cui un listato ci crea dei problemi. Vi consiglio di darci un'occhiata.

L'ambiente di lavoro

Le applicazioni JavaScript possono essere eseguite in tanti ambiti o ambienti. Quello che utilizzeremo maggiormente è il browser. Come tutti sappiamo i principali sono: Chrome, Mozilla Firefox e Edge della Microsoft, potremmo aggiungere Opera e Safari oltre alle stesse versioni, non meno importanti, anzi, entro breve di più, per IOS e Android. Bene, giusto per ribadire il concetto, su tutte queste piattaforme, nonostante sia stato standardizzato a Giugno 2021, il metodo *replaceAll()* è perfettamente compatibile e funzionante già dopo pochi mesi.

Oltre a sapere che il nostro ambiente preferito oramai è costantemente supportato e all'avanguardia, dovremo capire, anche solo genericamente, come funziona e come lavora. Di fatto, Il browser è un interprete, è una finestra aperta rivolta verso un click o una nostra ricerca. Tra le altre cose, e per quello che interessa sapere a noi, è composto da infrastrutture, concetti e API che andremo ad approfondire:

Gli *Eventi* fanno parte di quasi tutte le applicazioni JavaScript, il che significa che la maggior parte del codice viene eseguita nel contesto di una risposta a un particolare evento. Ad esempio: eventi relativi alla rete, eventi generati dagli utenti come: clic, movimenti del mouse, pressioni sulla tastiera e così via.

Il *Document Object Model (DOM)* è una rappresentazione strutturata ad oggetti della pagina web. Per sviluppare ottime applicazioni, non solo è necessario avere una profonda conoscenza dei meccanismi di base di JavaScript, ma anche studiare come il DOM è costruito (capitolo 2) e come scrivere codice efficace che manipoli il file DOM. Questo metterà la creazione di avanzate, altamente dinamici è a portata di mano.

Le *API (Application Programming Interface)* del browser ci aiutano ad interagire con il mondo. Letteralmente *Interfaccia di Programmazione*, nel browser mettono in comunicazione due programmi diversi, come, ad

esempio, quando cerchiamo un volo su sito, questo si mette in contatto con le API delle compagnie aeree per riportare i risultati.

Perfezionare le nostre capacità di programmazione JavaScript e raggiungere una profonda comprensione delle API offerte dal browser ci aiuterà veramente tanto, ma, come per tutti i campi relativi al digitale, bisognerà sempre tenersi aggiornati e studiare, ad esempio, qualche anno fa JavaScript sfuggì ai confini del browser venendo utilizzato sul server con Node.js.

I progressi apportano miglioramenti, ma hanno un rovescio della medaglia, nel mondo digitale relativo alla programmazione si parla di bug e di supporto a tutte le piattaforme (browser in questo caso). Sebbene la qualità dei browser sia migliorata notevolmente negli ultimi tempi, hanno ancora diversi bug, API mancanti e problematiche che dovremo affrontare. Sviluppare una strategia completa per superare queste difficoltà e acquisire familiarità con i vari intoppi, le differenze tra le varie piattaforme e le altre problematiche, non è meno importante della competenza in JavaScript stesso.

Scriveremo applicazioni browser e dovremo controllarne la compatibilità, soprattutto nei confronti del mondo mobile, perché forse non conosci questi dati (fonte: *Gartner*): nel 2022 in tutto il mondo ci sono circa 6,5 miliardi di dispositivi tra notebook e desktop, tablet e smartphone, di cui circa 4,5 miliardi di smartphone, 950 milioni di notebook (in crescita rispetto al 2021 di circa 100 milioni di unità), e il resto suddiviso tra tablet e desktop. Oramai gli smartphone da qualche tempo hanno sostituito il computer, e questi dati dovremo tenerli bene a mente mentre progettiamo e sviluppiamo il nostro listato. In sintesi estrema, multiplatforma arricchisce il suo significato focalizzandosi sui dispositivi mobili, e qui, oltre ai due colossi Android e IOS, stanno nascendo e nasceranno nuovi sistemi operativi, come *Fuchsia* di Samsung e Google e *HarmonyOS 2.0* di Huawei per esempio, e, come già detto, sebbene la situazione sia notevolmente migliorata, i browser purtroppo non sono e non saranno privi di bug e non supportano e non supporteranno gli standard web senza alcun intoppo.

Ne consegue che le maggiori sfide nello scrivere codice JavaScript che può essere eseguito in vari browser sono: correzioni di bug, bug del browser, funzionalità mancanti e codice esterno, ma soprattutto, come regola principale, lo sviluppo multiplatforma implica il controllo di alcuni fattori fondamentali:

- Dimensione del codice*, mantenere piccole dimensioni del file.

-*Prestazioni*, non appesantire il lavoro, mantenendo un livello di calcolo medio basso.

-*API*, assicuriamoci che le API funzionino in modo uniforme su tutti i browser.

Lo sviluppo di codice efficace e cross-browser può dipendere in modo significativo dall'abilità e dall'esperienza di uno sviluppatore. Tuttavia esistono diversi strumenti utili ad aiutarci, infatti, inizialmente eseguire il debug di JavaScript voleva dire enormi perdite di tempo. Oggi, tutti i browser oltre ad eseguire il nostro codice ci aiutano anche a correggerlo con: *Firebug*, popolarissima estensione per sviluppatori del browser Firefox (<http://getfirebug.com/>), *Chrome DevTools* utilizzato in Chrome e Opera, *strumenti per sviluppatori di Firefox* incluso in Firefox, *strumenti per sviluppatori F12* presente su Internet Explorer e Microsoft Edge, *WebKit Inspector* per quanto riguarda Safari e il mondo Apple.

Tutti questi strumenti offrono funzionalità abbastanza uniformate tra di loro come: il debug del codice Javascript, controllare il DOM, modificare gli stili CSS, con *Chrome Dev Tools* eseguiremo anche il debug di altri tipi di applicazioni, come le app *Node.js*, e l'elenco potrebbe continuare a lungo. Quindi, riassumendo, non sarà necessario orientarsi su di una scelta specifica, forse dovremo solo conoscere le molteplici possibilità che abbiamo a disposizione ed imparare ad utilizzarle un po' tutte per comprenderne al meglio i meccanismi. Per questo motivo apriremo una piccolissima parentesi su *Node.js*, *Google Apps Script*, e *Three.js*, che servirà a farci capire le potenzialità e l'eterogeneità di questo ecosistema. dopodiché come antipasto avremo un po' di HTML, giusto per prepararci finalmente la bocca al piatto forte: le basi di Javascript.

Node.js

Node.js è un sistema di esecuzione da remoto multiplatforma orientato agli eventi per l'uso di codice JavaScript, costruito sul motore di Google Chrome. Molti dei suoi moduli base sono scritti in JavaScript, e, volendo, gli sviluppatori possono aggiungerne altri.

In origine JavaScript veniva sfruttato soprattutto dal lato client. Di solito gli script erano incorporati all'interno dell'HTML di una pagina web, venivano interpretati da un motore di esecuzione direttamente all'interno di un Browser. Node.js consente invece di utilizzare JavaScript anche per scrivere codice da eseguire lato server, ad esempio per la produzione del

contenuto delle pagine web dinamiche prima che la pagina venga visualizzata dal Browser dell'utente.

Node.js ha un'architettura orientata agli eventi che rende possibile l'Input/Output asincrono, ed è ottimo, ad esempio, per applicazioni web in real-time, come programmi di comunicazione o browser game.

Infine, su Node.js abbiamo NPM (*Node Package Manager*), ossia il più grande sistema di librerie libere, modificabili e gratuite esistente. Ci permetterà di usufruire di un'enorme quantità di materiale già pronto, testato da altri, disponibile per tutti, come ad esempio un breve ed eccellente listato su come creare una blockchain con poche righe di codice (il link è nell'ultima pagina).

Google Apps Script

Un'altra parentesi la vorrei dedicare a questa piattaforma. Tutti noi conosciamo Google e i suoi servizi veramente utili. Bene, con *Google Apps Script* potrai iniziare a scrivere codice senza l'ausilio di alcun software, in maniera del tutto gratuita, con la possibilità di interagire con i prodotti della *G suite*, ossia Documenti, Presentazioni, Fogli e Moduli, in pratica la versione online del famoso Office di Microsoft. Si basa una versione più vecchia di Javascript, la 1.6, e parzialmente sino alla 1.8. Ha un suo debug basato su cloud. In pratica potrebbe avere un'utilità nell'eseguire qualche attività di amministrazione del sistema e la sua vasta community è sempre utilissima per qualunque dubbio o intoppo del nostro programma.

Three.js

Cosa c'entrerà mai questa libreria? vi starete chiedendo. Forse poco, forse moltissimo. Di fatto l'annuncio del cambio di *Facebook* in *Meta* avvenuto alla fine del 2021 ci ha forzatamente spediti verso il futuro del Web 3.0. Senza entrare troppo nei dettagli e iniziare speculazioni e previsioni di sorta, bisogna ammettere che molti cambiamenti già sono in atto e da un po' alcuni siti si stanno proponendo in 3D puro (qui un elenco di alcuni tra quelli più significativi: <https://www.awwwards.com/websites/three-js/>). Fatta questa doverosa premessa è bene sapere che questa libreria esiste già dal lontano 2010, ed è utilizzata per creare e visualizzare grafica tridimensionale in movimento in un browser per mezzo dell'API WebGL.

Nel libro dedicheremo un piccolo paragrafo che ci farà capire le potenzialità di *Three.js* ma soprattutto di Javascript.

HTML5 e CSS3

Da questo punto in poi entriamo un po' di più nel concreto. Sappiamo che JavaScript è principalmente utilizzato per la creazione di siti web interattivi, quindi è evidente che abbiamo bisogno di interfacciarci con i linguaggi di sviluppo web. Ecco quindi HTML, acronimo di *HyperText Markup Language*, che è un linguaggio di markup utilizzato per strutturare il contenuto delle pagine web utilizzando tag e markup. JavaScript è comunemente utilizzato in combinazione con HTML per aggiungere interattività.

Molti si chiederanno la differenza tra HTML e HTML5. Quest'ultimo è una versione più recente e avanzata di HTML che è diventata lo standard per lo sviluppo web moderno. Tuttavia, il termine 'HTML' può ancora essere utilizzato per riferirsi a versioni precedenti o per scopi generali quando non è necessario specificare una versione.

Detto questo, ci sono alcune cose che dovremo conoscere, come la posizione del nostro codice JavaScript nel documento HTML. Può posizionarsi sia internamente che esternamente in un file separato che verrà indicato nel nostro listato.

Iniziamo osservando la struttura standard HTML:

```
<!DOCTYPE html>
<html>
<head></head>
<body></body>
</html>
```

Per includere il codice JavaScript all'interno del documento, useremo i tag `<script>` e `</script>`. Posizioneremo il codice tra questi due tag in modo che il browser possa distinguere JavaScript dal resto del codice HTML o CSS nella pagina web.

Essendoci altri linguaggi di programmazione dal lato client (come ad esempio VBScript) è sempre meglio specificare quale intenderemo utilizzare. Lo faremo in questa maniera:

```
<script type="text/javascript">
// Il nostro primo listato Javascript
</script>
```

Possiamo inserire il nostro codice sia all'interno del tag `<head>` che all'interno del tag `<body>`.

Proviamo con un programmino semplice, creiamo un file HTML con questo

codice anche usando semplicemente il notepad, come già accennato inizialmente:

```
<!DOCTYPE html>
<html>
<head>
  <title>Il mio grandioso programma interno</title>
  <script type="text/javascript">
    let data = new Date();
    alert(data);
  </script>
</head>
<body>
</body>
</html>
```

Se apriamo questo file HTML nel nostro browser, vedremo una finestra di avviso che mostra la data corrente.

Come detto, possiamo inserire il nostro codice JavaScript esternamente in un file separato, e quindi aggiungere un link al file nel documento HTML. Ad esempio, se vogliamo mantenere lo stesso codice JavaScript dell'esempio precedente in un file esterno, possiamo copiare le seguenti linee in un nuovo file che chiameremo *datadioggi.js*:

```
let data = new Date();
alert(data);
```

Poi, possiamo collegare questo file nel nostro documento HTML in questo modo:

```
<!DOCTYPE html>
<html>
<head>
  <title>Il mio grandioso programma esterno</title>
  <script type="text/javascript" src="datadioggi.js"></script>
</head>
<body>
</body>
</html>
```

Salviamo questo file come *datadioggi.html*. Quindi, assicuriamoci che i file *datadioggi.html* e *datadioggi.js* siano posizionati nella stessa cartella, altrimenti dovremo specificare il percorso completo nel tag `<script>`.

Ovviamente starà a noi decidere se utilizzare il codice esternamente o internamente in base alla lunghezza del listato o anche alla quantità di pagine che lo utilizzano. Ad esempio, in quest'ultimo caso, se avessimo trenta

pagine con la medesima funzione, sarà sicuramente una buona idea quella di lasciare il codice fuori in un file .js, senza ripeterlo per trenta volte. È semplicemente una questione di praticità, pulizia e ordine, ma anche di logica.

Passiamo ad osservare alcuni dei tag più comuni che incontreremo:

Tag HTML da conoscere:

<code><!DOCTYPE html></code>	<code><!-- Inizio e fine del documento HTML --></code>
<code></html></code>	
<code><head></head></code>	<code><!-- Informazioni descrittive --></code>
<code><title></title></code>	<code><!-- Titolo della pagina --></code>
<code><body></body></code>	<code><!-- Corpo della pagina --></code>
<code><div></div></code>	<code><!-- Contenitore per varie parti di un sito suddivise per id --></code>
<code><header></header></code>	
<code><nav></nav></code>	
<code><main></main></code>	<i>Tag semantici per indicare chiaramente il significato e il ruolo dei vari elementi all'interno di una pagina web.</i>
<code><article></article></code>	
<code><footer></footer></code>	

Proprietà della pagina: sfondi e colori

<code><!-- Immagine di sfondo --></code>	<code><body background="URL"></code>
<code><!-- Colore di sfondo --></code>	<code><body bgcolor="#*****"></code>
<code><!-- Colore del testo --></code>	<code><body text="#*****"></code>
<code><!-- Colore dei collegamenti --></code>	<code><body link="#*****"></code>
<code><!-- Colore dei collegamenti cliccati --></code>	<code><body vlink="#*****"></code>
<code><!-- Colore del collegamento selezionato --></code>	<code><body alink="#*****"></code>

Formattazione del testo, tag validi e obsoleti

<code><!-- Neretto: --></code>	<code></code>	<i>(obsoleto)</i>
<code><!-- Corsivo: --></code>	<code><I></I></code>	<i>(obsoleto)</i>
<code><!-- Sottolineare: --></code>	<code><U></U></code>	<i>(obsoleto)</i>
<code><!-- Apice: --></code>	<code><SUP></SUP></code>	
<code><!-- Pendice: --></code>	<code><SUB></SUB></code>	
<code><!-- Macc.da scrivere: --></code>	<code><TT></TT></code>	<i>(obsoleto)</i>
<code><!-- Paragrafo: --></code>	<code><P></P></code>	
<code><!-- Preformattato: --></code>	<code><PRE></PRE></code>	<i>(compresi gli spazi)</i>
<code><!-- Centrare: --></code>	<code><CENTER></CENTER></code>	<i>(obsoleto)</i>
<code><!-- Dimensioni Font: --></code>	<code></code>	<i>(obsoleto)</i>
<code><!-- Dimensioni testo: --></code>	<code></code>	<i>(obsoleto)</i>
<code><!-- Colore Font: --></code>	<code></code>	<i>(obsoleto)</i>
<code><!-- Tipologia Font: --></code>	<code></code>	<i>(obsoleto)</i>

Molti di questi tag sono obsoleti o non raccomandati nelle versioni più recenti di HTML e CSS. Alcuni di essi sono stati sostituiti da approcci più moderni. Vediamo la revisione dei tag:

Neretto, Corsivo, e Sottolineare: questi tag sono validi in HTML, ma sono considerati obsoleti. È preferibile utilizzare CSS per applicare stili come il grassetto (*font-weight: bold;*), il corsivo (*font-style: italic;*), e il sottolineato (*text-decoration: underline;*) per una maggiore flessibilità e controllo.

Apice e Pendice: i tag <SUP> e <SUB> sono ancora validi e utilizzati per rappresentare il testo in apice o in pedice. Ad esempio, possono essere utili per scrivere formule chimiche o equazioni matematiche.

Macc. da scrivere (TypeWriter Text): il tag <TT> è obsoleto. È preferibile utilizzare CSS per controllare il tipo di carattere, ad esempio utilizzando il valore font-family appropriato.

Paragrafo: <P> è valido ed è utilizzato per definire un paragrafo di testo. È ancora comunemente utilizzato per strutturare il testo in paragrafi separati.

Preformattato: anche il tag <PRE> è tuttora utilizzato per visualizzare il testo preformattato, conservando gli spazi e il formato del testo originale. È utile per mostrare il codice sorgente o testo con una formattazione specifica.

Centrare testo e immagini: <CENTER> è obsoleto. È preferibile utilizzare CSS per centrare elementi, ad esempio con *text-align: center;* per il testo e *margin: 0 auto;* per le immagini.

Dimensioni Font: i tag con attributi SIZE sono obsoleti. È preferibile utilizzare CSS per definire le dimensioni del testo, ad esempio con *font-size*.

Dimensioni testo (+|-?): anche questi tag con attributi SIZE sono obsoleti. È preferibile utilizzare CSS per gestire le dimensioni del testo in modo più flessibile.

Colore Font: il tag con attributo COLOR è obsoleto. È preferibile utilizzare CSS per definire il colore del testo, ad esempio con *color*.

Tipologia Font: Il tag con attributo FACE è obsoleto. È preferibile utilizzare CSS per definire il tipo di carattere (font-family).

In generale, per il controllo dello stile e della formattazione, è altamente raccomandato utilizzare CSS invece di utilizzare i tag HTML obsoleti. CSS offre un maggiore controllo, flessibilità e separazione tra contenuto e

presentazione, migliorando la manutenibilità e l'accessibilità del codice HTML.

Abbiamo parlato brevemente di HTML, ci torneremo a breve, mentre ora è il momento di esplorare CSS3. L'acronimo CSS sta per *Cascading Style Sheets*, che in italiano significa "fogli di stile". Questo linguaggio svolge un ruolo cruciale nella gestione dell'aspetto e dello stile delle pagine web e collabora strettamente con HTML, il linguaggio che gestisce i contenuti delle pagine.

Con CSS3, è possibile definire regole per la formattazione e lo stile degli elementi web, tra cui font, colori, dimensioni e molto altro. Una caratteristica importante di CSS è il concetto di "cascata", che significa che le regole di stile possono essere sovrascritte in base a diverse fonti di stile. Questo meccanismo di cascata consente una maggiore flessibilità e controllo sul design delle pagine web.

Ci sono tre tipi principali di fogli di stile CSS:

- *Esterni*: i fogli di stile esterni sono file separati con estensione .css che contengono le regole di stile. Possono essere utilizzati su più pagine del sito web, garantendo uniformità nell'aspetto.
- *Interni*: sono definiti all'interno del documento HTML utilizzando il tag `<style>`. Questi stili si applicano solo alla pagina specifica in cui sono definiti.
- *In linea*: vengono applicati direttamente agli elementi HTML utilizzando l'attributo style. Questi stili hanno la priorità più alta e si applicano solo a un singolo elemento o tag.

È possibile utilizzare più fogli di stile contemporaneamente, e il sistema di cascata stabilisce l'ordine di priorità tra di essi. In generale, gli stili in linea hanno la precedenza sui fogli interni, mentre questi ultimi hanno la precedenza sui fogli esterni.

La creazione di un foglio di stile CSS3 e la scrittura delle regole avvengono mediante l'utilizzo di un semplice editor di testo. Per far sì che le regole di stile vengano applicate alle pagine web, è necessario collegare il foglio di stile al documento HTML utilizzando il tag `<link>` nell'elemento `<head>` del documento.

Ecco un esempio di regola CSS3 applicata al titolo della pagina, che crea un effetto di ombra sul testo:

```
#titolo {  
  text-shadow: 5px 5px 5px #FF0000;  
}
```

Con questa regola, il titolo della pagina avrà un'ombra, creando un effetto visivo interessante. Continueremo ad esplorare altre caratteristiche e tecniche di CSS3 in esempi futuri. Per approfondire l'argomento ed esplorare l'enorme quantità di possibilità offerte consiglio una ricerca su internet, come queste pagine web dell'organizzazione Mozilla:

https://developer.mozilla.org/it/docs/Learn/Getting_started_with_the_web/CSS_basics.

Per concludere, proseguiamo con un breve ripasso su HTML5 e le sue peculiarità. Partiamo dagli elementi semantici di HTML5, i quali hanno introdotto nuove tag come `<header>`, `<nav>`, `<main>`, `<section>`, ecc. Questi forniscono una struttura più chiara e significativa per il contenuto della pagina web, migliorando l'accessibilità e il SEO. Ad esempio, `<header>` rappresenta l'intestazione di una sezione, `<nav>` il menu di navigazione, `<main>` il contenuto principale della pagina, `<article>` un contenuto autonomo e così via. Oltre a questo sono stati introdotti elementi fondamentali come `<audio>` e `<video>` che consentono di incorporare facilmente file audio e video nelle pagine web senza dipendere da plugin esterni, e si può utilizzare JavaScript per interagire con i media incorporati. Non tutti sanno che l'elemento `<canvas>` offre un'area in cui è possibile disegnare grafici, animazioni e visualizzazioni direttamente nel browser, creare grafici dinamici, giochi e altre applicazioni grafiche interattive.

Si può accedere alla posizione geografica dell'utente attraverso il browser con l'API *Geolocation*. HTML5 offre anche soluzioni di archiviazione locale, come *localStorage* e *sessionStorage*, che consentono di memorizzare dati nel browser dell'utente, come le preferenze, i dati di sessione o qualsiasi altra informazione che deve persistere tra le visite. Vi sono nuovi tipi di input come *date*, *email*, *number*, `<textarea>` e altre opzioni avanzate per i moduli web, e molti framework e librerie JavaScript popolari, come React, Angular e Vue.js, sono compatibili con HTML5.

Adesso vediamo un esempio riepilogativo:

```
<!DOCTYPE html>  
<html>  
<head>  
  <meta charset="UTF-8">  
  <title>Esempio HTML5</title>
```

```

<style>
  /* Stile per il componente Web personalizzato */
  my-custom-element {
    color: blue;
  }
</style>
</head>
<body>
  <!-- Elementi Semantici -->
  <header>
    <h1>Il mio sito web</h1>
  </header>
  <nav>
    <ul>
      <li><a href="#">Home</a></li>
      <li><a href="#">Servizi</a></li>
      <li><a href="#">Contatti</a></li>
    </ul>
  </nav>
  <main>
    <article>
      <h2>Articolo Interessante</h2>
      <p>Questo è un articolo molto interessante.</p>
    </article>
  </main>
  <footer>
    <p>&copy; 2023 Il mio sito web</p>
  </footer>

  <!-- Elementi Multimediali -->
  <audio controls>
    <source src="audio.mp3" type="audio/mpeg">
    Il tuo browser non supporta l'audio.
  </audio>
  <video controls width="320" height="240">
    <source src="video.mp4" type="video/mp4">
    Il tuo browser non supporta il video.
  </video>

  <!-- Canvas e Grafica -->
  <canvas id="myCanvas" width="400" height="200"></canvas>

  <!-- Geolocalizzazione -->
  <script>
    function getLocation() {
      if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(function(position)
{
          alert('Latitudine: ' + position.coords.latitude + ',
Longitude: ' + position.coords.longitude);

```

```

        });
    } else {
        alert('La geolocalizzazione non è supportata dal tuo
browser.');
```

```
    }
}
```

```
</script>
```

```
<button onclick="getLocation()">Ottieni Posizione</button>
```

```
<!-- Archiviazione Locale -->
```

```
<script>
```

```
    localStorage.setItem('preferenza', 'valore');
```

```
    let preferenza = localStorage.getItem('preferenza');
```

```
    console.log('Preferenza:', preferenza);
```

```
</script>
```

```
<!-- Formulazioni Avanzate -->
```

```
<form>
```

```
    <label for="email">Email:</label>
```

```
    <input type="email" id="email" name="email" required>
```

```
    <br>
```

```
    <label for="data">Data di nascita:</label>
```

```
    <input type="date" id="data" name="data">
```

```
    <br>
```

```
    <textarea placeholder="Inserisci il tuo commento"></textarea>
```

```
</form>
```

```
<!-- Accessibilità -->
```

```
    
```

```
    <p id="descrizione-immagine">Questa è un'immagine significativa
con una descrizione.</p>
```

```
<!-- Web Component -->
```

```
    <my-custom-element>Questo è un componente personalizzato.</my-
custom-element>
```

```
<!-- Framework JavaScript -->
```

```
    <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js">
</script>
```

```
    <div id="app">
```

```
        {{ message }}
```

```
    </div>
```

```
    <script>
```

```
        new Vue({
            el: '#app',
            data: {
```

```
                message: 'Questa è una pagina Vue.js'
```

```
            }
        });
```

```
</script>
</body>
</html>
```

Questo esempio di codice HTML5 mostra una pagina web completa che utilizza una varietà di funzionalità avanzate offerte da HTML5, CSS3 e JavaScript. Ecco una spiegazione dettagliata di ciascun aspetto dell'esempio:

`<!DOCTYPE html>`: dichiara il tipo di documento come HTML5, il quale è la versione più recente di HTML.

`<html>`: elemento radice del documento HTML.

`<head>`: contiene metadati e collegamenti a fogli di stile e script. È importante notare che hai impostato il set di caratteri dell'intero documento su UTF-8, il che è una pratica comune per garantire la corretta visualizzazione di caratteri speciali.

`<title>`: definisce il titolo della pagina visualizzato nella barra del browser.

`<style>`: incorpora le regole CSS direttamente nel documento. In questo caso, è definito uno stile per un componente personalizzato `<my-custom-element>`, impostando il colore del testo su blu.

`<body>`: è la parte principale del tuo documento HTML, contenente tutto il contenuto visibile nella pagina.

Elementi semantici: rappresentano parti strutturali della pagina:

`<header>`: l'intestazione della pagina con un titolo principale.

`<nav>`: il menu di navigazione con collegamenti a "Home", "Servizi" e "Contatti".

`<main>`: la sezione principale del contenuto della pagina.

`<article>`: un articolo con un titolo e un paragrafo di testo.

`<footer>`: il piè di pagina con l'anno di copyright.

Elementi multimediali: consentono di incorporare media nella pagina:

`<audio>`: un lettore incorporato con un pulsante di controllo e una sorgente audio.

`<video>`: un lettore incorporato con un pulsante di controllo, larghezza e altezza personalizzate e una sorgente video.

Canvas e grafica: un elemento `<canvas>` può essere utilizzato per disegnare grafica dinamica tramite JavaScript.

Geolocalizzazione: si usa JavaScript per ottenere la posizione geografica dell'utente quando viene fatto clic su un pulsante "Ottieni Posizione".

Archiviazione locale: con *localStorage* di JavaScript si può archiviare una preferenza utente e recuperarla successivamente.

Formulazioni avanzate: un modulo con campi di input avanzati, inclusi input email e input data.

Accessibilità: è incluso un'immagine con un attributo *alt* significativo per garantire l'accessibilità, utilizzando *aria-describedby* per fornire una descrizione aggiuntiva per l'immagine.

Web component: un componente personalizzato `<my-custom-element>` dimostra l'uso di componenti Web personalizzati all'interno di HTML5.

Framework JavaScript: è presente il framework Vue.js per la creazione di interfacce utente dinamiche.

In sintesi, questo esempio rappresenta un'applicazione completa che fa uso delle funzionalità avanzate di HTML5, CSS3 e JavaScript per creare una pagina web interattiva e accessibile.

Check del codice con le asserzioni

Nei prossimi capitoli, esamineremo il funzionamento effettivo degli argomenti di studio utilizzando le cosiddette asserzioni per verificare che il codice che stiamo studiando funzioni come previsto e per testare il codice in generale. Uno degli strumenti che ci permetterà di farlo è la funzione *assert*, il cui scopo è affermare se una condizione è vera o falsa. È importante notare che, mentre Firebug offre un modo comodo per utilizzare la funzione *assert* nel browser Firefox, questa funzione è una parte standard di JavaScript ed è utilizzabile senza dipendere da strumenti specifici come Firebug.

Per utilizzare *assert*, è sufficiente fornire una condizione come primo parametro e un messaggio come secondo. Ad esempio:

```
let A = 5;  
assert(A > 3, 'A è maggiore di 3, quindi abbiamo un problema..');
```

Se la condizione fornita è vera, l'asserzione passa senza problemi e il messaggio non viene visualizzato. Se è falsa, l'asserzione fallisce e il messaggio verrà visualizzato per indicare un problema nel codice.

Se per qualche motivo non si desidera utilizzare la funzione *assert*, è possibile sostituirla con *console.assert()*. Questa funzione è disponibile nelle

console dei browser moderni ed è utile per verificare condizioni specifiche durante il debugging del codice. Nel prossimo esempio osserviamo la differenza tra *console.assert* e *console.log*:

```
let x = 2;
let y = 2;

console.assert(x + y == 5, "Risultato sbagliato");

let A = 5;

console.log(A > 3, 'Purtroppo A è maggiore di 3, quindi abbiamo un problema..');
```

console.assert è specificamente progettato per il testing e il debugging, mentre *console.log* è utilizzato per registrare messaggi di debug o informazioni durante l'esecuzione del programma. Entrambe le funzioni sono utili a seconda del contesto e delle esigenze di debugging.

Quiz & esercizi

- 1) Nella creazione di un listato dove andremo a posizionare i file relativi ad un applicazione “esterna”?
- 2) Qual è il tipo di estensione per i file legati ad applicazioni esterne?
- 3) Creare una pagina HTML con del testo di almeno 5 colori diversi.
- 4) Applicare alla pagina HTML creata sopra l’ombra a tutto il testo.
- 5) Modificare il codice affinché l’asserzione riporti *true* (** = *elevare a potenza*):

```
let x = 2;
let y = 10;

console.assert(x ** y == 50, "Risultato sbagliato");
```

- 6) Aggiungi un commento all'immagine nell'esempio a pag 33 per descriverla in modo significativo e migliorare la sua accessibilità.
- 7) Crea un nuovo paragrafo all'interno della sezione principale dell'esempio a pag 33 e scrivi del testo al suo interno.
- 8) Aggiungi una lista ordinata di almeno tre elementi al menu di navigazione nell'esempio a pag 33.
- 9) Crea un nuovo componente personalizzato chiamato `<my-custom-element>` e definisci uno stile CSS per modificarne il colore del testo.

- 10) Modifica l'app Vue.js nell'esempio a pag 33 per visualizzare un messaggio diverso nell'elemento con id "app".
- 11) Utilizza JavaScript per ottenere la posizione geografica dell'utente quando viene caricata la pagina anziché solo al clic del pulsante.
- 12) Crea un nuovo modulo con un campo di input per l'indirizzo e uno per il numero di telefono nell'esempio a pag 33.
- 13) Aggiungi un commento all'elemento <video> nell'esempio a pag 33 per indicare cosa rappresenta il video.
- 14) Aggiungi un commento all'elemento <audio> nell'esempio a pag 33 per indicare cosa rappresenta l'audio.
- 15) Modifica il messaggio Vue.js nell'esempio a pag 33 in modo che sia basato su una variabile definita nel JavaScript anziché essere una stringa statica.
- 16) Quale attributo HTML viene utilizzato per collegare un file JavaScript a una pagina HTML?
- 17) Come si dichiara una variabile in JavaScript?
- 18) Dichiara una variabile chiamata nome e assegna il valore "Alice".
- 19) Qual è il metodo JavaScript utilizzato per selezionare un elemento HTML in base al suo ID?
- 20) Come si stampa un messaggio nella console del browser utilizzando JavaScript?
- 21) Stampa la frase "Hello, world!" nella console del browser utilizzando *console.log()*.
- 22) Qual è il risultato dell'operazione $2 + 2$ in JavaScript?

Riassunto

Abbiamo iniziato a capire come lo sviluppo di Javascript sia sempre in costante evoluzione grazie alle varie edizioni dell'ECMA che, dal 2015, ogni anno aggiungono e migliorano le funzionalità di Javascript.

Gli strumenti e l'ambiente di lavoro saranno le prime cose che esploreremo ed utilizzeremo per addentrarci nello studio. La panoramica iniziale descrive alcune librerie tra le più conosciute ed utilizzate (jQuery) e altre meno ma con grandi potenzialità (Three.js).

L'HTML ci accompagnerà spesso in questo viaggio e dovremo averne una conoscenza minima, almeno all'inizio. Infine con le asserzioni potremo verificare diverse cose nel nostro codice.

2 - Le prime basi

*L'ordine degli eventi
Ancora sprazzi di HTML qui e là
La sequenza del codice JS
Il ciclo degli eventi
DOM e BOM*

Prima di iniziare il capitolo pensiamo al nostro browser, a tutte le volte che lo abbiamo eseguito e a quanto abbiamo atteso i caricamenti dei vari siti. In qualche caso, dopo un nostro click avremmo visto la fase di costruzione della pagina immaginando il codice che stava dietro. Sicuramente in pochi si saranno soffermati su questi aspetti, soprattutto negli ultimi anni, dove la velocità (con la fibra ottica) e l'evoluzione verso il web 3.0 sta portando a diversi cambiamenti ma anche a standardizzazioni di molti siti web. Tuttavia, nonostante i numerosi futuri progressi, lo studio e l'apprendimento di queste fasi fondamentali non è cambiato. Parliamo dell'ordine della sequenza di eventi o “*ciclo di vita*”, e, in particolare, di come il nostro codice JavaScript ci si relaziona. Il ciclo di vita inizia nel momento in cui la pagina viene richiesta, attraverso le interazioni eseguite dall'utente, fino alla chiusura della stessa. Inizieremo esplorando come viene costruita la pagina elaborando il codice HTML. Procederemo focalizzandoci sull'esecuzione del codice JavaScript, esamineremo come vengono gestiti gli eventi e infine, durante questo percorso, esploreremo alcuni concetti fondamentali delle applicazioni web come il BOM, il DOM e il ciclo di eventi.

Impariamo l'ordine

L'ordine (o la successione regolare di una serie di eventi) in un listato Javascript è molto importante e, parlando dell'esempio più semplice, cioè un sito internet, deve interfacciarsi con le API, il codice HTML, il CSS e così via. Inizia con l'utente che digita un URL nella barra degli indirizzi del browser o fa clic su un collegamento. Se volessimo dare un'occhiata alle

ultime news dal mondo, apriremo il nostro browser e digiteremo, ad esempio, l'indirizzo `www.corriere.it`, a questo punto il browser, fungendo da tramite, formulerà una richiesta che verrà inviata a un server, che la elaborerà, inoltrando poi una risposta. Nel momento in cui il browser riceve questa risposta l'applicazione web inizierà il suo ciclo di vita. L'ordine di tutti questi eventi ha una sua sequenza, un suo ordine, che impareremo a conoscere e che bisognerà tenere a mente.

Poiché le applicazioni web lato client (come i siti internet) sono applicazioni che si avvalgono di una GUI (Graphical User Interface, tradotto: interfaccia grafica), il loro ciclo di vita è contraddistinto da passaggi del tutto analoghi. Ad esempio, un programma per la videoscrittura, per il fotoritocco o anche un videogame, viene eseguito in due fasi, la prima sarà la costruzione della pagina, dove l'interfaccia utente sarà modulata al meglio per rispondere allo scopo del programma. La seconda sarà la gestione degli eventi, dove una serie di questi resta sopita, finché l'utente non esegue le sue richieste, innescando così i gestori degli eventi.

Il ciclo di vita dell'applicazione termina quando l'utente termina e chiude il programma o la pagina del sito sul nostro browser.

Andiamo nello specifico: nel linguaggio Javascript, il ciclo di vita della *funzione* ha 3 fasi: dichiarazione, inizializzazione e assegnamento. Ad esempio:

```
function funzione1() { }  
funzione1();
```

Le 3 fasi vengono eseguite all'inizio della funzione che la contiene, per cui la sua posizione non è importante.

```
funzione2();  
function funzione2() { }
```

Quando il codice viene seguito, *funzione2()* passa tutte e tre le fasi all'inizio dello *scope* (cioè l'ambito di validità) in cui è racchiuso il codice. Quindi, la funzione può essere utilizzata ovunque nel programma, ma questo lo vedremo nel dettaglio nei prossimi capitoli.

Andiamo avanti. Anche gli eventi stessi, di cui abbiamo parlato pocanzi, hanno un ciclo di vita, un loro ordine. Precisamente sono tre fasi:

- la fase di *cattura* dell'evento: durante questa fase, l'evento si propaga dalla radice del DOM all'elemento di destinazione. È la fase di "cattura" in cui è possibile intercettare l'evento prima che raggiunga il suo obiettivo.
- la fase *obiettivo* dell'evento: qui l'evento raggiunge l'elemento di destinazione e scatena il suo comportamento predefinito o gli eventi associati.
- la fase cosiddetta di "*bolitura*" dell'evento: dopo la fase obiettivo, l'evento risale dal punto di destinazione verso la radice del DOM. Durante questa fase, è possibile ancora intercettare l'evento.



L'immagine appena vista illustra come si sviluppa il ciclo di un classico evento come il click di un pulsante su schermo. Inizialmente l'istruzione addetta all'ascolto cattura ed indirizza il nostro evento verso l'area o l'elemento designati, dopodiché, nella fase di "bolitura" l'evento si unirà al codice associato alla sua cattura. Vediamo un esempio su cui dovremo tornare per capire appieno tutti i meccanismi:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Esempio Event Listener</title>
</head>
<body>
  <div id="outer">
    <div id="inner">
      <button id="button">Cliccami</button>
    </div>
  </div>
```

```

<script type="text/javascript">
    document.getElementById('outer').addEventListener('click',
function() {
    alert('Outer Div');
});

    document.getElementById('inner').addEventListener('click',
function() {
    alert('Inner Div');
});

    document.getElementById('button').addEventListener('click',
function() {
    alert('Button');
});
</script>
</body>
</html>

```

In questo esempio, ci sono tre elementi HTML nidificati: un div esterno con id "outer", un *div interno* con id "inner", e un pulsante con id "button". Abbiamo aggiunto tre *event listeners* per l'evento click su ciascuno di questi elementi.

L'evento click sul div esterno ("outer") è gestito prima di tutto. Quando si fa clic su qualsiasi parte del div esterno, verrà visualizzato l>alert "Outer Div".

L'evento click sul div interno ("inner") è gestito dopo l'evento sul div esterno. Quindi, se si fa clic sulla parte interna del div ("inner"), verrà visualizzato prima "Outer Div" e poi "Inner Div".

L'evento click sul pulsante ("button") è gestito dopo gli eventi sui div. Quando si fa clic sul pulsante, verrà visualizzato l>alert "Button".

Questo esempio illustra chiaramente il concetto di propagazione degli eventi nell'albero DOM, dove questi vengono catturati nell'ordine in cui gli elementi sono annidati. È un modo efficace per capire come gli eventi si propagano e quali *event listeners* vengono attivati quando si interagisce con gli elementi HTML.

Proseguiamo con lo stesso argomento osservando la sequenza di eventi che si avvia ogni volta che una pagina web viene eseguita.

La sequenza di eventi in una pagina web

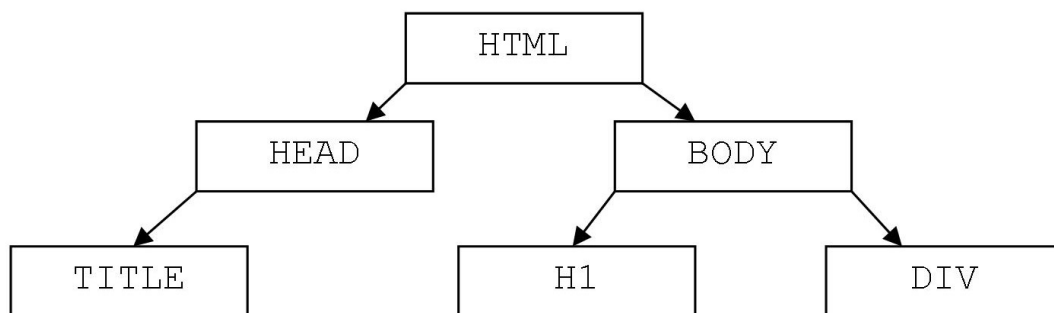
Quando apriamo il browser e scriviamo un sito sulla barra degli indirizzi dando poi l'invio, prima ancora che la nostra pagina sia visualizzata e che noi possiamo interagirci, essa dovrà essere “costruita” mediante le

informazioni contenute nel codice HTML, incluse le istruzioni dei CSS (fogli di stile, essenzialmente l'aspetto della pagina, come visto nel precedente capitolo) e ovviamente il codice JavaScript. Precisamente questo è quello che accade in sequenza:

- 1) *Analisi dell'HTML e creazione del DOM (Document Object Model, ne parliamo brevemente più avanti nel capitolo)*
- 2) *Esecuzione dei CSS*
- 3) *Esecuzione del codice JavaScript*

Il passaggio 1 e 2 vengono eseguiti quando il browser elabora i nodi HTML e il passaggio 3 viene eseguito ogni volta che si incontra un tipo speciale di elemento HTML, l'elemento script (che contiene o fa riferimento al codice JavaScript). Durante la fase di creazione della pagina, il browser può alternare questi tre passaggi tutte le volte che è necessario.

Il DOM è una rappresentazione strutturata della pagina HTML in cui ogni elemento è rappresentato come nodo. Ad esempio, si può vedere come viene visualizzato schematicamente prima dell'esecuzione di uno script:



Si può notare che come questo tipo di struttura ad albero suddivisa in oggetti *padre*, da cui si generano oggetti *figlio* e i *rami* che li collegano, sia di facile lettura e comprensione.

Durante la costruzione della pagina, il browser può incontrare un tipo speciale di elemento HTML, l'elemento script, che, come abbiamo già visto,

viene utilizzato per includere il codice JavaScript. Quando ciò accade, il browser sospende la costruzione del DOM dal codice HTML e avvia l'esecuzione del codice JavaScript.

Tutto il codice JavaScript contenuto nell'elemento script viene eseguito dal motore JavaScript del browser; ad esempio, Spidermonkey di Firefox, Chrome e Opera's V8 o Chakra di Edge (IE). Poiché lo scopo principale del codice JavaScript è fornire dinamicità alla pagina, il browser fornisce un'API tramite un *oggetto globale* che può essere utilizzato dal motore JavaScript per interagire e modificare la pagina.

L'oggetto globale principale che il browser espone al motore JavaScript è l'oggetto *finestra* (o meglio: *window*), dove è contenuta una pagina. In essa sono accessibili tutti gli altri oggetti globali, le variabili e le API del browser. Una delle proprietà più importanti dell'oggetto finestra globale è il documento, che rappresenta il DOM della pagina corrente. Utilizzando questo oggetto, il codice JavaScript può alterare il DOM della pagina in qualsiasi misura, modificando o rimuovendo elementi esistenti, e persino creandone e inserendone di nuovi.

```
<body>
  <div id="outer">
    <div id="inner">
      <button id="button">Cliccami</button>
    </div>
  </div>
  <script type="text/javascript">
    document.getElementById('outer').addEventListener('click',
function() {
    alert('Outer Div');
});
```

Questa porzione di codice vista in precedenza utilizza l'oggetto *documento globale* per selezionare prima un elemento con l'ID dal DOM (in questo caso l'elemento 'outer') e eseguendo poi un'azione (verrà visualizzato un alert con il testo "Outer Div"). Possiamo quindi utilizzare il codice JavaScript per apportare tutti i tipi di modifiche a quell'elemento, come cambiare il suo contenuto testuale, modificare i suoi attributi, creare dinamicamente e aggiungere nuovi *figli* ad esso e persino rimuovendo l'elemento dal DOM.

Codice globale e codice funzione

Abbiamo visto gli oggetti globali, adesso vedremo che ci sono due tipi differenti di codice in Javascript: il codice globale e il codice funzione:

```

<script>
let a = 3
function aggiungisei(x) {
  let ret = x + 6          <- Codice funzione
  return ret
}
let b = aggiungisei(a)
console.log(b)
</script>

```

La principale differenza tra questi due tipi di codice è la loro posizione: il codice contenuto in una funzione è chiamato *codice funzione*, cioè quella porzione di codice utilizzata ogni volta che il flusso di esecuzione entra nel corpo di una funzione. Mentre il codice posto all'esterno di tutte le funzioni è chiamato *codice globale*, cioè quell'ambiente predefinito in cui il codice viene eseguito per la prima volta.

Ogni volta che eseguiamo un listato, JavaScript crea un nuovo contesto di esecuzione locale che avrà il proprio insieme di variabili, le quali saranno locali a quel contesto di esecuzione. Il nuovo contesto di esecuzione verrà gettato nello stack di esecuzione. Pensiamo allo stack di esecuzione come a un meccanismo per tenere traccia di dove si trova il programma nella sua esecuzione. Quindi i contesti di esecuzione locali vengono visualizzati dallo stack di esecuzione. Le funzioni restituiscono il valore restituito al contesto chiamante. Il contesto chiamante è il contesto di esecuzione che ha chiamato questa funzione, potrebbe essere il contesto di esecuzione globale o un altro contesto di esecuzione locale. Spetta al contesto di esecuzione chiamante gestire il valore restituito a quel punto. Il valore restituito potrebbe essere un oggetto, un vettore (insieme o array), una funzione, un booleano (cioè *true* o *false*), qualsiasi cosa. Se la funzione non ha l'istruzione *return*, verrà restituito *undefined*.

Il contesto di esecuzione locale viene distrutto. Questo è importante. *Distrutto*. Tutte le variabili dichiarate all'interno del contesto di esecuzione locale vengono cancellate. Non sono più disponibili. Ecco perché vengono chiamate variabili locali.

Esecuzione del codice Javascript

Quando il browser raggiunge il nodo dello script nella fase di creazione della pagina, mette in pausa la costruzione del DOM basato sul codice HTML e inizia invece l'esecuzione di codice JavaScript:

```

<script>
function esecuzione(){aggiunta('Devo studiare!');
}
function aggiunta(altro)
{alert('Questo è l\'argomento ricevuto da esecuzione(): ' +
altro);
}
    esecuzione();
</script>

```

Ciò significa eseguire il codice JavaScript globale contenuto nell'elemento script (e vengono eseguite anche le funzioni chiamate dal codice globale). Una volta eseguita l'ultima riga di codice nello script, il browser esce dalla modalità di esecuzione e continua a costruire nodi nel DOM elaborando il codice HTML rimanente. Se, durante tale elaborazione, il browser incontra nuovamente un elemento di script, la creazione del DOM dal codice HTML viene nuovamente sospesa e il motore di JavaScript inizia a eseguire il codice contenuto. È importante notare che lo stato globale dell'applicazione JavaScript persiste nel frattempo. Tutte le variabili globali definite dall'utente create durante l'esecuzione del codice in un elemento dello script sono normalmente accessibili al codice JavaScript in altri elementi dello script. Ciò accade perché l'oggetto *window*, che memorizza tutte le variabili globali, è attivo e accessibile durante l'intero ciclo di vita della pagina. Vediamo questo codice che ci renderà più chiari i concetti esposti:

```

<html>
  <head>
    <title>Prova codice HTML+JS</title>
  <style>
#z3 {text-shadow: 5px 5px 5px #FF0000;}
  </style>
  </head>
  <body>
    <p id="z2">Paragrafo 10</p>
    <p id="z3">Paragrafo 30</p>
  </body>
  <script>
    let elmnt = document.getElementById("z2");
    elmnt.remove();
  </script>
</html>

```

Come abbiamo appena osservato in questo listato inizialmente il DOM viene costruito con 2 nodi: z2 e z3, poi Javascript ne eliminerà uno, per cui il DOM

verrà aggiornato mostrando solamente z3.

Riassumendo, la fase di costruzione del DOM a partire dall'HTML, e l'esecuzione del codice JavaScript, vengono ripetuti fintanto che ci sono elementi HTML da elaborare e codice JavaScript da eseguire. Infine, quando il browser esaurisce gli elementi HTML da elaborare, la fase di creazione della pagina sarà completa. A questo punto, dato che anche lo script è terminato, il browser passerà senza ulteriori indugi alla gestione degli eventi. In questo particolare stato ciò che si trova all'interno della nostra *window* sarà sensibile e reattiva a diversi tipi di eventi: movimenti del mouse, clic, pressioni sulla tastiera e così via. Per questo motivo il codice JavaScript eseguito durante la fase di costruzione della pagina, oltre ad influenzare lo stato globale dell'applicazione e modificare il DOM, può registrare anche funzioni che vengono eseguite dal browser quando un evento si verifica. Con questi gestori di eventi, offriremo interattività alle nostre applicazioni. Ma prima di dare un'occhiata da più vicino alla registrazione dei gestori di eventi, esaminiamo i principi generali che regolano la gestione degli eventi.

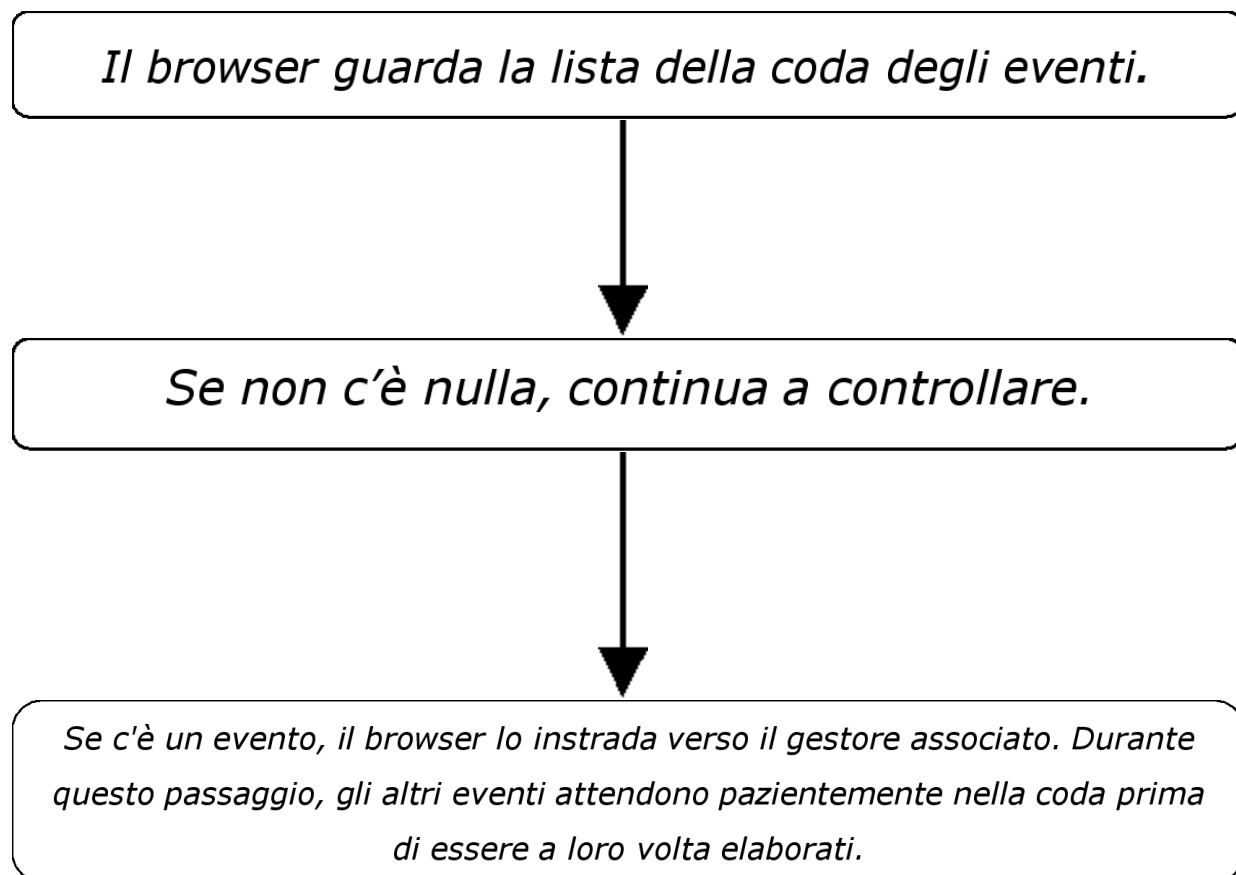
La “coda” degli eventi

Abbiamo imparato come nel browser ci sia una sequenza ben precisa per gestire la costruzione di una pagina. Forse dagli esempi estremamente brevi non si riesce a capire che, nel caso appunto di qualcosa di molto più complesso, ci possa essere un po' di intasamento. Immaginiamo un'autostrada dove si scorre velocemente fino ai caselli, dove purtroppo ne funziona solamente uno. Finchè le auto sono poche non è un problema, ma quando il traffico aumenta, come in un programma complesso, gli eventi automobili finiscono per incrementare una lunga fila al casello che deve processarli uno per uno.

Quindi, seguendo la metafora, il nostro casello autostradale browser è sempre in attesa dell'evento automobile e al momento in cui arriverà sarà pronto ad eseguire la funzione di gestore di eventi associata. Per evitare confusione e garantire uno scorrere fluido del traffico, il browser necessita di un modo per tenere traccia degli eventi sia di quelli che si sono verificati sia di quelli che devono ancora accadere ed essere elaborati. Per fare ciò, il browser utilizza una *coda degli eventi*.

In questo elenco saranno inserite tutte quelle azioni dirette provocate da un utente quali click su un pulsante, scorrimento della pagina, digitazione di un

indirizzo internet, ma anche eventi indiretti, quindi slegate dall'azione di un utente, come un timer, il caricamento di una foto o di una pagina. Tutti verranno inseriti nella stessa lista, nell'ordine in cui sono rilevati dal browser. Potremo rappresentarlo graficamente in questo modo:



Dato che verrà elaborato un solo evento alla volta, dovremmo prestare la massima attenzione alla quantità di tempo necessaria per la loro gestione, se dovessero richiedere molto tempo per essere eseguiti le applicazioni web ne risentirebbero negativamente.

È importante notare che il meccanismo del browser che inserisce gli eventi nella coda è esterno alle fasi di creazione della pagina. L'elaborazione necessaria per determinare quando essi si sono verificati non partecipa al *thread* che gestisce gli eventi.

Possono capitare in momenti e in un ordine imprevedibili (è difficile costringere gli utenti a premere i tasti o fare clic in un ordine particolare). Diciamo che la gestione degli eventi, e quindi l'invocazione delle loro funzioni di gestione, è *asincrono*, cioè non segue un elenco predefinito.

Gli eventi più comuni di solito sono i seguenti:

Utente, come l'uso della tastiera, il clic o lo spostamento del mouse.

Del browser, come quando una pagina è stata caricata.

Di rete, come le risposte provenienti dal server.

Praticamente la totalità del nostro codice verrà eseguita come risultato di questi avvenimenti, e, prima che essi possano essere gestiti, il programma dovrà notificare al browser che stiamo interagendo nella gestione di eventi particolari.

In attesa di un evento

In pratica stiamo descrivendo e sezionando un processo che normalmente si svolge in pochi secondi. Riassumendo siamo partiti dalla pagina web che si apre nella nostra finestra (*window*) del browser, ora che il sito si è caricato i cosiddetti “gestori degli eventi” attenderanno una nostra azione. Quando la effettueremo, come ad esempio cliccare su un link, si eseguirà un gestore di eventi, che altro non è che una funzione con del codice Javascript. Affinché ciò avvenga, dobbiamo avvisare il browser che quella condizione si è verificata. Questo passaggio possiamo chiamarlo, traducendo dall'inglese, registrazione della gestione di un (possible) evento. Per far sì che esso si verifichi, trovandoci dalla parte client di un'applicazione web, abbiamo due possibilità, cioè:

- Assegnare una proprietà ad una funzione assegnata.
- Usare il metodo *addEventListener* (ascoltatore di eventi).

Vediamo un paio di esempi relativi alle funzioni:

```
window.onmousemove = function(prova) {};  
document.body.onclick = function(altraprova) {};
```

Nella prima riga assegniamo la proprietà *onmousemove* alla funzione *prova* dell'oggetto finestra (*window*), cioè in pratica viene *registrato un gestore di eventi* per il movimento del mouse. Allo stesso modo, nella seconda riga inseriamo un registratore per gestire l'evento click sul corpo (*body*) del documento che chiamerà la funzione *altraprova*. Per essere precisi, è importante notare che nel codice appena visto stai sovrascrivendo direttamente le funzioni degli eventi *onmousemove* e *onclick* sull'oggetto

window e *document.body*. Questo significa che qualsiasi eventuale gestore degli eventi precedente associato a questi eventi verrà sovrascritto. Se sei sicuro di volerlo fare, allora il codice è corretto. Tuttavia, se vuoi aggiungere gestori degli eventi *senza rimuovere quelli esistenti*, è consigliabile utilizzare il metodo *addEventListener*. Ecco come puoi farlo:

```
window.addEventListener("mousemove", function(prova) {  
    // Il codice del gestore dell'evento per il movimento del mouse  
});  
  
document.body.addEventListener("click", function(altraprova) {  
    // Il codice del gestore dell'evento per il click del mouse sul  
body  
});
```

Quindi, il metodo *addEventListener* ci consentirà di registrare tutte le funzioni di gestione degli eventi di cui avremo bisogno, vediamo un esempio più dettagliato:

```
document.addEventListener("DOMContentLoaded", function() {  
    let movimento = document.getElementById("movimento");  
    let clic = document.getElementById("clic");  
  
    document.body.addEventListener("mousemove", function() {  
        console.log(movimento, "Il mouse si sta muovendo");  
    });  
  
    document.body.addEventListener("click", function() {  
        console.log(clic, "Click del mouse");  
    });  
});
```

Questo esempio mostra il metodo integrato *addEventListener* su un elemento HTML per specificare il tipo di evento (*mousemove* o *click*) e la funzione del gestore di eventi. Ciò significa che ogni volta che il mouse viene spostato sulla pagina, il browser chiama una funzione che aggiunge il messaggio, *"Il mouse si sta muovendo"* mentre il testo *"Click del mouse"* viene aggiunto ogni volta che si fa click.

Il verificarsi di un evento

Come già detto, quando si verifica un evento, il browser chiama il gestore associato, e, a causa del modello di esecuzione a thread singolo, è possibile eseguire un solo gestore di eventi contemporaneamente. Tutte le circostanze seguenti vengono elaborate solo dopo che l'esecuzione del gestore eventi corrente è stata completata!

```
document.body.onclick = function(event) {
    let isRightMB;
    event = event || window.event;

    if ("which" in event) {
        // Firefox, Safari, Chrome e Opera
        isRightMB = event.which == 3;
    } else if ("button" in event) {
        // IE, Opera
        isRightMB = event.button == 2;
    }

    alert("Il pulsante destro " + (isRightMB ? "" : " non è stato") +
    " cliccato!");
}
```

Esaminiamo cosa sta succedendo qui. Al click del mouse si passa alla fase di gestione degli eventi, il ciclo di eventi controlla quindi la coda, vede che c'è un evento legato al click del mouse all'inizio della coda ed esegue il gestore associato. Questo ciclo continuerà a essere eseguito fino a quando l'utente non chiuderà l'applicazione Web.

Approfondire il ciclo degli eventi

Il ciclo di eventi è più complicato di come lo abbiamo descritto finora. Per cominciare, invece di una singola coda che contiene solo eventi, il ciclo ha almeno due code che contengono altre azioni eseguite dal browser. Queste azioni sono chiamate attività e sono raggruppate, come già detto, in due categorie: *macrotask* (o spesso chiamate semplicemente attività) e *microtask*. Esempi di *macrotask* includono la creazione dell'oggetto documento principale, l'analisi HTML, esecuzione del codice JavaScript principale (o globale), modifica dell'URL corrente, nonché vari eventi come caricamento della pagina, input, eventi di rete ed eventi timer. Dal punto di vista del browser, una *macrotask* rappresenta un'unità di lavoro autonoma. Dopo aver eseguito un'attività, il browser può continuare con altre assegnazioni come rieseguire il rendering dell'interfaccia utente della pagina o eseguire la garbage collection.

I *microtask*, d'altra parte, sono attività più piccole che aggiornano lo stato dell'applicazione e devono essere eseguite prima che il browser continui con altre assegnazioni, come il rendering dell'interfaccia utente. Gli esempi includono callback promesse e modifiche alla mutazione DOM. Le *microtask* dovrebbero essere eseguite il prima possibile, in modo asincrono, ma senza il costo di eseguire una *macrotask* completamente nuova. I

microtask ci consentono di eseguire determinate azioni prima che l'interfaccia utente venga nuovamente renderizzata, evitando in tal modo il rendering non necessario che potrebbe mostrare uno stato incoerente dell'applicazione. Ma andiamo con ordine: un'attività del ciclo di eventi rappresenta un'azione eseguita dal browser. I compiti sono raggruppati in due categorie:

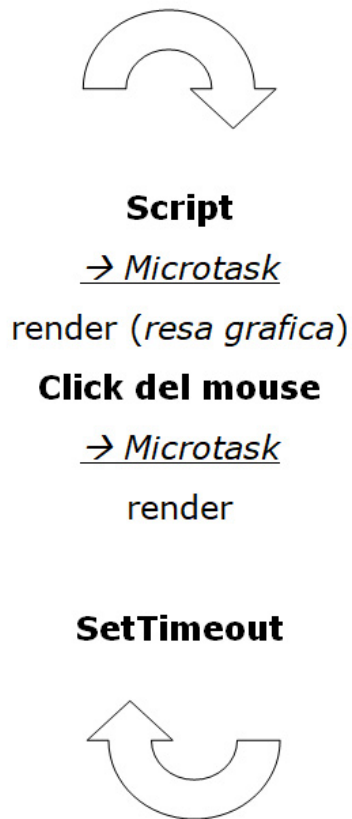
I *macrotask* sono azioni del browser autonome, come la creazione dell'oggetto documento principale, la gestione di vari eventi e la creazione di URL e i suoi cambiamenti.

Le *microtask* sono attività più piccole che dovrebbero essere eseguite il prima possibile.

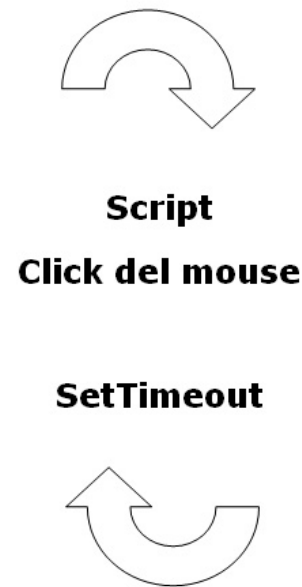
Gli esempi includono callback, promesse e modifiche al DOM.

A causa del modello di esecuzione a thread singolo, le attività vengono elaborate una alla volta e, dopo che un'attività inizia l'esecuzione, non può essere interrotta da un'altra attività. Il ciclo di eventi di solito ha almeno due code di eventi: una coda macrotask e una coda di microtask:

Loop di eventi con Micro e Macrotask



Loop di eventi di una Macrotask



I *timer* offrono la possibilità di ritardare in modo asincrono l'esecuzione di una parte di codice a partire da un certo numero di millisecondi e anche per suddividere un codice troppo complesso e pesante in blocchi che non ostruiranno il browser.

La funzione *setTimeout* si usa per eseguire un callback dopo che il valore specificato è trascorso.

Con *setInterval* potremo avviare un timer che tenterà di eseguire la richiamata all'intervallo di ritardo specificato, fino all'annullamento.

Entrambe le funzioni restituiscono un ID che possiamo utilizzare per annullare un timer tramite le funzioni *clearTimeout* e *clearInterval*.

Adesso diamo un'occhiata ad alcune piccole puntualizzazioni, alla modalità rigorosa, al BOM, al DOM e alla sua interazione con il codice Javascript.

Differenze di grammatica e punteggiatura

I linguaggi di programmazione orientati agli oggetti come Python, Java e Javascript sono simili ma anche molto diversi tra loro. Ora vedremo come per ottenere lo stesso risultato ad esempio JavaScript ha delle convenzioni sulla punteggiatura e sulla spaziatura differenti rispetto agli altri, oltre alla grammatica generale ovviamente:

```
function prova(z) {(Javascript)
return ++z
}
```

```
int pro_va(int z) {(Java)
z += 1;
return z;
}
```

```
def pro_va(z):(Python)
z += 1
return z
```

Quindi per Javascript, come visto, avremo zero spazi tra il nome della funzione e le parentesi, il nome della funzione non necessita del simbolo underscore (_) e anche la punteggiatura è diversa. Tuttavia le differenze non solo solamente queste, tra le altre, banalmente è rimarcabile la difformità strutturale rispetto a Java, è bene ricordarlo sempre.

Strict Mode (modalità rigorosa)

Nel corso del nostro studio, qualche volta, dovremo ricorrere a questa modalità. Si tratta dell'utilizzo di una variante limitata del codice JavaScript. Ci servirà in alcune fasi dello sviluppo soprattutto per eliminare alcuni bug e errori altrimenti difficili da individuare *lanciando (throw)* delle eccezioni e ottimizzare determinati processi.

La si può utilizzare per tutto lo script o anche solo in una funzione. Vediamo un banale esempio dove evidenziamo un problema nascosto:

```
"use strict"
let miaProva = 06
```

Perché viene riportato un errore? E come mai se eseguo il codice senza *strict mode* va tutto liscio? Semplice, in questo caso abbiamo scoperto che l'uso di **numeri ottali** non è consentito in Javascript, in quanto potrebbero essere fonte di confusione nella stesura del codice.

L'esempio che abbiamo appena visto ci mostra come attivare la modalità rigorosa ("use strict").

In determinati casi ci servirà per scrivere codice più sicuro e per semplificare l'uso di alcune variabili.

BOM e DOM

Il modello BOM, ossia *Browser Object Model*, permette l'accesso e la possibilità di editare la finestra (Window) del browser. Lo sviluppatore potrà modificare il testo della barra di stato della finestra, muoverla, e agire su diversi parametri della stessa. Non esiste uno standard come per il DOM, quindi ogni browser ne ha uno suo.

Più importante ed incisivo per noi è il DOM (Document Object Model), il quale definisce uno standard per l'accesso ai documenti. Il modello DOM è una piattaforma e un'interfaccia indipendente dal linguaggio che consente a programmi e script di accedere dinamicamente e aggiornare il contenuto, la struttura e lo stile di un documento.

Lo standard W3C (*World Wide Web Consortium*) del DOM è suddiviso in 3 parti differenti:

Core DOM - modello standard per tutti i tipi di documenti

XML DOM - modello standard per documenti XML

HTML DOM - modello standard per documenti HTML

A noi interessa l'HTML DOM, il quale è un modello di oggetti standard e un'interfaccia di programmazione che definisce gli elementi HTML come oggetti, le loro proprietà, i metodi per accedere a tutti gli elementi e i loro eventi.

Ne consegue che con questa piattaforma potremo ottenere, modificare, aggiungere o eliminare elementi HTML i quali sono organizzati all'interno del documento con una struttura ad albero.

I metodi HTML DOM sono azioni che possiamo eseguire (sugli elementi), mentre le proprietà HTML DOM sono valori (di elementi) che puoi impostare o modificare.

È possibile accedere al DOM HTML con JavaScript (e con altri linguaggi di programmazione), qui tutti gli elementi sono definiti come oggetti e l'interfaccia di programmazione è costituita dalle proprietà e dai metodi di

ogni oggetto. Una proprietà è un valore che puoi ottenere o impostare (come modificare il contenuto di un elemento HTML).

Un metodo è un'azione che puoi eseguire (come aggiungere o eliminare un elemento HTML), qui sotto vediamo la modifica del contenuto (*innerHTML*) <div> dell'elemento con id="prova":

```
<html>
<body>
  <div id="prova"></div>
  <script>
    document.getElementById ("prova").innerHTML = "La nostra prima
stringa";
  </script>
</body>
</html>
```

Notiamo che *getElementById* è un metodo, mentre *innerHTML* è una proprietà, con il primo avremo il modo più comune per accedere a un elemento HTML, tramite l'elemento id, come nell'esempio sopra, dove abbiamo utilizzato il metodo *getElementById* per trovare l'elemento *id="prova"*.

La proprietà innerHTML

Il modo più semplice per ottenere il contenuto di un elemento è utilizzare la proprietà *innerHTML*, essa è utile per ottenere o sostituire e modificare il contenuto degli elementi HTML, inclusi `<html>` e `<body>` (in questo caso non sarà possibile sostituirli ovviamente). Vediamo:

```
document.getElementById(id).innerHTML = new HTML
```

Questo esempio cambia il contenuto dell'elemento `<div>`:

```
<html>
<body>
<div id="prova2">Ecco il nostro esempio</div>
<script>
document.getElementById("prova2").innerHTML = "Nuovo testo";
</script>
</body>
</html>
```

Il documento HTML sopra contiene un elemento `<p>` con *id*="prova2", usiamo il DOM HTML per ottenere l'elemento con *id*="prova2" e poi JavaScript cambia il contenuto (*innerHTML*) di quell'elemento in "Nuovo testo". Quest'altro esempio cambia il contenuto dell'elemento `<div>`:

```
<!DOCTYPE html>
<html>
<body>
<div id="pr1">Piè pagina</div>
<script>
let element = document.getElementById("pr1");
element.innerHTML = "Nuovo piè pagina";
</script>
</body>
</html>
```

Il documento HTML sopra contiene un elemento `<div>` con *id*="pr1", usiamo il DOM HTML per ottenere l'elemento con *id*="pr1", dopodiché JavaScript cambia il contenuto (*innerHTML*) di quell'elemento in "Nuovo piè pagina".

Modifica dello stile HTML

Il DOM HTML consente a JavaScript di modificare lo stile degli elementi HTML. Per modificare lo stile di un elemento HTML, utilizza questa

sintassi:

```
document.getElementById(id).style.property = new style
```

L'esempio seguente cambia lo stile di un elemento *<div>*:

```
<html>
<body>
<div id="pr5">Buongiorno</div>
<script>
document.getElementById("pr5").style.color = "blue";
</script>
<div>Colore cambiato!</div>
</body>
</html>
```

Utilizzo di eventi

Il DOM HTML consente di eseguire codice quando si verifica un evento. Essi vengono generati dal browser quando "qualcosa" accade agli elementi HTML, ad esempio quando si fa clic su un elemento, la pagina è stata caricata, i campi di input vengono modificati.

Questo esempio cambia lo stile dell'elemento HTML con *id="id9"*, quando l'utente fa clic su un pulsante:

```
<!DOCTYPE html>
<html>
<body>
<div id="id9">Intestazione</div>
<button type="button"
onclick="document.getElementById('id9').style.color = 'green'">
Cliccami!</button>
</body>
</html>
```

Creazione di nuovi elementi HTML (nodi)

Per aggiungere un nuovo elemento al DOM HTML, è necessario prima creare l'elemento (nodo dell'elemento), quindi aggiungerlo a un elemento esistente:

```
<div id="d5">
  <p id="a1">Paragrafo</p>
  <p id="a2">Un altro paragrafo</p>
</div>
<script>
let pfo = document.createElement("p");
```

```
let nodo = document.createTextNode("Ecco quello nuovo");
pfo.appendChild(nodo);
let element = document.getElementById("d5");
element.appendChild(pfo);
</script>
```

Questo codice crea un nuovo elemento `<p>`:

```
let pfo = document.createElement("p");
```

Per aggiungere testo all'elemento `<p>`, bisogna prima creare un nodo di testo, dopo lo aggiungeremo all'elemento `<p>`:

```
let nodo = document.createTextNode("Ecco quello nuovo");
pfo.appendChild(nodo);
```

Infine è necessario aggiungere il nuovo elemento a un elemento esistente, prima cercandolo tramite l'id e poi aggiungendolo:

```
let element = document.getElementById("d5");
element.appendChild(pfo);
```

Rimozione di elementi HTML esistenti

Per rimuovere un elemento HTML, utilizzeremo il metodo `remove()`:

```
<div>
  <p id="z2">Paragrafo 10</p>
  <p id="z3">Paragrafo 30</p>
</div>
<script>
let elmnt = document.getElementById("z2");
elmnt.remove();
</script>
```

Il documento HTML contiene un elemento `<div>` con due nodi figlio (due elementi `<p>`):

```
<div>
  <p id="z2">Paragrafo 10</p>
  <p id="z3">Paragrafo 30</p>
</div>
```

Trova l'elemento che desideri rimuovere e poi lo eliminiamo con il metodo `remove()`:


```
let elmnt = document.getElementById("z2");  
elmnt.remove();
```

Dopo aver visto velocemente alcune peculiarità del nostro linguaggio cerchiamo di andare più in profondità con uno dei pilastri di Javascript: ossia le funzioni e il loro utilizzo.

Quiz & esercizi

- 1) Creare un diagramma di flusso con i nodi del dom e scriverci un listato con almeno 10 nodi.
- 2) Utilizzare il metodo *remove()* per ridurre i nodi del DOM a 8.
- 3) Aggiungere il nodo figlio “Mario”.
- 4) Rinominare il titolo della pagina in “esercizio numero 3”
- 5) È possibile usare la modalità rigorosa nel codice HTML?
- 6) Modifichiamo questo codice in modo che la proprietà *.innerHTML* cambi l'id in “prova studente”:

```
<html>  
<body>  
<div id="prova">Prova</div>  
<script>  
alert("prova");  
</script>  
</body>  
</html>
```

- 7) Se volessi scrivere Mario su una finestra che istruzione dovrei usare?
- 8) Utilizzare *console.log* per scrivere, con delle istruzioni Javascript, Mario sulla console di Chrome (Shift+CTRL+J per attivare).
- 9) Crea una pagina HTML con un pulsante. Utilizza JavaScript per assegnare due eventi al pulsante: uno per mostrare un messaggio quando viene premuto e un altro per cambiare il colore del pulsante quando si passa il mouse sopra.
- 10) Crea una pagina con un elemento padre e un elemento figlio. Assegna eventi a entrambi e verifica l'ordine in cui vengono scatenati gli eventi quando si fa clic sull'elemento figlio.
- 11) Crea una pagina con un elemento padre e un elemento figlio. Assegna eventi a entrambi e verifica l'ordine in cui vengono propagati gli eventi quando si fa clic sull'elemento figlio.
- 12) Modifica l'esercizio precedente in modo che l'evento sul figlio impedisca la propagazione all'elemento padre.

13) Crea una pagina con un pulsante che aggiunge un evento clicca quando viene premuto e rimuove l'evento quando viene premuto nuovamente.

14) Crea una pagina con un pulsante. Al clic del pulsante, avvia un timer di 2 secondi e visualizza un messaggio dopo il timer. Verifica se l'ordine degli eventi è rispettato.

15)

Riassunto

Abbiamo iniziato ad immergerci nell'oceano di Javascript, trovandoci di fronte ad alcuni concetti basilari come il ciclo di vita (di un'applicazione, di un evento, di una porzione di codice), la coda ed il ciclo degli eventi, micro e macrotask. Tutti questi hanno delle sequenze ben precise che dovremmo imparare bene. L'esplorazione è poi proseguita con alcune nozioni sul DOM ed il codice HTML.

Tutte queste nozioni possiamo vederle come una sorta di infarinatura che ci permetterà di immergerci in Javascript pronti a comprenderne tutte le sfaccettature.

3 - Stringhe e cicli

Maiuscolo e minuscolo
Regole sulla grammatica
Proprietà, metodi e funzioni utilizzabili
Ripasso dei loop

Quando girovaghiamo in internet con l'intenzione di fare shopping, troveremo prima o poi la pagina che ci interessa. In quel caso, dopo aver scelto il prodotto completeremo l'acquisto. Per farlo dovremo inserire dei dati, sicuramente la prima volta il nostro indirizzo, nome, cognome, etc. Queste informazioni saranno processate e quindi estrapolate dal form che avremo compilato e lo strso dicasi per l'ordine eseguito. Tutte cose che noi e altri milioni di persone eseguono giornalmente. Ciò che ci interessa sono proprio le stringhe dei dati inseriti. Queste dovranno essere lavorate dal nostro codice, completamente o solo parzialmente, ad esempio se compro 10 bottiglie di spumante, nel mio codice in un determinato momento avrò bisogno di sapere proprio la quantità. Di contro, del mio indirizzo prenderò l'intera stringa. Altri dati dovranno essere sommati o sottratti, come le quantità o il prezzo e così via.

Per tutti questi motivi daremo una breve occhiata alle regole base che sono dietro le stringhe, alle proprietà e ai metodi applicabili.

Maiuscolo e minuscolo

In Javascript il testo è sensibile alle lettere maiuscole o minuscole (case sensitive), quindi "Prova" e "prova" sono 2 parole totalmente diverse e nei listati non vanno assolutamente confuse.

```
let prova, Prova;
```

Le due variabili si assomigliano, ma, come detto, non sono uguali:

```
let prova, Prova;  
prova = 10;  
const Prova = 50;  
console.log(Prova, prova, prova + Prova);
```

Difatti abbiamo appena visto che *prova* e *Prova* avremmo forse potuto chiamarli *bianco* e *nero* oppure *valore1* e *valore2* per evitare confusione.

Quindi, ad esempio, se dichiariamo una variabile denominata *coloreSfondo*, per vederla non potremo usare *ColoreSfondo*, *coloresfondo* o altre combinazioni.

In JavaScript, la distinzione tra maiuscole e minuscole non si applica semplicemente ai nomi delle variabili, ma in aggiunta alle parole chiave, ai gestori di eventi e alle proprietà o ai modi degli oggetti. Le parole chiave, ad esempio, saranno tutte minuscole: *for*, *if*, *let*, *const* e così via. D'altra parte, per le proprietà si utilizza la convenzione cosiddetta "*a dorso di cammello*", ossia scritta in questo modo: *miaProva()*, *listaOggetti()*, *proVa()*. Teniamolo bene a mente quando andremo a provare i nostri primi listati.

Regole generali

La prima cosa da sapere è che le stringhe non possono essere mai modificate, al massimo sostituite.

Quando scrivete una stringa non vi dimenticate le virgolette, altrimenti avremo delle noie:

```
proVa=Mario;
```

L'errore viene evidenziato perché *Mario* non è tra virgolette (quindi non è una stringa) e il motore di Javascript si metterà a cercare un'altra variabile che si chiama *Mario*. Ricordiamo sempre che quando creiamo delle stringhe dobbiamo seguire delle semplici regole:

- tutto il contenuto della stringa deve essere compreso tra le virgolette di apertura e quelle di chiusura, vale sia `“”` sia `“` sia `”` sia ```
- non bisogna andare a capo tra l'apertura e la chiusura, a meno che non usiamo i *backticks* (```)
- non si possono aprire altre virgolette all'interno della stringa, salvo che siano diverse da quelle di apertura (`proVa=“testo‘altro’”`)
- possiamo assegnare a una variabile il contenuto di un'altra variabile:

```
proVa="Mario";  
miChiamo=proVa;
```

```
alert(proVa);
```

Non ci sarà alcun errore, perché *proVa* è una variabile che abbiamo creato noi nel codice.

Le virgolette, o apici, le conosciamo tutti e in Javascript hanno la stessa valenza, invece i backticks sono sicuramente meno famosi, ma non meno utili. Difatti li possiamo utilizzare per includere un'espressione all'interno di una stringa tramite `${}` come possiamo vedere:

```
function moltiplicazione(x, y) {  
    return x * y;  
}  
alert(`5 * 2 = ${moltiplicazione(5, 2)}.`);
```

E ci permettono anche di dividere la stringa su diverse righe:

```
let listaDolci = `Dolci:  
Cioccolato  
Torta  
Bigné  
`;  
alert(listaDolci);
```

Esistono anche altri modi di farlo, potrebbe essere un buon esercizio (magari più avanti nello studio) scriverci un listato.

Prima di proseguire oltre diamo un'occhiata ad un esempio proposto all'inizio di questo libro:

```
function esecuzione()  
{  
    aggiunta('Devo studiare!');  
}  
function aggiunta(altro)  
{  
    alert('Questo è l'argomento ricevuto da esecuzione(): ' +  
        altro);  
}  
esecuzione();
```

I più attenti avranno notato il simbolo `\` all'interno di *alert()*. Sveliamo l'arcano, quel simbolo serve ad impedire che l'apice `'` concluda la stringa. Si può anche ovviare impostando in questo modo:

```
alert("Questo è l'argomento ricevuto da... ")
```

Proprietà metodi e funzioni

In questo capitolo daremo una breve occhiata ad alcune cose che sicuramente ci saranno utili nella stesura di qualche listato, mischiate con altre che dobbiamo ancora studiare. Tuttavia prima di iniziare ad imparare le funzioni e tutto il resto è molto importante avere queste basi che ci aiuteranno a proseguire meglio il cammino verso la comprensione di Javascript.

Iniziamo con la proprietà *.length* ci permetterà di conoscere la lunghezza della stringa:

```
alert("mario".length); // il risultato è 5
```

Per estrapolare un carattere da una stringa si utilizzano le parentesi quadre [*numero*] dove all'interno avremo un numero corrispondente alla posizione del carattere che ci interessa, partendo da zero. In alternativa possiamo utilizzare la chiamata al metodo *.charAt(numero)* come da esempio di seguito:

```
let frase = "Primavera";  
alert(frase[5]); // v  
alert(frase.charAt(0)); // P  
alert(frase[frase.length - 1]); // a
```

I metodi *toLowerCase()* e *toUpperCase()* ci permettono di cambiare da minuscolo a maiuscolo o viceversa uno o più caratteri della nostra stringa:

```
alert('Mario'.toUpperCase()); // MARIO  
alert('Mario'.toLowerCase()); // mario  
alert('Mario'[0].toLowerCase()); // m
```

Con la funzione *.substr* possiamo estrapolare la stringa selezionata, anche parzialmente.

```
let stringa = "Che bella giornata";  
alert(stringa.substr(1,10));
```

il risultato sarà *“he bella g”*. La funzione *.substr*, come indicato nelle parentesi tonde, ha estratto la seconda lettera e le seguenti dieci, da sinistra a destra, spazi compresi. Ricordiamo che il conteggio delle lettere parte da 0. La funzione *.substring* è molto simile, vediamo l'esempio con il medesimo codice:

```
let stringa = "Che bella giornata";  
alert(stringa.substring(1,10));
```

il risultato sarà *“he bella”*, anche in questo caso andremo a estrarre le lettere nelle rispettive posizioni, dove la posizione 0 è la *C* e così via. Il conteggio in questo caso terminerà prima dell’ultima lettera, nel nostro listato la *g*, cioè alla posizione 9: lo spazio tra *a* e *g*, il quale viene contato.

E’ possibile usare l’operatore negativo (-), come esercizio fate alcune prove. Con il metodo *.indexOf* andremo a trovare, perdonatemi il gioco di parole, una stringa all’interno di una stringa, ottenendo in caso positivo un numero corrispondente alla posizione, -1 se non viene trovato nulla. Vediamo come:

```
let si = 'La pizza con l'ananas fa SCHIFO';  
alert(si.indexOf('SCHIFO') ); // 25  
alert(si.indexOf('schifo') ); // -1  
alert(si.indexOf("con") ); // 9  
alert(si.indexOf("a", 6) ); // 7
```

Il secondo ed il quarto risultato potrebbero sembrare strani, in realtà dobbiamo ricordarci che la ricerca tiene conto di maiuscole e minuscole, e infine possiamo far partire la ricerca da una posizione più avanzata ("a", 6 cioè la lettera *a* partendo dalla sesta posizione).

Con *.includes* verificheremo la corrispondenza di una data stringa, senza il bisogno di doverne necessariamente conoscerne la posizione, il risultato ottenuto sarà *true/false*:

```
alert("Che bella giornata".includes("bella")); // true  
alert("Che bella giornata".includes("brutta")); // false  
alert("Che bella giornata".includes("bella", 6)); // false
```

Molto simili i metodi *.startsWith* e *.endsWith*:

```
alert("Mario".startsWith("Ma") ); // true  
alert("Mario".endsWith("io") ); // true
```

Abbiamo visto all’inizio del capitolo che con le parentesi potevamo estrapolare una lettera, bene, tramite il metodo *.slice* faremo lo stesso con una stringa:

```
let stringa = "Smartphone";  
alert(stringa.slice(0, 3) ); // 'Sma'  
alert(stringa.slice(-5, -1) ); // 'phon'
```

Nell'ultima riga inserendo numeri negativi il conteggio partirà da destra anzichè da sinistra.

I cicli o iterazioni

In qualunque linguaggio, a partire dal Basic fino al C, avremo bisogno di utilizzare i loop, cioè i cicli, tecnicamente detti iterazioni. Javascript non è da meno e ci offre i tre classici: *Il ciclo for()* - *Il ciclo while()* - *Il ciclo do while()*

Il primo serve a ripetere una o più istruzioni un determinato numero determinato di volte. Per avviare un ciclo *for()* dovremo impostare:

una variabile contatore a cui si assegna un valore di partenza, la condizione di conclusione del ciclo, la quale restituirà un valore booleano (cioè vero o falso), quindi un valore della variabile contatore assegnata in precedenza che, una volta raggiunto, termina il loop, in caso contrario esso continuerà all'infinito di fatto bloccando l'esecuzione del programma e del browser. Infine avremo il cosiddetto step, cioè l'incremento, o decremento, da applicare alla variabile contatore ad ogni passaggio del ciclo:

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

La prima parte del ciclo for è `let i = 0;`, dove inizializziamo una variabile *i* con il valore 0. Questa verrà utilizzata come un contatore per tenere traccia delle iterazioni. La seconda parte del ciclo for è `i < 5;`. Finché questa condizione è vera (true), il ciclo continuerà ad eseguire il suo blocco di istruzioni. In questo caso, il ciclo si fermerà quando *i* diventerà uguale o superiore a 5. La terza parte del ciclo for è `i++`, cioè l'incremento. Dopo ogni iterazione del ciclo, il valore di *i* aumenterà di uno. Ecco cosa accade durante ogni iterazione: all'inizio, *i* è 0. La condizione `i < 5` è vera, quindi il ciclo entra nel suo blocco di istruzioni. Il `console.log` stampa "Iterazione 0". Dopo questa istruzione, *i* viene incrementato di uno diventando 1. Ora, *i* è 1. La condizione `i < 5` è ancora vera, quindi il ciclo esegue il `console.log` che stampa "Iterazione 1" e incrementa nuovamente *i*. Questo processo si ripete fino a quando *i* diventa 5, quindi la condizione `i < 5` non è più vera, il che significa che il ciclo termina e ovviamente non viene eseguita un'altra iterazione. Il risultato sarà la stampa di "Iterazione 0", "Iterazione 1",

"Iterazione 2", "Iterazione 3" e "Iterazione 4" sulla console.
Vediamo lo stesso esempio con un decremento:

```
for (let i = 4; i >= 0; i--) {  
  console.log('Iterazione ' + i);  
}
```

A differenza di *for()*, il ciclo *while()* non prevede l'utilizzo di alcun contatore ma semplicemente l'indicazione di una condizione di conclusione, quindi l'iterazione cesserà al verificarsi della condizione data:

```
let numero = 1;  
while (numero < 5) {  
  console.log('Iterazione ' + numero);  
  numero++;  
}
```

Il ciclo *while* verifica la condizione prima di ogni iterazione, quindi se la condizione è falsa all'inizio, il blocco di istruzioni non verrà mai eseguito.

L'esempio appena visto lo possiamo rimodulare per *for*:

```
for (let numero = 1; numero < 5; numero++) {  
  console.log('Iterazione ' + numero);  
}
```

Quindi in *while()*, come in *for()*, il ciclo è determinato da una variabile numerica, ma potremo anche utilizzare un valore booleano.

La struttura *do while()* è una variante di *while()*. La differenza sostanzialmente consiste nel fatto che il controllo della condizione avviene dopo l'esecuzione dell'istruzione e non all'inizio. Inoltre il codice all'interno del blocco *do* verrà eseguito *almeno una volta* prima di verificare la condizione nel *while*. Quindi, il ciclo continuerà ad eseguire il blocco fino a quando numero sarà inferiore a 5, proprio come nel ciclo *while* originale:

```
let numero = 1;  
do {  
  console.log('Iterazione ' + numero);  
  numero++;  
} while (numero < 5);
```

Come esercizio provate a creare dei loop utilizzando *while* e *do while*, e anche utilizzando un valore booleano (*true* o *false*) al posto di un numero.

Infine segnaliamo l'istruzione *for await.. of* la quale crea un ciclo su degli oggetti iterabili asincroni (vedi il capitolo 8), compresi array, *map* e *set*.

La sintassi è: `for await (const elemento of oggettoAsincrono) {
dichiarazione}`

vediamo un esempio:

```
async function* a() {  
  let x = 1;  
  while (x < 5) {  
    yield x++;  
  }  
}  
(async() => {  
  for await(const numeri of a()) {  
    console.log(numeri);  
  }  
})(); // 1, 2, 3, 4
```

Forse è ancora troppo presto per comprendere del tutto il listato, lo riguarderemo dopo aver visto il capitolo 8.

Quiz & esercizi

- 1) Creare un listato che cerchi in una stringa la parola “Prova”
- 2) Creare un listato che estrapoli la sotto stringa “Prova” per inserirla in un altro contesto che abbia un senso (es. “La prova mi ha dato soddisfazione” in “La soddisfazione mi ha dato prova”)
- 3) Creare un programma che restituisca *true* in caso di parola con lettere minuscole e in tal caso avvi un ciclo per modificare le lettere in maiuscolo.
- 4) Modificare solo un paramentro per far si che la stringa Buongiorno si ripeta più volte:

```
for (let x = 1; x < 2; x++) {  
  console.log('Buongiorno: ' + x);  
}
```

- 5) Scrivi un programma che stampi la lunghezza di una stringa.
- 6) Scrivi un programma che inverta una stringa.
- 7) Scrivi un programma che conti quante volte appare una lettera specifica in una stringa.
- 8) Scrivi un programma che verifichi se una stringa è un palindromo (legge lo stesso al contrario).
- 9) Scrivi un programma che rimuova gli spazi da una stringa.
- 10) Scrivi un programma che converte una stringa in maiuscolo.
- 11) Scrivi un programma che trovi la posizione di una sottostringa all'interno

di una stringa.

12) Scrivi un programma che rimuova una sottostringa da una stringa.

13) Scrivi un programma che conta il numero di vocali in una stringa.

14) Scrivi un programma che generi una stringa casuale di lunghezza specifica.

15) Scrivi un programma che conta il numero di parole in una stringa.

16) Scrivi un programma che calcoli la somma dei primi N numeri interi positivi. L'utente dovrebbe inserire il valore di N e il programma dovrebbe restituire la somma.

Riassunto

Il breve ripasso che abbiamo affrontato in questo capitolo servirà a tutti quelli che sono alle prime armi, in questo caso suggerisco di approfondire ulteriormente questi temi in quanto proseguendo ci serviranno delle basi solide per proseguire nello studio. Difatti il discorso verrà ripreso in vari capitoli e soprattutto per quanto riguarda le stringhe quando andremo a trattare il temibile argomento delle espressioni regolari. Adesso, prima di tuffarci a pesce verso uno degli argomenti cardine di Javascript (le funzioni), passeremo a un'infarinatura sugli oggetti (software).

4 - Antipasto con oggetti e dati

Diversi tipi di dati
Gli oggetti
Chiavi, valori e operatori
L'oggetto Math

Nel capitolo precedente, parlando delle stringhe non abbiamo minimamente accennato al fatto che, nel codice Javascript, oltre ad essere banalmente una porzione di testo, un nome, o anche un numero, sono principalmente viste come *oggetti software*. Detto così potrebbe non significare nulla, tuttavia è un argomento molto importante, perciò in questo capitolo avremo un antipasto che ci chiarirà alcuni aspetti utili al nostro studio. Approfondiremo poi esaustivamente l'argomento nei capitoli nove, dieci e undici.

In Javascript, come abbiamo avuto modo di vedere anche precedentemente, ci sono nove tipi di dati utili per operare con le variabili. Forse non ce ne siamo neanche accorti, ma già nelle prime prove effettuate sul codice sicuramente di dato ne abbiamo manipolato qualcuno. Ma andiamo a definirli tutti:

- *String*: le stringhe possono essere composte da diversi caratteri.
- *Number*: numeri, interi o in virgola mobile.
- *Bigint*: viene utilizzato per definire interi di lunghezza arbitraria.
- *Boolean*: può essere *true* o *false*.
- *Symbol*: per identificatori unici.
- *Null*: un valore sconosciuto.
- *NaN*: un numero indefinito, non delineabile (es 0/0).
- *Undefined*: un valore non assegnato.
- *Object*: per strutture dati più complesse.

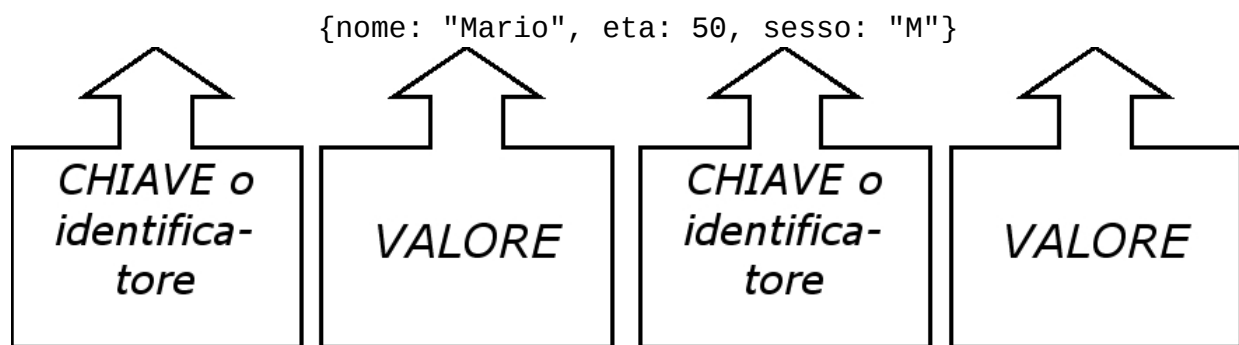
I primi otto sono chiamati “*primitivi*”, perché i loro valori contengono al massimo un singolo elemento come un numero, una stringa, un valore sconosciuto (*null*) e così via.

Gli oggetti, invece, vengono utilizzati per catalogare vari tipi di dati ed altri elementi più complessi. In Javascript, come detto più volte, sono uno dei cardini del linguaggio stesso e dovremo comprenderli bene prima di studiare altri argomenti.

Possiamo creare un oggetto con le parentesi graffe {...}, assieme a una lista di proprietà composte da stringhe di chiavi e valori, ma può anche non contenere alcuna proprietà. Questo tipo di impostazione accomuna l'oggetto in questione ad un archivio organizzato, con un nome del file principale, seguito da varie chiavi e valori che ne identificano il contenuto e che potranno essere poi consultati. Vediamo degli esempi per dichiararli:

```
let esempio = {nome: "Mario", eta: 50, sesso: "M"}
let esempio = {};
let esempio = new Object();
```

Ovviamente consideriamoli come tre righe separate, non un unico codice. L'oggetto dichiarato è *esempio*, gli ultimi due sono senza proprietà, l'ultimo in particolare è stato evocato dal costruttore *new Object* (vedremo più avanti i costruttori). I primi due sono oggetti letterali, e il primo ha delle chiavi e dei valori assegnati:



In questo caso abbiamo delle chiavi (o identificatori) prima dei due punti: *nome*, *eta*, *sesso* e dei valori: *"Mario"*, *50*, *"M"*. Anche qui valgono le stesse regole sulle virgolette, i numeri e le parole composte viste per le stringhe.

L'oggetto *esempio* quindi ha archiviato tre *files* che potranno essere richiamati, aggiunti o rimossi all'occorrenza. I valori delle proprietà potranno essere di qualunque tipo (numeri, simboli, booleani) e sono accessibili utilizzando *oggetto.identificatore*:

```
let esempio = {nome: "Mario", eta: 50, s: "M", ["n cliente"]: 5}
console.log(esempio.nome);
console.log(esempio.eta);
console.log(esempio.s);
console.log(esempio["n cliente"]);
```

In questi casi l'identificatore non potrà essere una parola composta, come “*n cliente*”, perciò dovremo utilizzare le parentesi quadre `["n cliente"]`.

Per rimuovere una proprietà, utilizzeremo l'operatore *delete*:

```
delete esempio["n cliente"];
```

Non ci serviranno solo a questo, difatti, le parentesi quadre, differentemente dalle stringhe, ci permetteranno di passare il nome della proprietà come risultato di un'espressione come, ad esempio, una variabile:

```
let esempio = {nome: "Mario", eta: 50, sesso: "M"};
let chiave = prompt("Cosa ci interessa di Mario?", "eta");
alert(esempio[chiave]);
```

Possiamo servirci anche di espressioni più complesse da inserire nelle parentesi quadre:

```
let cartamoneta = 'Euro';
let contenuto = {[cartamoneta + 'Spicci']: 100};
alert(contenuto.EuroSpicci) ;
```

È bene ricordare che una variabile non può avere il nome uguale ad una parola chiave come “for”, “var”, “delete”, e così via, tuttavia per le proprietà degli oggetti non esistono le stesse restrizioni, quindi oltre a *nome: Mario*, potremo anche assegnare il valore 10 a *return*, o anche *var: "Giovanni"*.

Sappiamo che possiamo accedere a una qualsiasi proprietà degli oggetti, anche se non è stata definita. L'operatore *in* ci darà una mano in questo:

```
let esempio = {nome: "Mario", eta: 50, sesso: "M"};
console.log("eta" in esempio);
console.log("altezza" in esempio);
```

Se il nome di una proprietà, solitamente una stringa, non esiste verrà riportato *false*. Questo operatore è molto utile anche utilizzando un loop differente da quelli già conosciuti: il ciclo *for.. in*, che useremo per attraversare tutte le chiavi di un oggetto. Proviamo vedendo come mostrare tutte le proprietà di *esempio*:

```
let esempio = {nome: "Mario", eta: 50, sesso: "M"};
  for (let key in esempio) {
    console.log(key);
    console.log(esempio[key]);
  }
```

È stato utilizzato *for* assieme a *let* e alla variabile *key* ottenendo un risultato interessante. Prendiamo nota che anche tutte le altre variabili potranno essere utilizzate allo stesso modo, per esempio: "*for(let prop in obj)*".

Vedremo nei prossimi capitoli che ovviamente ci sono altri tipi di oggetti oltre a quello di tipo “semplice” trattato finora.

Verifica del tipo di dati - Operatore typeof

L'operatore *typeof* è uno strumento semplice ma efficace per verificare il tipo di dati di una variabile. Restituisce una stringa che rappresenta il tipo di dati.

```
let numero = 42;
let testo = "Ciao, mondo!";
let abilitato = true;

console.log(typeof numero); // "number"
console.log(typeof testo);  // "string"
console.log(typeof abilitato); // "boolean"
```

Tuttavia, *typeof* ha alcune limitazioni. Ad esempio, restituisce "object" per gli array, anche se sono oggetti, e "object" per null, che potrebbe essere considerato un errore.

Metodo instanceof

Il metodo *instanceof* verifica se un oggetto è un'istanza di una classe o di un costruttore specifico. È particolarmente utile per verificare se un oggetto è un'istanza di un array o di altri oggetti personalizzati.

```
let numeri = [1, 2, 3, 4, 5];
let persona = { nome: "Alice", età: 30 };

console.log(numeri instanceof Array); // true
console.log(persona instanceof Object); // true
```

Funzione isNaN

La funzione *isNaN* verifica se un valore è NaN (*Not-a-Number*), che, come già detto è un valore speciale in JavaScript che indica un risultato non valido in operazioni aritmetiche.

```
let risultato = 10 / "Ciao"; // NaN
console.log(isNaN(risultato)); // true
```

Conversione esplicita

È possibile utilizzare le funzioni di conversione esplicita, come *parseInt* e *parseFloat*, per convertire una stringa in un numero o per estrarre un numero da una stringa.

```
let numeroDaStringa = parseInt("42");
console.log(numeroDaStringa); // 42
```

Verifica di valori Falsy e Truthy

In JavaScript, alcuni valori sono considerati "falsy" (valore false) e altri "truthy" (valore true) quando valutati in un contesto booleano. Ad esempio, 0, "", null, undefined, NaN, e false sono valori falsy. Tutti gli altri valori sono truthy.

```
let valore = 0;

if (valore) {
  console.log("Valore truthy");
} else {
  console.log("Valore falsy");
}
```

La verifica del tipo di dati è fondamentale per scrivere codice JavaScript robusto e privo di errori. Conoscere le diverse tecniche a disposizione per la verifica dei tipi ti consentirà di scrivere codice più affidabile e di evitare errori comuni. Utilizza le diverse tecniche di verifica del tipo di dati in base alle tue esigenze specifiche per garantire che il tuo codice funzioni correttamente in tutte le situazioni.

Conversione del tipo di dato

Oltre alla verifica del tipo di dati, è importante comprendere come avvenga la conversione dei tipi. JavaScript esegue automaticamente la conversione in determinate situazioni per gestire operazioni tra tipi diversi. Questo processo può talvolta comportare risultati inaspettati, quindi è essenziale conoscerlo bene.

Conversione Implicita

JavaScript esegue automaticamente la conversione dei tipi quando si effettuano operazioni tra tipi diversi. Ad esempio, quando si effettua

un'operazione di addizione tra un numero e una stringa, JavaScript converte la stringa in un numero:

```
let numero = 42;  
let testo = "10";  
  
let risultato = numero + testo; // Conversione implicita: "42" +  
"10" = "4210"  
console.log(risultato); // "4210"
```

In questo caso, JavaScript ha convertito la stringa "10" in un numero per eseguire l'operazione di addizione.

Conversione Esplicita

È possibile eseguire conversioni di tipo esplicitamente utilizzando alcune funzioni built-in (integrate):

parseInt e parseFloat

Le funzioni *parseInt* e *parseFloat* consentono di convertire una stringa in un numero intero o decimale, rispettivamente:

```
let numeroDaStringa = parseInt("42");  
console.log(numeroDaStringa); // 42  
  
let numeroDecimaleDaStringa = parseFloat("3.14");  
console.log(numeroDecimaleDaStringa); // 3.14
```

String()

La funzione *String()* converte qualsiasi valore in una stringa:

```
let valore = 42;  
let valoreConvertitoInStringa = String(valore);  
console.log(valoreConvertitoInStringa); // "42"
```

Number()

La funzione *Number()* tenta di convertire un valore in un numero:

```
let testo = "42";  
let testoConvertitoInNumero = Number(testo);  
  
console.log(testoConvertitoInNumero); // 42
```

Conversioni per Operazioni Booleane

Le conversioni dei tipi di dati sono anche comuni quando si lavora con operatori logici. Ad esempio, in una condizione `if`, qualsiasi valore può essere convertito in un valore booleano (*true* o *false*):

```
let valore = "Hello";

if (valore) {
  console.log("Valore true");
} else {
  console.log("Valore false"); // Questa parte verrà eseguita
}
```

In questo caso, la stringa "Hello" viene convertita in *true* poiché è un valore *truthy*.

La conversione dei tipi di dati è una parte fondamentale della programmazione in JavaScript. La comprensione di come avvenga la conversione implicita e l'uso delle conversioni esplicite ti consentirà di gestire con sicurezza i tipi di dati e di evitare errori di conversione inaspettati nel tuo codice. Ricorda di essere consapevole delle conversioni che JavaScript esegue automaticamente per garantire che il tuo codice si comporti come previsto.

Altro sugli oggetti

Vediamo velocemente alcuni argomenti che andremo ad approfondire nei prossimi capitoli:

Costruttori di oggetti: puoi creare oggetti personalizzati utilizzando funzioni costruttrici. Ad esempio:

```
function Persona(nome, età) {
  this.nome = nome;
  this.età = età;
}

const persona1 = new Persona("Alice", 30);
const persona2 = new Persona("Bob", 25);
```

Proprietà dei metodi: puoi aggiungere metodi a un oggetto come se fossero proprietà. Ad esempio:

```
const persona = {
```

```

    nome: "Alice",
    saluta: function () {
        console.log(`Ciao, sono ${this.nome}`);
    },
};

```

```

persona.saluta();

```

Prototype: gli oggetti in JavaScript hanno un prototype che può essere condiviso tra istanze. Puoi estendere le funzionalità di tutti gli oggetti di un certo tipo utilizzando il loro prototype.

```

Persona.prototype.schiarisciEtà = function () {
    this.età += 1;
};

```

```

persona1.schiarisciEtà();

```

Riferimenti agli oggetti: gli oggetti sono passati per riferimento, il che significa che se assegni un oggetto ad una variabile e poi modifichi l'oggetto attraverso un'altra variabile, le modifiche saranno visibili in entrambi i casi.

```

const oggetto1 = { valore: 5 };
const oggetto2 = oggetto1;

```

```

oggetto2.valore = 10;

```

```

console.log(oggetto1.valore); // Stampa 10

```

Clonazione degli oggetti: se vuoi creare una copia indipendente di un oggetto, puoi farlo utilizzando diverse tecniche, ad esempio:

```

Object.assign(): const copia = Object.assign({}, oggettoOriginale);
Operatore di propagazione (...): const copia = { ...oggettoOriginale };
JSON (solo per oggetti semplici): const copia =
JSON.parse(JSON.stringify(oggettoOriginale));

```

Proprietà compute: puoi creare nomi di proprietà dinamicamente utilizzando le parentesi quadre [].

```

const chiave = "nome";
const persona = { [chiave]: "Alice" };

```

```

console.log(persona.nome); // Stampa "Alice"

```

Adesso facciamo una piccola digressione verso la matematica, analizziamo l'oggetto integrato Math.

L'oggetto Math

Math è un oggetto con proprietà e metodi associate a costanti o funzioni matematiche. Ci consente di effettuare diverse operazioni matematiche sui numeri. Non è né un costruttore né un oggetto funzione. La sintassi per le proprietà Math è: *.Math.property*, il linguaggio JavaScript ci fornisce 8 costanti a cui è possibile accedere come proprietà matematiche:

Math.E: la costante di Eulero

Math.LN2: il logaritmo di 2

Math.LN10: il logaritmo di 10

Math.LOG2E: il logaritmo in base 2 di E

Math.LOG10E: il logaritmo in base 10 di E

Math.PI: il PI Greco

Math.SQRT1_2: la radice quadrata di 1/2

Math.SQRT2: la radice quadrata di 2

Per quanto riguarda i metodi, la sintassi è: *Math.method(numero)*. Vediamo come prima cosa come arrotondare un numero in un numero intero, Javascript ci mette a disposizione ben quattro metodi:

Math.ceil(n): n arrotondato per eccesso al numero intero più vicino

Math.floor(n): n arrotondato per difetto al numero intero più vicino

Math.round(n): n arrotondato al suo intero più vicino

Math.trunc(n): la parte intera di n

Esempi:

```
Math.ceil(1.1); // 2
Math.floor(1.9); // 1
Math.round(1.6); // 2
Math.trunc(1.9); // 1
```

Vediamo altri metodi con alcune classiche operazioni, come la radice quadrata e la potenza di un numero:

Math.abs(n): il valore assoluto (positivo) di n
Math.log(n): il logaritmo di n
Math.pow(n, n2): il valore di n alla potenza di n2:
Math.sign(n): con n negativo dà nullo o positivo
Math.sqrt(n): la radice quadrata di n

Esempi:

```
Math.abs(-1.2); // 1.2  
Math.log(10); // 2.30...  
Math.pow(2, 3); // 8  
Math.sign(-20); // -1  
Math.sqrt(25); // 5
```

Con questi metodi potremo calcolare il seno, il coseno oppure prendere dei numeri da una lista in base al loro valore:

Math.sin(n): il seno dell'angolo n espresso in radianti
Math.cos(n): il coseno dell'angolo n espresso in radianti
Math.min(): trova il valore più basso in un elenco
Math.max(): trova il valore più alto in un elenco
Math.random(): un numero casuale compreso tra 0 e 1

Esempi:

```
Math.sin(45); // 0.85...  
Math.cos(45); // 0.52...  
Math.min(1, 7, -22, 10, -3, -2); // -22  
Math.max(1, 7, -22, 10, -3, -2); // 10  
Math.random(); // 0.....
```

Gli altri metodi che non abbiamo visto finora:

acos(n): l'arcoseno di n in radianti
acosh(n): l'arcoseno iperbolico di n
asin(n): l'arcoseno di n in radianti
asinh(n): l'arcoseno iperbolico di n
atan(n): l'arcotangente di n
atan2(n, n2): l'arcotangente del quoziente dei suoi argomenti
atanh(n): l'arcotangente iperbolico di n
cbrt(n): la radice cubica di n
cosh(n): il coseno iperbolico di n
exp(n): il valore di Eulero alla potenza di n

$\sinh(n)$: il seno iperbolico di n
 $\tan(n)$: la tangente di un angolo
 $\tanh(n)$: la tangente iperbolica di n

Quiz & esercizi

- 1) Creare un elenco di 10 persone da 40 anni in su, dopo estrapolare i nomi di quelli che ne hanno 50.
- 2) Creare un listato che calcoli l'ipotenusa di un triangolo rettangolo con lati di 15 e 6 cm.
- 3)

```
let lista = {nome: "Mario", eta: 50, sesso: "M", ["Giorni di vacanza"]: 10, contratto: true};
```

Creare un listato con un ciclo che riporti tutti i valori sulla console.

- 4) Creare un listato in cui vi siano tutte e 4 le operazioni basilari assieme all'oggetto `math` in modo che il risultato sia 50.
- 5) Creare un listato che calcoli l'area di un cerchio.
- 6) Crea una funzione che accetti un argomento e restituisca una stringa che indica il tipo di dati dell'argomento (ad esempio, "stringa", "numero", "booleano", "oggetto", ecc.).
- 7) Crea una funzione che accetti due argomenti e verifichi se hanno lo stesso tipo di dati.
- 8) Crea una funzione che accetti un argomento di tipo stringa che rappresenta un numero e restituisca il valore numerico corrispondente.
- 9) Crea una funzione che accetti due argomenti di tipo stringa e li concateni insieme con uno spazio tra di loro.
- 10) Crea una funzione che accetti una stringa e restituisca il primo carattere della stringa in maiuscolo.

Riassunto

Abbiamo visto che gli oggetti sono come insiemi associativi con diverse caratteristiche speciali: Possono memorizzare proprietà (coppie di chiave-valore) in cui: Il nome della proprietà (chiave) deve essere composta da una o più stringhe o simboli (solitamente stringhe). I valori possono essere di qualsiasi tipo.

Gli oggetti vengono assegnati e copiati per riferimento. In altre parole, la variabile non memorizza il "valore dell'oggetto", ma piuttosto un "riferimento" (indirizzo di memoria). Quindi copiando questa variabile o passandola come argomento ad una funzione, fornirà un riferimento

all'oggetto e non una copia. Tutte le operazioni effettuate su un oggetto copiato per riferimento (come aggiungere/rimuovere proprietà) vengono effettuate sullo stesso oggetto.

5 - *Gli insiemi*

I vettori (o array)

Lavorare con i vettori

Altri insiemi: le mappe e i set

In tutti i linguaggi di programmazione, soprattutto in quelli orientati agli oggetti come Javascript, vi è la necessità di gestire raccolte di dati, o meglio, raccolte di diversi oggetti della stessa tipologia, composte solitamente da variabili. Stiamo parlando degli insiemi, meglio conosciuti come *vettori*, *matrici* o più comunemente *array*, la tipologia più basilare di raccolta di dati in Javascript. Questi vettori o array possiamo vederli come se fossero dei contenitori dove posizioneremo e ordineremo i nostri valori, ai quali potremo accedere nel momento in cui ne avremo bisogno.

Continueremo esplorando alcuni dei metodi utilizzabili con i vettori che ci aiuteranno a scrivere del codice più funzionale e leggibile. Successivamente, studieremo due diversi tipi di insiemi o raccolte con delle peculiarità differenti rispetto agli array: le *mappe* e i *set*. Usando le mappe, puoi creare dizionari contenenti valori, oggetti e funzioni sia come chiave che come valore. *Map* possiede anche diversi metodi rispetto agli array. I *set*, d'altra parte, sono raccolte di elementi unici in cui ogni elemento non può essere ripetuto più di una volta. Adesso cominciamo la nostra lezione con i vettori e le modalità per sfruttarli al meglio, incluse le nuove istruzioni che sono state recentemente standardizzate da ECMAScript a Giugno del 2023.

I vettori (o array)

Il vettore (o array), come già preannunciato, è uno dei tipi di dati più comuni e lo utilizzeremo per gestire raccolte di oggetti. Per definirne uno, essendo l'array un costruttore verrà evocato con la classica sintassi: *new Array*, vediamo:

```
lista=new Array
```



```
lista[0]="Pane";  
lista[1]="Pomodori";  
lista[2]="Pasta";  
console.log(lista);
```

Tramite il costruttore *new Array* abbiamo creato il vettore *lista* assegnandogli tre nomi che abbiamo poi visualizzato nella console.

Le parentesi quadre sono il secondo modo per creare un vettore:

```
const gatti = ["Briciola", "Kitty", "Pongo"];  
  
console.log(gatti);
```

Differentemente da quello che potrebbe pensare un programmatore di un linguaggio come il C, il quale considera i vettori come blocchi sequenziali di memoria che ospitano elementi dello stesso tipo, dove ogni pezzo di memoria è di dimensione fissa e ha un indice attraverso il quale si può accedere facilmente, in JavaScript, questi insiemi hanno una concezione intrinseca distinta: sono principalmente visti come oggetti. Di conseguenza i vettori possono accedere a metodi, come gli altri oggetti, rendendoci così la programmazione anche più facile, andiamo subito a vedere in che modo con un semplice esempio in cui creiamo una banda di gatti e una di cani:

```
const gatti = ["Briciola", "Kitty", "Pongo", "Thor"];  
const cani = new Array ("Fuffi", "Coda");  
  
assert(gatti.length === 4, "Ci sono quattro gatti");  
assert(cani.length === 2, "E due cani")  
assert(gatti[0] === "Briciola", "Briciola è il primo gatto");  
assert(cani[cani.length-1] === "Coda", "Coda è l'ultimo cane")  
assert(gatti[5] === undefined, "Il quinto gatto non c'è");
```

Accediamo agli elementi del vettore con la notazione dell'indice: il primo elemento è indicizzato con 0 e l'ultimo con *array.length - 1*

```
gatti[4] = "Pussy";  
assert(gatti.length === 5, "Il vettore si può ingrandire");
```

Possiamo estendere il nostro vettore inserendo un altro oggetto che segue la sequenza di numerazione di quelli precedenti, partendo da zero.

```
gatti.length = 2;  
assert(gatti.length === 2, "Possiamo sempre modificare la dimensione dell'array");
```

Se riscriviamo manualmente la proprietà *length* con un valore inferiore andiamo ad eliminare automaticamente gli elementi in eccesso.

Come sappiamo, accediamo agli elementi del vettore utilizzando la notazione dell'indice, dove il primo elemento è posizionato all'indice 0 e l'ultimo elemento all'indice *array.length - 1*. Ma se proviamo ad accedere a un indice al di fuori di questi limiti, ad esempio, con il *gatto[4]*, non avremo l'eccezione dell'"*indice di array fuori dai limiti*" che riceviamo nella maggior parte degli altri linguaggi di programmazione. Invece, indefinito è restituito, segnalando che non c'è niente.

Questo comportamento è una conseguenza del fatto che gli array JavaScript sono oggetti. Proprio come diventeremmo indefiniti se provassimo ad accedere a una proprietà dell'oggetto inesistente, otteniamo indefinito quando si accede a un indice di array inesistente.

D'altra parte, se proviamo a scrivere in una posizione al di fuori dei limiti dell'array, come visto precedentemente con: *gatto[4] = "Pussy"* l'array si espanderà per adattarsi alla nuova situazione, ma la posizione numero tre sarà considerata vuota, e, se richiamata, restituirà *undefined*.

A differenza della maggior parte degli altri linguaggi, in JavaScript, gli array mostrano anche una caratteristica peculiare relativa alla proprietà *length*: niente ci impedisce di modificare manualmente il suo valore. L'impostazione di un valore superiore alla lunghezza corrente espanderà l'array con elementi *undefined*, mentre l'impostazione con un valore inferiore taglierà l'array, come in *gatti.length = 2 ;*.

Oltre a questo avremo sicuramente notato che per creare vettori l'uso di valori letterali di matrice, ossia le parentesi quadre, è preferibile rispetto alla creazione tramite il costruttore. Il motivo principale è la semplicità: `[]` solamente due parentesi contro: *new Array()*. Inoltre, poiché JavaScript è altamente dinamico, nulla impedisce a qualcuno di confondersi sovrascrivendo il costruttore, cioè chiamare *new Array()* non deve necessariamente creare un array. Pertanto, consigliamo di attenersi generalmente ai valori letterali degli array. Proseguiamo con le funzionalità avanzate permesse dall'utilizzo dei metodi.

I metodi dei vettori (o array)

Avendo a che fare con i vettori le prime necessità potranno essere quelle di aggiungerci o rimuoverci eventuali elementi in base alle necessità. Per questo potremo utilizzare quattro metodi:

- *pop* rimuove un elemento dalla fine del vettore.
- *push* aggiunge un elemento alla fine dell'array.
- *shift* rimuove un elemento dall'inizio dell'array.
- *slice(a, b)* ritorna un array compreso tra a e b.
- *splice()* aggiunge, rimuove o sostituisce un elemento in un vettore.
- *unshift* aggiunge un elemento all'inizio dell'array.
- *.filter* ritorna un array filtrato dagli elementi che passino *true*.
- *.forEach* avvia una funzione per ogni elemento dell'array.
- *.map* ritorna un array da una funzione eseguita su un altro array.
- *.reduce* restringe l'array ad un valore tramite un loop.

Osserviamo un listato di esempio che prova sul campo alcuni di questi metodi:

```
const gatto = [];
assert(gatto.length === 0, "Il vettore è vuoto");
gatto.push("Kitty");
assert(gatto[0] === "Kitty");
assert(gatto.length === 1, "Abbiamo un gatto nell'array");
gatto.push("Briciola");
assert(gatto[1] === "Briciola", "Briciola viene aggiunto alla fine dell'array");
assert(gatto.length === 2, "Abbiamo due gatti nell'array!");
gatto.unshift("Fuffi");
assert(gatto[0] === "Fuffi", "Ora Fuffi è il primo gatto");
assert(gatto[1] === "Kitty", "Kitty è passato al secondo posto");
assert(gatto[2] === "Briciola", "E Briciola al terzo posto");
assert(gatto.length === 3, "Abbiamo tre gatti nell'array!");
```

Utilizziamo il metodo *unshift* per inserire l'elemento all'inizio dell'array. Gli altri vengono adeguati di conseguenza.

```
const ultimoGatto = gatto.pop();
assert(ultimoGatto === "Briciola", "Abbiamo rimosso Briciola dalla fine dell'array");
assert(gatto[0] === "Fuffi", "Fuffi è ancora il primo gatto");
assert(gatto[1] === "Kitty", "Kitty è ancora secondo");
assert(gatto.length === 2, "Ora abbiamo due gatti nell'array")
```

Estrae l'ultimo elemento dall'array.

```
const primoGatto = gatto.shift();
assert(primoGatto === "Fuffi", "Abbiamo rimosso Fuffi dall' inizio dell'array");
assert(gatto[0] === "Kitty", "Kitty è al primo posto");
```

```
assert(gatto.length === 1, "Abbiamo solo un gatto");
```

Rimuove il primo elemento dall'array. Gli altri elementi vengono spostati di conseguenza a sinistra.

I metodi *pop* e *push* influenzano solo l'ultimo elemento in un array: *pop* rimuovendo l'ultimo elemento e *push* inserendo un elemento alla fine della matrice. D'altra parte, i metodi *shift* e *unshift* cambiano il primo elemento nell'array. Ciò significa che gli indici di tutti gli elementi dell'array successivi devono essere modificati. Per questa ragione, *push* e *pop* sono operazioni notevolmente più veloci rispetto a *shift* e *unshift* e si consiglia di utilizzarli a meno che non si abbia una buona ragione per fare altrimenti.

L'esempio precedente ha rimosso gli elementi dall'inizio e dalla fine dell'array. Ma questo è troppo vincolante: in generale, dovremmo essere in grado di rimuovere elementi da qualsiasi posizione di array.

Come risposta a questi problemi, tutti gli array JavaScript hanno accesso al metodo *splice*: a partire da un dato indice, questo metodo rimuove e inserisce elementi:

```
const gatto = ["Fuffi", "Kitty", "Briciola", "Coda"];
let gattoRimosso = gatto.splice(1, 1);
assert(gattoRimosso.length === 1, "Un gatto è stato rimosso");
assert(gattoRimosso[0] === "Kitty");
assert(gatto.length === 3, "Ora ci sono tre elementi nell'array");
```

Uso del metodo *splice* per rimuovere un oggetto dall'array.

```
assert(gatto[0] === "Fuffi", "Il primo gatto rimane Fuffi");
assert(gatto[1] === "Briciola", "Briciola al secondo posto");
assert(gatto[2] === "Coda", "Coda terzo");
```

La schiera dei gatti non contiene più Kitty; gli elementi successivi sono stati spostati automaticamente.

```
gattoRimosso = gatto.splice(1, 2, "Pelo", "Zampa");
assert(gattoRimosso.length === 2, "Rimossi due gatti");
assert(gattoRimosso[0] === "Briciola", "Briciola rimosso");
assert(gattoRimosso[1] === "Coda", "Coda rimosso");
assert(gatto.length === 3, "Aggiunti di nuovo i gatti!");
assert(gatto[0] === "Fuffi");
assert(gatto[1] === "Pelo");
assert(gatto[2] === "Zampa");
```

Questo esempio ci fa capire come facilmente rimuovere o inserire elementi. Ora vedremo le operazioni più comuni sugli array: *Iterazione degli array*, *Mappatura elementi dell'array esistenti per creare uno nuovo basato su di essi*, *Testare elementi dell'array per verificare se soddisfano determinate condizioni*, *Trovare elementi specifici dell'array*, *Aggregare array e calcolarne un singolo valore in base ai suoi elementi*.

Vettori e loop

Per l'iterazione useremo il metodo `forEach()`, il quale eseguirà il callback fornito una volta per ciascun elemento presente nell'array in ordine crescente. Non sarà evocato per le proprietà dell'indice cancellate o non inizializzate, ma lo sarà tramite: il valore dell'elemento, l'indice dell'elemento o l'array da iterare.

Se viene fornito il parametro `thisArg` a `forEach()`, esso verrà utilizzato come valore `this` del callback. In caso contrario, lo sarà il valore `undefined`. Il risultato sarà determinato dalle stesse regole a cui è soggetto `this` nella funzione.

L'intervallo di elementi elaborati da `forEach()` viene impostato prima dell'iniziale chiamata del callback. Dopo che `forEach()` è partito, gli elementi aggiunti all'array non saranno calcolati dal callback. Se i valori degli elementi esistenti dell'array vengono modificati, saranno passati al callback quelli nel momento in cui `forEach()` controlla. Gli elementi che vengono cancellati prima, ovviamente non saranno considerati. Se gli elementi già visitati vengono rimossi (ad esempio usando `shift()`) durante l'iterazione, i successivi verranno saltati.

`forEach()` esegue la funzione callback una volta per ogni elemento dell'array; a differenza di `map()` o `reduce()` restituisce sempre `undefined` e non è concatenabile. Il tipico caso d'uso è eseguire effetti collaterali alla fine della catena.

```
const arrayprova = ['mario', 'gino', 'sara'];
arrayprova.forEach(element => console.log(element));
```

Avremo come risultato *mario*, *gino*, *sara*. Questo esempio utilizza un ciclo `for` per controllare ogni elemento dell'array. Vediamo una variante dello stesso codice, sicuramente più familiare:

```
const tizi = ["Mario", "Gino", "Sara"];
for(let x = 0; x < tizi.length; x++){
```

```
console.log(tizi[x] !== null, tizi[x]);  
}
```

Qui vediamo come, per far funzionare il loop dovremmo gestire un file variabile, in questo caso *x*, che funge da contatore. Praticamente specifica il numero fino al quale vogliamo contare (*tizi.length*), e definire come verrà modificato il contatore (*x++*). Questo esempio è più familiare ma indubbiamente più arzigogolato e soggetto a possibili errori.

Riguardiamo come arrivare allo stesso risultato servendosi del metodo integrato *forEach*:

```
const verdure = ["Pomodoro", "Zucchina", "Melanzane"];  
verdure.forEach(element => console.log(element));
```

Utilizziamo un callback (in questo caso, una funzione freccia) che viene chiamato immediatamente, per ogni elemento dell'array. Questo è tutto: non dovrai più preoccuparti dell'indice iniziale, delle condizioni finali o della natura esatta dell'incremento. Il motore JavaScript si occupa di tutto ciò per noi, dietro le quinte. Da notare quanto sia più facile capire questo codice e di conseguenza ci siano meno probabilità di generare fastidiosi errori.

Utilizzare il metodo map in un vettore

Come impareremo, la creazione di nuovi vettori basati sugli elementi di uno già esistente è una pratica molto utilizzata. Questo passaggio viene denominato *mappatura* e cercheremo di sfruttarlo a nostro vantaggio, proviamo:

```
const soldati = [  
  {nome: "Mario", arma: "pistola"},  
  {nome: "Pino", arma: "fucile"},  
  {nome: "Gianni", arma: "fionda"}  
];  
const armi = soldati.map(soldato => soldato.arma);  
  
assert(armi[0] === "pistola" && armi[1] === "fucile" && armi[2] ===  
"fionda" && armi.length === 3); // true
```

Il metodo integrato *map* costruisce un vettore completamente nuovo e quindi itera su quello iniziale. Per ogni elemento del primo vettore, *map* inserisce esattamente un elemento nell'array appena costruito, in base al risultato della richiamata fornita per *mappare*.

A questo punto potrebbe sorgere spontanea una domanda (a parte quella della mancata comprensione di cui sopra): perché non utilizzare il metodo *forEach*?

```
const armi = [];  
soldati.forEach(soldato => {armi.push(soldato.arma);  
});  
  
assert(armi[0] === "pistola" && armi[1] === "fucile" && armi[2] ===  
"fionda" && armi.length === 3); // true
```

Creiamo un nuovo array e utilizziamo un ciclo *forEach* sui soldati per estrarre le singole armi. Quindi, per ogni oggetto soldato, aggiungiamo l'arma alla schiera.

Quando vi troverete di fronte ad un problema di questo tipo dovrete trovare la soluzione migliore, sicuramente in questo caso il metodo *map* ha i suoi vantaggi, comunque è bene conoscere le alternative e testarle sul campo.

Ora che sappiamo come mappare i vettori, vediamo come testare gli elementi di un insieme per determinate condizioni.

Controllare gli elementi di un vettore

Quando lavoriamo con raccolte di elementi, ci imbattiamo spesso in situazioni in cui dobbiamo sapere se tutti o almeno alcuni degli elementi dell'array soddisfano determinate condizioni. Per scrivere questo codice nel modo più efficiente possibile, tutti gli array JavaScript hanno accesso ai metodi *every* e *some*:

```
const soldati = [  
  {nome: "Mario", arma: "pistola"},  
  {nome: "Pino", arma: "fucile"},  
  {nome: "Gianni"}  
];  
  
const nS = soldati.every(soldato => "nome" in soldato);  
const aS = soldati.every(soldato => "arma" in soldato);  
  
assert(nS, "Ogni soldato ha un nome"); // true  
assert(aS, "Ma non tutti i soldati sono armati"); // false
```

Qui sopra controlliamo che ogni soldato abbia un nome e un'arma utilizzando il metodo *every*, il quale richiede un callback che, per ogni

soldato nella raccolta, controlla se c'è il nome e l'arma. *Every* restituisce *true* solo se il callback passato restituisce *true* per ogni elemento dell'array. In altri casi, ci interessa solo se alcuni elementi dell'array soddisfano una determinata condizione. Per queste situazioni, possiamo utilizzare il metodo integrato *some*:

```
const aS = soldati.some(soldato => "arma" in soldato);
```

A partire dal primo elemento dell'array, *some* chiama il callback finché non ne viene trovato uno per il quale il callback restituisce un valore *true*, in caso contrario, il valore restituito è *false*.

Utilizzare i metodi di ricerca in un vettore

Un'altra operazione comune che sei obbligato a usare, prima o poi, è trovare elementi in un array. Ancora una volta, questa attività è notevolmente semplificata con un altro metodo integrato: *find*:

```
const gatti = [
  {nome: "Apollo", cibo: "gamberi"},
  {nome: "Briciola"},
  {nome: "Coda", cibo: "seppie"}
];

const gS = gatti.find(gatto => {return gatto.cibo === "seppie";
});
assert(gS.nome === "Coda" && gS.cibo === "seppie"); // true
```

Utilizza il metodo *find* per trovare il primo elemento dell'array che soddisfa una determinata condizione, rappresentato da un callback passato.

```
const gZ = gatti.find(gatto => {return gatto.cibo === "sardine";
});
assert(gZ === undefined); // true
```

Il metodo *find* restituisce *undefined* se un elemento non può essere trovato.

```
const cG = gatti.filter(gatto => "cibo" in gatto)
assert(cG.length === 2, "Ci sono due gatti che mangiano:");
// true
assert(cG[0].nome === "Apollo" && cG[1].nome === "Coda");
// true
```


Utilizza il metodo *filter* per trovare più elementi che soddisfano tutti una determinata condizione, passandogli un callback che viene richiamato per ogni elemento nella raccolta finché non viene trovato l'elemento mirato. Ciò è indicato dal callback che restituisce true. Per esempio, l'espressione:

```
gatti.find(gatto => gatto.cibo === "seppia");
```

trova *Coda*, il primo gatto che mangia una seppia.

Se abbiamo esaminato l'intero array senza che un singolo elemento restituisse true, il risultato finale della ricerca non è definito. Ad esempio, il codice:

```
gZ = gatti.find(gatto => {return gatto.cibo === "sardine";
```

restituisce indefinito, perché non c'è un gatto che mangia le *sardine*.

Se dobbiamo trovare più elementi che soddisfano un determinato criterio, possiamo utilizzare il metodo *filter*, che crea un nuovo array contenente tutti gli elementi che soddisfano quel criterio. Ad esempio, l'espressione:

```
const cG = gatti.filter(gatto => "cibo" in gatto);
```

crea un nuovo array di gatti affamati che contiene solo gatti con il loro pasto. Il povero Briciola a digiuno viene escluso.

In questo esempio, hai visto come trovare elementi particolari in un array, ma in molti casi potrebbe anche essere necessario trovare l'indice di un elemento. Diamo un'occhiata più da vicino, con il seguente esempio:

```
const gatti = ["Apollo", "Briciola", "Coda", "Briciola"];
assert(gatti.indexOf("Briciola") === 1); // true
assert(gatti.lastIndexOf("Briciola") === 3); // true
const bI = gatti.findIndex(gatto => gatto === "Briciola");
assert(bI === 1); // true
```

Per trovare l'indice di un particolare elemento, usiamo il metodo integrato *indexOf*, passandogli l'elemento di cui vogliamo trovare l'indice:

```
gatti.indexOf("Briciola");
```

Nei casi in cui un particolare elemento può essere trovato più volte in un array (come nel caso con "*Briciola*" e l'array *gatto*), potremmo anche essere

interessati a trovare l'ultimo indice in cui appare *Briciola*. Per questo, possiamo utilizzare il metodo *lastIndexOf*:

```
gatti.lastIndexOf("Briciola");
```

Infine, nel caso più generale, quando non abbiamo un riferimento all'elemento esatto di cui vogliamo cercare l'indice, possiamo usare il metodo *findIndex*:

```
const bI = gatti.findIndex(gatto => gatto === "Briciola");
```

Il metodo *findIndex* accetta un callback e restituisce l'indice del primo elemento per il quale il callback restituisce *true*. In sostanza, funziona praticamente come il metodo *find*, l'unica differenza è che *find* restituisce un particolare elemento, mentre *findIndex* restituisce l'indice di quell'elemento.

Utilizzare il metodo sort per sistemare gli elementi in un vettore

Come abbiamo ampiamente osservato precedentemente, i nostri insiemi sono un po' come gli armadi di casa, hanno bisogno di essere messi in ordine, ogni cosa al suo posto, i pantaloni, i maglioni e così via. Lo stesso vale per i vettori dove potremo utilizzare il metodo *sort*, ripassiamo ancora la sintassi e le sue peculiarità:

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

E fin qui tutto bene, ma se i numeri fossero così:

```
let numeri = [30, 1, 2, 4, 35]
numeri.sort();
console.log(numeri); // 1, 2, 30, 35, 4
```

In questi casi sappiamo benissimo che i numeri vengono confrontati in base all'ordine *code point* di *Unicode* e quindi per forza di cose dovremo avvalerci di una funzione di confronto che ci permetta di ordinare il nostro vettore in maniera corretta:

```
function(x, y) {return x - y}
```

Questa semplice espressione ci aiuterà nel nostro lavoro. L'unica cosa che dovremmo fare è fornire un callback che informa l'algoritmo sulla relazione tra due elementi dell'array. I possibili risultati saranno i seguenti:

Se x fosse minore di y restituendo un risultato negativo allora x andrà prima di y , se la differenza riportasse un valore uguale a 0, x e y sarebbero su un piano di parità e il loro ordinamento sarebbe indifferente, infine se x fosse maggiore di y , allora x dovrebbe venire dopo di y .

L'ordinamento effettivo viene eseguito dietro le quinte in automatico:

```
let numeri = [30, 1, 2, 4, 35];
numeri.sort(function(x, y) {
  return x - y;
});
console.log(numeri); // 1, 2, 4, 30, 35
```

Ricordiamoci delle funzioni freccia (viste nel capitolo 6) e riscriviamo ulteriormente il codice:

```
let numeri = [30, 1, 2, 4, 35];
numeri.sort((x, y) => x - y);
console.log(numeri);
```

Vediamo lo stesso con delle stringhe di testo da ordinare in ordine alfabetico:

```
const gatto = [{nome: "Coda"}, {nome: "Briciola"}, {nome: "Apollo"}]

gatto.sort(function(gatto1, gatto2) {if (gatto1.nome <gatto2.nome)
{return -1;} if (gatto1.nome> gatto2.nome) {return 1;}

return 0;
});

console.log(gatto[0].nome); // Apollo
console.log(gatto[1].nome); // Briciola
console.log(gatto[2].nome); // Coda
```

Qui abbiamo una serie di gatti con un nome. Il nostro obiettivo è ordinare quell'array in ordine alfabetico, usando naturalmente la funzione di ordinamento:

```
gatto.sort(function(gatto1, gatto2) {
  if (gatto1.nome <gatto2.nome) {
    return -1;
  }
})
```

```
    if (gatto1.nome > gatto2.nome) {  
        return 1;  
    }  
    return 0;  
});
```

Per la funzione di ordinamento dobbiamo solo passare un callback che viene utilizzato per confrontare due elementi dell'array. Poiché vogliamo fare un confronto lessicale, affermiamo che se il nome di *gatto1* è "minore" del nome di *gatto2*, la richiamata restituisce -1 (*gatto1* dovrebbe venire prima di *gatto2*, nell'ordine finale ordinato); se è maggiore, il callback restituisce 1 (*gatto1* dovrebbe venire dopo *gatto2*); se sono uguali, il callback restituisce 0. Si noti che possiamo usare semplici operatori minore di (<) e maggiore di (>) per confrontare due nomi del gatto. Questo è tutto! Il resto dei dettagli essenziali dello smistamento è lasciato al motore JavaScript, senza che noi dobbiamo preoccuparci di loro.

Il metodo reduce

Questo metodo potrà servirci per evitare di incappare in tediosi loop senza via d'uscita. *Reduce* funziona prendendo il valore iniziale, chiamando poi la funzione di callback "riduttore" su ogni elemento dell'array, fino ad arrivare ad un unico valore. Se non ne viene fornito uno verrà utilizzato 0, mentre il loop comincerà da 1. Vediamo come:

```
const numeri = [3, 6, 9, 12, 20];  
const somma = numeri.reduce((ristretti, numeri) => ristretti +  
    numeri, 0);  
console.log(somma); // 50
```

A questo punto i vettori (o array) dovrebbero averci svelato la maggior parte dei loro segreti, quindi è ora di passare al livello successivo studiando le mappe e il costruttore *Set*.

Le mappe in Javascript

L'oggetto *Map* esiste praticamente in tutti i linguaggi di programmazione, in Javascript assomiglia ad *Object*, infatti possiamo memorizzare svariati dati abbinabili ad una chiave, che con *map* possono essere di qualsiasi tipologia, quindi sia una chiave stringa o oggetto, come possiamo vedere nell'esempio di seguito:

```
let mappa = new Map();
```

```
mappa.set('1', 'stringa');  
mappa.set(2, 'numero 2');
```

```
alert(mappa.get('1'));  
alert(mappa.get(2));  
alert(mappa.size);
```

Come abbiamo visto nel capitolo 9, tutti gli oggetti hanno prototipi; anche l'oggetto *map*, e ha comunque accesso alle proprietà degli oggetti prototipo. Una di queste proprietà è il costruttore (il costruttore è la proprietà dell'oggetto prototipo che rimanda alla funzione costruttore).

Come già detto, con gli oggetti le chiavi possono essere solo valori stringa; se vuoi creare una mappatura per qualsiasi altro valore, quel valore verrà convertito silenziosamente in una stringa senza alcun avviso.

Iterazione su Map

Come sappiamo l'iterazione, al verificarsi di una determinata condizione ci permette di eseguire delle istruzioni o una sequenza di esse.

Nel prossimo esempio vediamo come iterare con gli elementi di *map*:

```
let scatolaMap = new Map([  
  ['giornali', 33],  
  ['fotografie', 110],  
  ['scartoffie', 28]  
]);  
  
// itera sulle chiavi, cioè sul contenuto della scatola  
for (let contenuto of scatolaMap.keys()) {  
  alert(contenuto);  
}  
  
// itera sui valori, il numero delle cose contenute  
for (let quantita of scatolaMap.values()) {  
  alert(quantita);  
}  
  
// itera sulle voci contenute nella scatola  
for (let testo of scatolaMap) {  
  alert(testo);  
}
```

Per l'iterazione verrà seguito l'ordine di inserimento.

In ogni iterazione viene fornito un array di due elementi, dove il primo elemento è una chiave e il secondo elemento è il valore di un elemento dalla

nostra mappa. Possiamo anche usarli per iterare sulle chiavi e i valori contenuti in una mappa:

```
const dir = new Map();

dir.set("Totti", "20");
dir.set("Del Piero", "18");
dir.set("Cavani", "25");
```

Crea una *directory* che memorizza il numero di goal di una vecchia stagione di Serie A di calcio:

```
for (let item of dir) {
  assert(element [0]! == null, "Goleador:" + element [0]);
  assert(element [1]! == null, "Goal:" + element [1]);
}

for (let goleador of dir.keys ()) {
  assert(goleador! == null, "Goleador:" + goleador);
  assert(dir.get(goal)! = null, "Goal:" + dir.get(goal));
}
```

Ripete ogni elemento usando il ciclo *for ... of*. Ogni elemento è un array di due elementi: una chiave e un valore. Possiamo anche iterare sulle chiavi usando il metodo integrato *keys*.

```
for (let goleador of dir.values()) {
  assert(goleador! == null, "Goleador:" + goleador);
}
```

Come mostra l'elenco precedente, una volta creata una mappatura, possiamo facilmente iterare usando il ciclo *for ... of*:

```
for (let item of dir) {
  assert(element [0]! == null, "Goleador:" + element [0]);
  assert(element [1]! == null, "Goal:" + element [1]);
}
```

Map contiene un suo metodo simile a quello che è stato già visto precedentemente con gli array: parliamo di *forEach*. Vediamo come lavora assieme ad entrambi i codici visti precedentemente:

```
scatolaMap.forEach((contenuto, quantita, map) => {
  alert(`${contenuto}: ${quantita}`);
});
```

```
dir.forEach((goleador, goal, map) => {  
    alert(`${goleador}: ${goal}`);  
});
```

Con il metodo *delete* potremo rimuovere gli elementi, cioè le chiavi, dalla nostra mappa:

```
scatolaMap.delete('scartoffie');  
dir.delete("Cavani");
```

Uno dei concetti fondamentali quando si ha a che fare con le mappe in Javascript è determinare quando le due chiavi della mappa sono uguali. Infatti chi proviene da un background un po' più tradizionale, come C #, Java o Python, non troverà familiare il prossimo esempio sull'uguaglianza delle chiavi in *map*:

```
const map = new Map();  
const posizione = location.href;  
const a = new URL(posizione);  
const b = new URL(posizione);
```

Abbiamo usato la proprietà incorporata *location.href* per ottenere l'URL della pagina corrente, dopo creiamo due collegamenti alla pagina corrente.

```
map.set(a, {descr: "LinkA"});  
map.set(b, {descr: "LinkB"});
```

Aggiunge una mappatura per entrambi i link,

```
assert(map.get(LinkA).descr === "LinkA", "Posizione LinkA");  
assert(map.get(LinkB).descr === "LinkB", "Posizione LinkB");  
assert(map.size === 2, "Due mappature");
```

Ogni collegamento riceve la propria mappatura, anche se punta alla stessa pagina. In questo esempio usiamo la proprietà incorporata *location.href* per ottenere l'URL della pagina web. Successivamente, utilizzando il costruttore *URL*, creiamo due nuovi oggetti che si collegano alla pagina corrente. Associamo quindi un oggetto descrizione a ciascun collegamento. Infine, controlliamo che siano state create le mappature corrette.

Persone che hanno lavorato principalmente in JavaScript potrebbero non trovare questo risultato inaspettato: abbiamo due oggetti diversi per i quali creiamo due diverse mappature. Ma nota che i due oggetti URL, anche se sono oggetti separati, puntano ancora alla stessa posizione URL: la posizione

della pagina corrente. Potremmo sostenere che, durante la creazione delle mappature, questi due oggetti dovrebbero essere considerati uguali. Ma in JavaScript, non è possibile sovraccaricare l'operatore di uguaglianza e i due oggetti, anche se hanno lo stesso contenuto, sono sempre considerati diversi. Questo non è il caso di altri linguaggi, come Java e C #, quindi facciamo attenzione!

Ci siamo fatti un'idea delle mappe e sulla loro funzionalità, adesso passiamo alle raccolte di oggetti unici e all'uso del costruttore *Set*.

Il costruttore Set

La funzione di costruzione chiamata *Set* (con la S maiuscola) è alla base della creazione di insiemi. Essa può accettare un array di elementi con cui il set viene inizializzato. Diamo un'occhiata a un esempio:

```
const cani = new Set(["Thor", "Fido", "Macchia"]);

assert(cani.has("Macchia"), "Macchia è presente nel set");
assert(cani.size === 3, "Ci sono tre cani nel set")
assert(!cani.has("Musetto"), "Musetto non c'è ancora");

cani.add("Musetto");
assert(cani.has("Musetto"), "Musetto aggiunto");
assert(cani.size === 4, "Adesso ci sono quattro cani");
assert(cani.has("Fido"), "Fido è già parte del gruppo");

cani.add("Fido");
assert(cani.size === 4, "L'aggiunta doppia non ha effetto");
```

Possiamo aggiungere nuovi elementi che non sono già contenuti nel set, tuttavia l'aggiunta di elementi esistenti non ha alcun effetto, come vediamo nell'esempio mostrato qui sopra.

Ora useremo il costruttore per creare un nuovo *set* che conterrà alcuni cani. Se non passiamo alcun argomento, viene creato un insieme vuoto. Possiamo anche passare in un array, come questo, che precompila il set:

```
new Set(["Thor", "Fido", "Fido", "Musetto"]);
```

Come già accennato, gli insiemi sono raccolte di oggetti unici e il loro scopo principale è impedirci di memorizzare più occorrenze dello stesso oggetto. Nel caso specifico *"Fido"*, viene aggiunto solo una volta, anche se abbiamo tentato di inserirlo doppio nel nuovo set appena creato.

Vediamo ora i diversi metodi a disposizione, ad esempio, il metodo *has* controlla se un elemento è contenuto nel set:

```
cani.has("Fido");
```

e il metodo *add* viene utilizzato per aggiungere elementi unici al set:

```
cani.add("Musetto");
```

Se vogliamo sapere quanti articoli ci sono in un set o insieme, useremo la proprietà *size*. Simili alle mappe e agli array, gli insiemi sono raccolte, quindi possiamo iterare su di esse con un ciclo *for ... of*. Gli elementi vengono sempre ripetuti nell'ordine in cui sono stati inseriti.

Ora che abbiamo studiato le basi, vediamo le operazioni classiche che potremo effettuare anche con i *Set*.

Operazioni tra set

Questa tipologia di oggetto ha una peculiarità che ci permetterà di effettuare alcune semplici operazioni con esse, come l'unione (o la somma) di due set, la quale crea un nuovo insieme che contiene tutti gli elementi dei primi due. Osserviamo come unire due elenchi di cani e gatti per ottenerne uno nuovo chiamato *pet*:

```
const cani = ["Thor", "Fido", "Macchia"];
const gatti = ["Thor", "Pussy", "Kitty"];
const pet = new Set([...cani, ...gatti]);

assert(pet.has("Thor") && pet.has("Fido") && pet.has("Macchia") &&
pet.has("Pussy") && pet.has("Kitty")); // true
assert(pet.size === 5, "Ci sono 5 adorabili pet in totale");
```

Non è possibile avere due copie dello stesso elemento nel nuovo set nonostante il caso di omonimia sussistente tra un cane ed un gatto, infatti avremo solo un *Thor* in *pet*.

Da notare che quando creiamo questo nuovo set, usiamo l'operatore *spread* *[...cani, ...gatti]* (vedi il capitolo 6).

Oltre ad unire seccamente due interi set avremo anche la possibilità di selezionarne il contenuto e di utilizzarne solo una parte. Ad esempio, possiamo scegliere gli omonimi, come in questo caso *Coda*, e farne un nuovo set:

```
const gatti = new Set(["Kitty", "Briciola", "Coda"]);
const cani = new Set(["Fuffi", "Fido", "Coda"]);
const omonimi = new Set([...gatti].filter (gatti => cani.has(gatti)));
```

Per farlo, come già visto useremo l'operatore *spread* e poi tramite il metodo *.filter* selezioneremo gli omonimi:

```
assert(omonimi.size === 1, "C'è solo un omonimo");
assert(omonimi.has("Coda"), "Coda è il suo nome");
```

Ovviamente per questa operazione chiamata *intersezione* avremmo anche potuto utilizzare come parametro di scelta la lunghezza della stringa (ad esempio tutte le parole di 4 lettere), o le iniziali (tutte le parole che iniziano con c) e così via.

Infine, come ultima operazione, potremo anche eseguire una differenza tra due *Set*, ottenendone uno nuovo che, al contrario dell'operazione precedente, non abbia un oggetto in comune:

```
const gatti = new Set(["Briciola", "Kitty", "Coda"]);
const cani = new Set(["Fuffi", "Osso", "Coda"]);
const soloGatti = new Set([...gatti].filter (gatti => !cani.has(gatti)));
assert(soloGatti.size === 2);
assert(soloGatti.has("Briciola") && soloGatti.has("Kitty"));
// true
```

Da segnalare che il punto esclamativo posizionato davanti ad una funzione o a un'espressione, come in questo caso, segnala al motore Javascript di **invertire un'operazione booleana**, quindi sostanzialmente nel caso specifico stiamo chiedendo di restituire tra i due set solo dei gatti.

I nuovissimi metodi per gli array

Questo paragrafo è dedicato a chi ha raggiunto almeno un livello medio nella programmazione Javascript, infatti parleremo di metodi standardizzati nella quattordicesima edizione dell'ECMA (a Giugno del 2023) quindi, chi ancora deve assimilare bene i concetti base può tranquillamente passare oltre. Tutti gli altri si assicurino che i nuovi metodi siano pienamente compatibili con il browser prima di implementarli.

Ultima annotazione, nei prossimi esempi ci serviremo del metodo *assert.deepEqual()*, il quale verifica se due oggetti e i loro oggetti figli sono uguali, utilizzando l'operatore `==`. Se questi sono diversi, otterremo un errore di asserzione e il programma sarà terminato.

Parliamo adesso di queste nuove funzionalità, le quali forniscono dei metodi aggiuntivi su *Array.prototype* e *TypedArray.prototype* per abilitare le modifiche sull'array e poi restituirne una nuova copia con i cambiamenti apportati. Essenzialmente sono introdotte le seguenti proprietà della funzione su *Array.prototype*:

Array.prototype.toReversed() -> *Array*

Questo metodo restituisce una copia dell'array invertito senza modificare l'array di origine.

Array.prototype.toSorted(compareFn) -> *Array*

Questo metodo restituisce una copia dell'array ordinato utilizzando la funzione di confronto *compareFn*, senza modificare l'array di origine.

Array.prototype.toSpliced(start, deleteCount, ...items) -> *Array*

Questo metodo restituisce una copia dell'array dopo aver eseguito l'operazione di splicing specificata, senza modificare l'array di origine.

Array.prototype.with(index, value) -> *Array*

Questo metodo restituisce una copia dell'array dopo aver aggiunto o modificato un elemento all'indice specificato, senza modificare l'array di origine.

Tutti questi metodi hanno il vantaggio di mantenere inalterato l'array di destinazione e ne restituiscono una copia con le modifiche applicate.

I nuovi metodi *toReversed*, *toSorted*, e *with*, sono disponibili anche per gli array tipizzati, e le relative firme dei metodi sono simili a quelle degli array standard.

TypedArray.prototype.toReversed() -> *TypedArray*

TypedArray.prototype.toSorted(compareFn) -> *TypedArray*

TypedArray.prototype.with(index, value) -> *TypedArray*

Vediamo adesso qualche esempio concreto:

```
const sequenza = [6, 7, 8, 9];
sequenza.toReversed(); // [9, 8, 7, 6]
sequenza; // [6, 7, 8, 9]

const typedSequenza = new Uint8Array([3, 1, 2]);
typedSequenza.toSorted(); // Uint8Array[1, 2, 3]
typedSequenza; // Uint8Array[3, 1, 2]

const numeri = [1, 1, 3];
numeri.with(1, 2); // [1, 2, 3]
numeri; // [1, 1, 3]
```

I nuovi metodi con gli array sono disponibili anche per gli array tipizzati. Andiamo avanti, ricordiamo che la maggior parte dei metodi degli array non sono distruttivi, cioè non modificano gli array su cui vengono invocati, ad esempio:

```
const test = ['a', 'b', 'b', 'a'];
const result = test.filter(x => x !== 'b');
assert.deepEqual(result, ['a', 'a']); // L'array result contiene le
stringhe 'a' non distruttivamente
assert.deepEqual(test, ['a', 'b', 'b', 'a']); // L'array test rimane
invariato
```

Vediamo come in pratica sono state rimosse tutte le stringhe *b* dall'array *test* in modalità non distruttiva. Tuttavia, ci sono anche metodi distruttivi come *.sort()*, ad esempio, che cambia i ricevitori:

```
const test = ['c', 'a', 'b'];
const result = test.sort();

assert.deepEqual(result, ['a', 'b', 'c']); // L'array result è stato
modificato
assert.ok(result === test); // L'array result e test sono lo stesso
oggetto
assert.deepEqual(test, ['a', 'b', 'c']);
```

test.sort() prima ordina l'array in posizione e poi lo restituisce. In questa riga: *assert.ok(result === test)*; possiamo vedere che *test* (il destinatario della chiamata al metodo) e *result* (il valore restituito dal metodo) sono lo stesso oggetto.

Ricapitoliamo quindi i tre metodi dell'array distruttivi:

```
.reverse() , .sort() , .splice()
```

Se vogliamo applicare uno di questi metodi a un array senza modificarlo, possiamo utilizzare uno degli esempi riportati di seguito:

```
const s1 = array.slice().sort();
const s2 = [...array].sort();
const s3 = Array.from(array).sort();
```

In pratica, prima facciamo una copia dell'array, e andiamo ad apportarci le modifiche.

In alternativa abbiamo le versioni *non distruttive* dei tre metodi distruttivi dell'array che ci eviteranno di servirci dei modelli sopra menzionati:

.toReversed(): Array al posto di *.reverse()*
.toSorted(compareFn): Array al posto di *.sort()*
.toSpliced(start, deleteCount, ...items): Array al posto di *.splice()*

È stato anche introdotto anche un metodo non distruttivo che non ha un metodo distruttivo corrispondente:

.with(index, value): Array

Questo metodo sostituisce in modo non distruttivo un elemento array in un dato indice, come ad esempio *array[index] = value*.

Andiamo a vedere degli esempi più concreti partendo da *.toReversed()*:

```
const array = ['1', '2', '3'];

assert.deepEqual(
  array.toReversed(), ['3', '2', '1']
);
assert.deepEqual(
  array, ['1', '2', '3']
);
```

Non potendo ancora usufruire di *.toReversed()* vediamo del codice surrogato compatibile:

```
if (!Array.prototype.toReversed) {
  Array.prototype.toReversed = function() {
    return this.slice().reverse();
  };
}
```

Passiamo a *.toSorted(compareFn)*:

```
const array = ['3', '1', '2'];

assert.deepEqual(
  array.toSorted(), ['1', '2', '3']
);
assert.deepEqual(
  array, ['3', '1', '2']
);
```

Vediamo ancora del codice surrogato compatibile per *.toSorted()* :

```
if (!Array.prototype.toSorted) {
  Array.prototype.toSorted = function(compareFn) {
    return this.slice().sort(compareFn);
  };
}
```

Eccoci a *.toSpliced*. Prima di procedere è meglio ricordare che Il metodo *.splice()* è un po' più complesso degli altri metodi distruttivi. Sappiamo che, ad esempio, con il paramentro *deleteCount* andremo ad eliminare gli elementi, a partire dal primo dell'indice. Si inserisce l'*item* (l'articolo contenuto nell'insieme) nell'indice *start* che restituisce gli elementi eliminati. In pratica, con *deleteCount* gli elementi dell'array vengono sostituiti con gli *items*:

```
const array = ['a', 'b', 'c', 'd'];

assert.deepEqual(
  array.splice(1, 2, 'Z'), ['b', 'c']
);
assert.deepEqual(
  array, ['a', 'Z', 'd']
);
```

.toSpliced() è la versione non distruttiva di *.splice()*. Deve restituire la versione modificata del suo ricevitore e quindi non ci dà accesso agli elementi eliminati:

```
const array = ['a', 'b', 'c', 'd'];

assert.deepEqual(
  array.toSpliced(1, 2, 'Z'), ['a', 'Z', 'd']
);
```

```
assert.deepEqual(
  array, ['a', 'b', 'c', 'd']
);
```

Vediamo ancora del codice surrogato compatibile per *.toSpliced()*:

```
if (!Array.prototype.toSpliced) {
  Array.prototype.toSpliced = function(start, deleteCount, ...items)
  {
    const copia = this.slice();
    copia.splice(start, deleteCount, ...items);
    return copia;
  };
}
```

Infine siamo arrivati a *.with()*:

```
const array = ['a', 'b', 'c'];

assert.deepEqual(
  array.with(1, 'Z'), ['a', 'Z', 'c']
);
assert.deepEqual(
  array, ['a', 'b', 'c']
);
```

L'ultimo codice surrogato compatibile per *.with()*:

```
if (!Array.prototype.with) {
  Array.prototype.with = function(index, value) {
    const copia = this.slice();
    copia[index] = value;
    return copia;
  };
}
```

Quiz & esercizi

1) Aggiungere al listato l'istruzione per ridurre il vettore ad un unico elemento:

```
lista = new Array
lista[0]="Pane"; lista[1]="Pomodori"; lista[2]="Pasta";
```

2) Utilizzare il metodo *unshift* per modificare la posizione dei pomodori da seconda a prima nel listato precedente.

3) Utilizzare il metodo *reduce* per ottenere zero come risultato.

- 4) Creare un Set e utilizzare l'intersezione per accomunare i vettori.
- 5) Iterare sul seguente listato per ottenere il contenuto del frigo sulla console:

```
let frigo = new Map([
  ['carote', 5],
  ['pomodori', 4],
  ['limoni', 3]
]);
```

- 6) Scrivi una funzione che prenda due array e restituisca un nuovo array che contiene tutti gli elementi unici da entrambi gli array.
- 7) Scrivi una funzione che prenda due array e restituisca un nuovo array che contiene gli elementi che sono presenti in entrambi gli array.
- 8) Scrivi una funzione che prenda due array e restituisca un nuovo array che contiene gli elementi presenti solo nel primo array ma non nel secondo.
- 9) Scrivi una funzione che prenda un elemento e un array e restituisca true se l'elemento è presente nell'array, altrimenti restituisca false.
- 10) Scrivi una funzione che prenda un array e rimuova tutti gli elementi duplicati, restituendo un nuovo array con elementi unici.
- 11) Scrivi una funzione che prenda due oggetti Set e restituisca un nuovo oggetto Set che contiene tutti gli elementi unici da entrambi gli oggetti Set.
- 12) Scrivi una funzione che prenda due array e verifichi se il primo array è un sottoinsieme del secondo. Restituisci true se è un sottoinsieme, altrimenti false.
- 13) Scrivi una funzione che calcoli il numero di elementi presenti in un insieme (array o Set).

Riassunto

In questo paragrafo abbiamo trattato le operazioni fondamentali che eseguiremo con i vettori, array o insiemi, quindi con tutte quelle raccolte di stringhe o numeri con cui avremo a che fare sicuramente nel corso della creazione di un'applicazione.

Le diverse tipologie di istruzioni, come *map*, *set* o il vettore standard, variano in base alle nostre esigenze, infatti il primo ha diversi metodi al nostro servizio, il secondo servirà con gli insiemi di oggetti univoci.

Infine abbiamo avuto in anteprima i nuovi metodi per gli array, standardizzati nella quattordicesima edizione di ECMA, con anche degli esempi di codice che simula in nuovi metodi ed è compatibile con tutti i browser.

6 - Le funzioni

Funzioni e dichiarazione di variabili

Le funzioni come oggetti software

Le diverse tipologie di funzione

Arguments e this

Come invocare le funzioni

Problemi e paradossi delle funzioni

Le funzioni in JavaScript sono un concetto fondamentale che consente di organizzare e riutilizzare il codice in modo efficiente. Sono blocchi di codice autonomi che eseguono una specifica serie di istruzioni quando vengono chiamati o invocati. In JavaScript sono flessibili e versatili, consentendo di passare argomenti, restituire valori e persino definire funzioni annidate.

Le funzioni possono essere considerate come "mattoni" di base per la costruzione di applicazioni complesse. Consentono di suddividere il codice in porzioni gestibili, facilitando la manutenzione e la comprensione del programma nel suo complesso.

JavaScript offre diverse modalità per definire funzioni, inclusi i classici blocchi di codice denominati "function declarations", le funzioni anonime e le più recenti "arrow functions". Ogni tipo di funzione ha le sue caratteristiche e il suo scopo specifico.

Le funzioni possono essere chiamate in vari modi, tra cui l'invocazione diretta, l'assegnazione a variabili o oggetti e l'utilizzo come callback in eventi e chiamate asincrone. Questa flessibilità consente di utilizzare le funzioni in una vasta gamma di scenari, dall'elaborazione dati all'interazione utente e alla gestione degli eventi.

Inoltre, JavaScript supporta il concetto di funzioni di ordine superiore, che sono funzioni in grado di accettare altre funzioni come argomenti o restituire funzioni come risultato. Questa caratteristica è fondamentale per la programmazione funzionale e l'uso avanzato delle funzioni in JavaScript.

Essenzialmente le funzioni sono un pilastro fondamentale della programmazione in JavaScript, consentendo la creazione di codice organizzato, riutilizzabile e flessibile. Comprendere le funzioni è cruciale per diventare competenti nello sviluppo JavaScript.

Le funzioni sono presenti in qualunque linguaggio di programmazione e, banalmente, come detto sono blocchi di codice che svolgono una o più azioni, vediamo come si comportano direttamente sul campo:

```
function prodotto(x,y){return x * y;}  
let prodottoNum = prodotto(3,3);  
console.log(prodottoNum);
```

Come abbiamo appena visto, si dichiara la funzione, il suo nome (*prodotto*) e dei valori tra parentesi (*x* e *y*) che, in questo caso abbiamo deciso di moltiplicare tra di loro esplicitandolo nel corpo della funzione racchiuso tra le parentesi graffe {*return x * y;*}, nella seconda riga definiamo i numeri da moltiplicare e, alla fine invochiamo la funzione per ottenere il risultato.

Forse stiamo correndo troppo, prima di andare avanti credo sia una buona idea quella di capire la differenza tra i metodi per dichiarare le variabili che, come abbiamo visto, utilizzeremo poi nelle nostre funzioni, parliamo di: *var*, *let* e *const*.

Anche per chi ha già delle basi di programmazione è molto importante comprendere in maniera esaustiva le dichiarazioni di variabili in JavaScript. Difatti esse si comportano in maniera leggermente differente rispetto agli altri linguaggi orientati agli oggetti.

Le variabili in JavaScript potremmo vederle un po' come contenitori con dentro delle informazioni.

Dobbiamo anche etichettare quei contenitori con nomi comprensibili, in modo da poter facilmente risalire al significato e al contesto, praticamente per semplificare il lavoro con il programma.

Adesso andremo a vedere quali sono le differenze principali tra i metodi di dichiarazione delle variabili, essenzialmente: quando e cosa dovremmo usare.

Cominciamo con quello più semplice, che è *var* ; iniziamo dichiarando la variabile e poi assegnandole un nome: *nome*, e un valore: *Mario*:

```
var nome = 'Mario';
```

Dopodichè potremo, ad esempio, dichiararlo di nuovo cambiando il valore:

```
var nome = 'Pietro';  
console.log(nome); // Pietro
```

Vediamo che la variabile *nome* è originariamente dichiarata come *Mario* per poi sovrascriverla con *Pietro*.

Sicuramente non sarà un problema in piccoli programmi che dichiarano una, o poche, variabili con *var*. Tuttavia, consideriamo che l'uso di *var* in JavaScript è considerato obsoleto e non è più raccomandato nelle versioni recenti. Inoltre, quando la base di codice diventa più grande, molto probabilmente potrà capitare di sovrascrivere una variabile senza volerlo, creando non solo una gran confusione, e, ciò che è ancora peggio, senza lasciare alcuna traccia evidente, difatti il debug non restituirà alcun errore per questo. Di conseguenza, la ricerca e la correzione dei bug, diventa un compito difficile e dispendioso in termini di tempo.

Con il rilascio di ES6, è stata introdotta la parola chiave *let* per risolvere questi problemi, infatti se utilizzando *let*, provassimo ad assegnare allo stesso nome una variabile diversa, come *Pietro* nel caso precedente, otterremmo un errore:

```
let nome = 'Mario';  
let nome = 'Pietro';  
>> Syntax error ("nome" è già stato dichiarato)
```

```
let nome = 'Mario';  
let nome = 'Pietro';  
console.log(nome); // Pietro
```

Un'altra importante differenza tra le dichiarazioni di variabili *var*, *let* e *const* è l'ambito o il contesto di esecuzione.

Quando si dichiara una variabile con *var*, questa diventa accessibile globalmente o solo localmente all'interno di una funzione.

La parola chiave *let* si comporta in modo simile, ma con alcune funzionalità extra; quando dichiari una variabile con *let* all'interno di un blocco o di un'istruzione, il suo ambito è limitato solo a quel blocco o istruzione, e questo è normalmente il comportamento dell'ambito variabile che troveremo negli altri linguaggi di programmazione più comuni.

Quindi, quando si capisce la differenza tra *var* e *let*, *const* diventa più facile da comprendere.

Con *const*, dichiarare una variabile che non dovrebbe mai cambiare il suo valore e puoi vederla guardando la parola chiave stessa, *const*, che sta per

una costante, in altre parole, una variabile che non cambia.

Per ricordarlo la prossima volta che scrivi in JavaScript, è meglio usare *let* quando dichiari una variabile che intendi modificare in seguito e usare *const* quando non vuoi che il valore cambi di nuovo.

Oggetti JavaScript

Nella vita reale, un'auto è un oggetto. Essa ha proprietà come il peso e il colore e metodi come l'avvio e l'arresto:

Proprietà:

```
auto.marca = "Lancia";  
auto.modello = "Delta";  
auto.peso = "920 kg";  
auto.colore = "rossa";
```

Metodi:

```
auto.avvio();  
auto.guida();  
auto.frenata();  
auto.arresto();
```

Tutte le auto hanno le stesse proprietà, ma i valori delle proprietà cambiano per ogni modello. Tutte le auto hanno gli stessi metodi, ma i metodi vengono eseguiti in momenti diversi.

Le variabili JavaScript sono contenitori per valori di dati (lo vederemo più avanti) *che vanno inizialmente dichiarate*. Questo codice assegna un valore semplice ("*Lancia*") a una variabile denominata *auto*:

```
let auto; // Dichiarazione della variabile auto  
auto = "Lancia"; // Assegnazione del valore "Lancia" alla variabile auto
```

Anche gli oggetti sono variabili. Ma gli oggetti possono contenere molti valori. Questo codice assegna diversi valori (*Lancia*, *Delta*, *rossa*) a una variabile denominata *auto*:

```
let auto = {marca: "Lancia", modello: "Delta", colore: "rossa"};
```

I valori vengono scritti come coppie "*nome: valore*" (nome e valore separati da due punti). Gli oggetti JavaScript sono contenitori per valori denominati chiamati proprietà o metodi.

Definizione dell'oggetto

Definisci (e crei) un oggetto JavaScript con un oggetto letterale:

```
let persona = {primoNome: "Mario", cogNome: "Rossi", età: 40, coloreOcchi: "Verdi"};
```

Gli spazi e le interruzioni di riga non sono importanti. Una definizione di oggetto può estendersi su più righe:

```
let persona = {  
  primoNome: "Mario",  
  cogNome: "Rossi",  
  età: 40,  
  coloreOcchi: "Verdi"  
};
```

Proprietà dell'oggetto

Le coppie *nome: valori* negli oggetti JavaScript sono chiamate *proprietà*:

<u>Nome</u>	<u>Valore</u>
primoNome:	"Mario"
cogNome:	"Rossi"
età:	40
coloreOcchi:	"Verdi"

Accesso alle proprietà degli oggetti

È possibile accedere alle proprietà degli oggetti in due modi:

nomeOggetto.nomeProprietà

oppure: nomeOggetto["nomeProprietà"]

Esempio 1: nome.cognome;

Esempio 2: nome["cognome"];

I metodi dell'oggetto

Gli oggetti possono anche avere metodi, essi sono azioni che possono essere eseguite sugli oggetti. I metodi vengono memorizzati nelle proprietà come definizioni di funzioni.

<u>Proprietà</u>	<u>Valore</u>
primoNome	"Mario"
cogNome	"Rossi"
età	40
coloreOcchi	"Verdi"

Un metodo è una funzione memorizzata come proprietà.

```
persona = {
  primoNome: "Mario",
  cogNome: "Rossi",
  età: 40,
  nomeIntero: function() {
    return this.primoNome + " " + this.cogNome;
  }
};
console.log(persona.nomeIntero()); // "Mario Rossi"
console.log(persona.età); // Restituirà 40
```

La parola chiave this

In una definizione di funzione, “*this*” si riferisce al "proprietario" della funzione. Nell'esempio precedente, *this* è l'oggetto *persona* che "possiede" la funzione *nome Intero*, in altre parole, *this.primoNome* indica la proprietà dell'oggetto *primoNome*.

Accesso ai metodi degli oggetti

Si accede a un metodo oggetto con la seguente sintassi:

```
nomeOggetto.nomeMetodo()

nome = persona.nomeIntero();
```

Se accedi a un metodo senza le parentesi (), verrà restituita la definizione della funzione:

```
nome = persona.nomeIntero;
```

È importante non dichiarare stringhe, numeri e booleani come oggetti.

Quando una variabile JavaScript viene dichiarata con la parola chiave "*new*", la variabile viene creata come oggetto:

```
let a = new String(); // Dichiarare a come un oggetto String
let b = new Number(); // Dichiarare b come un oggetto Number
let c = new Boolean(); // Dichiarare c come oggetto Booleano
```

Cerchiamo di evitare gli oggetti *String*, *Number* e *Boolean*. Complicano il codice e rallentano la velocità di esecuzione.

Oggetti e funzioni

Uno dei motivi per cui le funzioni sono così rilevanti in JavaScript è che esse sono unità di esecuzione modulari primarie. Ad eccezione del codice JavaScript globale, eseguito nella fase di creazione della pagina, lo script che scriveremo sarà all'interno di una funzione.

Poiché la maggior parte del nostro codice verrà eseguito come risultato di una chiamata di funzione, vedrai che avere funzioni che sono costrutti versatili e potenti che ci offrono una grande flessibilità e influenza durante la scrittura del codice. Sono definite oggetti di prima classe. Ma cosa vuol dire? significa che, proprio come qualsiasi altra variabile, una funzione è un'istanza del tipo di oggetto, può avere proprietà e ha un collegamento al suo metodo di costruzione. Inoltre:

- Possiamo memorizzare una funzione in una variabile.
- Possiamo passare una funzione come parametro a un'altra funzione.
- Possiamo restituire una funzione da una funzione.

Proprio per questo diverse sezioni di questo libro spiegano proprio come la natura delle funzioni come oggetti di prima classe possa essere sfruttata a nostro grande vantaggio. Ma prima, diamo un'occhiata ad alcune delle azioni che possiamo intraprendere con gli oggetti. In JavaScript, gli oggetti godono di determinate funzionalità:

Possono essere creati tramite letterali: `{}` o essere assegnati a variabili, voci di insiemi (*vettori* o *array*) e proprietà di altri oggetti:

```
let gatto(dichiara una variabile)
gatto = {};           (assegna un oggetto ad una variabile)
gatto.push({});       (aggiunge un oggetto a un vettore o array)
gatto.data = {};      (assegna un oggetto alla proprietà di un'altro)
```

Possono diventare parti integranti delle funzioni:

```
function sfamare(gatto) {gatto.sazio = false;
}
sfamare({});
```

Possono essere restituiti come valori dalle funzioni:

```
function returnGatto() {return {};}
}
```

Possono possedere proprietà create e assegnate dinamicamente:

```
let gatto = {}; gatto.nome = "Briciola";
```

Da notare che, a differenza di molti altri linguaggi di programmazione, in JavaScript possiamo fare quasi le stesse cose anche con le funzioni.

La versatilità delle funzioni

Da questo paragrafo in poi insisteremo molto su uno dei cardini del linguaggio Javascript: le funzioni possiedono tutte le capacità degli oggetti software, e quindi potremo trattarle come tali. La definizione esatta è *oggetto di prima classe*, ossia in grado di svolgere tutti i compiti che possono eseguire gli oggetti, cioè: essere memorizzati in una variabile, restituito da un'altra funzione, archiviato come dato, passare come argomento di una funzione. Vediamo alcuni esempi, qui creiamo una funzione attraverso la sua sintassi base:

```
function prova() {}
```

Oppure assegnate a variabili, voci di insiemi e proprietà di altri oggetti:

```
let prova = function() {}; // assegna una funzione ad una variabile
prova.push(function() {}); // aggiunge una funzione ad un vettore o array
prova.data = function() {}; // assegna la proprietà di un oggetto a una funzione
```

Utilizzate come argomenti di altre funzioni:

```
function Prodotti(nome, prezzo) {
    this.nome = nome;
    this.prezzo = prezzo;
}
function Vino(nome, prezzo) {
    Prodotti.call(this, nome, prezzo);
    this.category = 'vino';
}
function Birra(nome, prezzo) {
    Prodotti.call(this, nome, prezzo);
    this.category = 'birra';
}
const rosso = new Vino('chianti', 10);
const bionda = new Birra('peroni', 2);
```

Restituite come valori dalle funzioni:

```
function prova() {return function() {};}
}
```


Possono possedere proprietà che possono essere create e assegnate dinamicamente:

```
let prova = function() {};  
prova.name = "Mario";
```

Insomma, come già accennato all'inizio, tutto ciò che potremo fare con gli oggetti, saremo in grado di farlo anche con le funzioni. Il grande vantaggio è che ovviamente potranno anche eseguire tutte le altre azioni che le contraddistinguono, come la capacità di essere invocabili: potremo sempre chiamarle o invocarle per eseguire un'azione.

Un'altra delle caratteristiche degli oggetti di prima classe, come abbiamo visto, è che possono essere passati alle funzioni come argomenti. Nel caso delle funzioni, ciò significa che passiamo una funzione come argomento a un'altra funzione che potrebbe, in un momento successivo dell'esecuzione dell'applicazione, chiamare la funzione passata. Questo è un esempio di un concetto più generale noto come funzione di callback. Procediamo nel nostro studio cercando di comprendere appieno questa tecnica prima di procedere oltre.

I callback

In un'applicazione, per ovvi motivi legati alla funzionalità, quando impostiamo una funzione da chiamare in un secondo momento, sia dal browser in fase di gestione degli eventi, che da un altro ambito del nostro codice, stiamo impostando una *richiamata*. Il termine deriva dal fatto che stiamo stabilendo una funzione che un altro codice successivamente "richiamerà" in un punto di esecuzione appropriato.

Perché mai avremo bisogno di questa funzionalità? capiterà sicuramente il caso in cui avremo l'esigenza di eseguire del codice dopo che è successo qualcosa che non sia in una sequenza logica temporale, ad esempio dopo il click su un pulsante piuttosto che da qualche altra parte. Da notare che stiamo anticipando la programmazione asincrona. Comunque i callback ci aiutano proprio a non eseguire una funzione prima che un'attività sia stata completata e sono una parte essenziale dell'utilizzo efficace di JavaScript, qui esamineremo come utilizzarli al meglio, vediamo un esempio:

```
const testoRitardato = function() {
```

```
console.log("Sono passati 2 secondi");
}
setTimeout(testoRitardato, 2000);
```

Nel caso specifico *setTimeout* ci impone di attendere due secondi prima di richiamare la funzione *testoRitardato*, ecco il nostro primo callback.

Per capire meglio continueremo con un esempio di una funzione che accetta un riferimento a un'altra funzione come parametro e chiama quella funzione come callback:

```
function diRichiamo(ilCallback) {return ilCallback ();}
```

Per quanto a prima vista questa funzione sia priva di senso, e che per molti non sia neanche un vero callback, ve la mostriamo come esemplificazione estrema e per comprenderne alcuni meccanismi apparentemente evidenti, come la capacità di passare una funzione come argomento a un'altra funzione e di invocare successivamente quella funzione attraverso il parametro passato. Possiamo testare questa funzione con il codice nell'esempio seguente:

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

function diRichiamo(ilCallback){
    let pranzo = parseInt(prompt('Quanti gamberi mangia il gatto Briciola a pranzo?'));
    let cena = parseInt(prompt('Quanti gamberi mangia il gatto Briciola a cena?'));
    alert(ilCallback(pranzo,cena));
}

diRichiamo(totpasti);
```

In questo esempio abbiamo visto il nostro *callback* in un ambito interno al codice, ma ovviamente essi possono essere richiamati anche dal browser:

```
document.querySelector("#PulsanteCallback").addEventListener(
"click", function() {
    console.log("Qualcuno ha pigiato il pulsante");
});
```

Questa è anche una funzione di callback, definita come gestore di eventi per il click dei pulsanti del mouse. L'evento verrà chiamato dal browser quando

si verificherà la condizione, ossia il click del mouse.

Proseguiamo oltre definendo la nozione di dati e operatori ad essi legati, per parlare poi della loro disposizione e raccolta.

Operatori e conversione

Per lavorare con i dati avremo bisogno di utilizzare gli operatori matematici e la conversione. I primi li conosciamo bene, sono i classici +, -, / e * con l'aggiunta di % che in una divisione intera restituisce il resto tra il dividendo e il divisore. Vediamo un esempio:

```
let Fa = prompt("Immettere la temperatura in gradi Fahrenheit, la  
covertirò in gradi Celsius",90);  
let Ce = (Fa - 32) * 5/9;
```

```
alert(Fa + "° Farhenheit equivalgono a " + Ce + "° Celsisus");
```

Ovviamente anche qui valgono le regole dell'algebra, quindi verranno eseguite prima moltiplicazioni e divisioni, poi addizioni e sottrazioni.

Nell'esempio di sopra noteremo che *alert* riporta due numeri, cioè i gradi Celsius e Farhenheit. Per la precisione, di fatto ciò che *alert* riporta è una stringa convertita con i valori richiesti.

L'espressione $(Fa - 32) * 5/9$ è stata semplicemente cercata sul motore di ricerca e riportata nel codice. Più avanti nel libro esploreremo anche l'uso delle espressioni regolari, un altro potente strumento a nostra disposizione per migliorare l'utilizzo di Javascript.

Raccolta e disposizione di dati

Quando si programma un'applicazione, tra le varie problematiche sicuramente avremo quella inerente alla raccolta di un certo numero di dati (vedere il capitolo 5 per approfondire l'argomento sugli insiemi e ripassare quanto trattato qui), e della loro successiva disposizione secondo un ordine definito. Questi potrebbero essere un insieme (*vettore* o *array*) di numeri in ordine casuale, delle lettere oppure delle stringhe: 3, 2, 5, 0, 4, 9, r, a, o, z, Mario, Gino. Questo ordine potrebbe andare bene, ma è probabile che, prima o poi, dovremmo riorganizzarlo secondo un ordine logico e sappiamo che non è semplice creare del codice che faccia questo per noi. Certo, potremo trovarlo su internet, ma è probabile che dovremo poi adattarlo. Proprio per questa esigenza esiste il metodo *sort* il quale, definendo alcuni paramentri, è in grado confrontare e ordinare determinati valori. Nel prossimo listato

osserveremo come funziona *sort*. In questo caso specifico lo abbineremo alla funzione *comparaN*, la quale effettuerà un semplice confronto per poi ordinare gli elementi assegnati. Se *comparaN* viene omesso tutti gli elementi saranno convertiti in stringhe e organizzati secondo la codifica *Unicode code point*, dove, riassumendo all'osso, ogni lettera e numero hanno una loro posizione in una tabella per questioni di uniformità e compatibilità con tutte le lingue. Quando verrà richiesto di mettere in ordine i numeri la logica della sequenza sarà differente perché non seguirà l'ordine di grandezza ma la posizione nella tabella, quindi: 200, 90, 1, 7 sarà: 1, 200, 7, 90:

```
let calciatori = ['Totti', 'Maradona', 'Platini'];
calciatori.sort(); // ['Maradona', 'Platini', 'Totti']

let goals = [1, 11, 25, 2];
goals.sort(); // [1, 11, 2, 25]

let cose = ['word', 'Word', '1 Word', '2 Words'];
cose.sort(); // ['1 Word', '2 Words', 'Word', 'word']

let strVettore = ['Maldini', 'Gullit', 'Rossi'];
let strNumeri = ['80', '9', '700'];
let numVettore = [40, 1, 5, 200];
let variNumeri = ['80', '9', '700', 40, 1, 5, 200];

function comparaN(a, b) {
    return a - b;
}

console.log('Vettore:', strVettore.join());
console.log('Ordinato:', strVettore.slice().sort());
console.log('Vettore di numeri:', strNumeri.join());
console.log('Ordinato senza compara:', strNumeri.slice().sort());
console.log('Ordinato          con          compara:',
strNumeri.slice().sort(comparaN));
console.log('Array con stringa di numeri:', strNumeri.join());
console.log('Ordinato senza compara:', variNumeri.slice().sort());
console.log('Ordinato          con          compara:',
variNumeri.slice().sort(comparaN));
```

Usando *slice()* prima di *sort()* creiamo una copia dell'array originale senza modificarlo direttamente. Abbiamo anche visto come, utilizzando *sort* e confrontando i vari numeri tramite un callback si riesce ad ordinare correttamente l'insieme: `function comparaN(a, b){return a - b;}`. Questa funzione, per evitare di ripeterla ogni volta l'abbiamo trattata come un

oggetto, passandola come argomento a un metodo, che può accettarlo come parametro. Questa modalità operativa ci apre davanti un mondo di possibilità, infatti la capacità di utilizzare le funzioni come oggetti ci permetterà di usufruire di alcuni vantaggi attribuiti solitamente solo agli oggetti, come ad esempio associare proprietà alle funzioni:

```
let gatto = {};  
gatto.nome = "Briciola";  
let ciotola = function(stipendio, spesa, tempo) {};  
ciotola.contenuto = "Croccantini";
```

Oltre a questo avremo accesso alle proprietà *name* e *length* e alla possibilità di usufruire di una particolare modalità legata alle espressioni di funzione utilizzate in determinati contesti chiusi usando *func*, parliamo quindi della NFE (*Named Function Expression*, espressioni di funzione con nome). Eccoli in azione:

```
function uno() {  
  console.log("saluti da Mario");  
}  
function due(x) {}  
function tre(x, y) {}  
  
console.log(uno.name); // uno  
console.log(uno.length); // 0  
console.log(due.length); // 1  
console.log(tre.length); // 2
```

Ecco invece *func* al lavoro:

```
let prova = function func(qualcosa) {  
  if (qualcosa) {  
    console.log(`Saluti ${qualcosa}`);  
  } else {  
    func("da Mario");  
  }  
};  
let mario = prova;  
prova = null;  
mario();
```

Qui abbiamo aggiunto un nome ad una espressione di funzione che verrà utilizzato solo all'interno della funzione stessa. Questa evenienza non è possibile estenderla alle dichiarazioni di funzione.

La possibilità di considerare oggetti le funzioni ci permetterà anche di memorizzarne una raccolta per gestire alcuni parametri, come ad esempio, i *callback* (di cui abbiamo parlato in un paragrafo precedente) da invocare quando si verifica un evento.

Esiste anche la cosiddetta “*memoizzazione*” (volutamente senza la *r*) che consente alla funzione di ricordare i valori calcolati in precedenza, per migliorare la fluidità del codice in caso di ulteriori chiamate.

Sfruttare gli oggetti funzione

Come abbiamo affermato prima, per ottimizzare la gestione dei *callback* dato un certo numero di funzioni, trattandole come oggetti software avremo la possibilità di raccoglierle in vettori. A quel punto incontreremo alcune problematiche tecniche, come determinare quali funzioni sono nuove e dovrebbero essere aggiunte alla raccolta e quali sono già residenti e non dovrebbero farne parte. Questo perché generalmente, nella gestione dei *callback*, la presenza di una o più copie della stessa funzione provocherebbe una gran confusione nel momento della chiamata.

Potremo istintivamente pensare di memorizzare tutte le funzioni in un vettore e poi scorrerlo, controllando l’eventuale presenza di doppi. Purtroppo per noi questa tecnica non è assolutamente valida. Disponiamo di altre frecce per il nostro arco, difatti potremo usare le proprietà delle funzioni per ottenere ciò che abbiamo esposto inizialmente senza correre il rischio di scrivere del codice che potremo definire eufemisticamente problematico:

```
let uno = {y: 1, z:{}},
add: function(x) {if (!x.y) {
  x.y = this.y ++;
  this.z[x.y] = x;
  return true;
}
}
};
function due() {}
uno.add(due); // true
```

In questo listato, creiamo un oggetto e lo assegniamo alla variabile *uno*, in cui, nel corso dell’esecuzione, archiveremo una raccolta di funzioni. Il nostro oggetto chiamato *uno* ha due proprietà: la prima ricorda il valore *y*, nell’altra verranno memorizzate le funzioni tramite il metodo *add()*.

All'interno di *add*, controlliamo prima se la funzione è già stata aggiunta alla raccolta, cercando l'esistenza di *y*. Se la variabile dovesse palesarsi significherà che la funzione è già stata elaborata, di conseguenza non verrà considerata. In caso contrario le verrà assegnato il valore *y* che aumenterà ad ogni passaggio e infine, utilizzando *add*, verrà memorizzata con il valore *y* come nome della proprietà. Otterremo *true* quando la funzione verrà aggiunta attraverso il metodo *add()*.

Possiamo ripassare il capitolo dedicato agli insiemi cercando di approfondire ulteriormente l'argomento ed eventualmente utilizzare anche altri approcci.

Siamo così giunti alla *memoizzazione* la quale altro non sarebbe che il processo di memorizzazione dei valori restituiti da una funzione per averli a disposizione al bisogno senza ricalcolarli. Questo incrementerà considerevolmente le prestazioni evitando inutili calcoli complessi già svolti. Ciò essenzialmente è utile ad esempio quando si eseguono calcoli per le animazioni o per complesse operazioni matematiche.

Proviamo con un listato a testarne l'uso:

```
const val = (x) => (x + 5);
console.log('Risultato ottenuto direttamente', val(5));
const mem = (a) => {
  let ch = {};
  return (...altro) => {
    let x = altro[0];
    if (x in ch) {
      console.log('Risultato ottenuto recuperando x, cioè 5, dalla
nostra memoria cache e aggiungendo 1');
      return ch[x]+1;
    }
    else {
      console.log('Risultato ottenuto sostituendo il valore presente
in memoria con r, che è sempre 5');
      let r = a(x);
      ch[x] = r;
      return r;
    }
  }
}
const memVal = mem(val);

console.log(memVal(2)); // 5 + 2 = 7
console.log(memVal(2)); // 5 + 2 + 1 = 8
console.log(memVal(9)); // 5 + 9 = 14
console.log(memVal(9)); // 5 + 9 + 1 = 15
```

A una prima occhiata la funzione non sembra essere troppo diversa dalle altre, anche se con questa istruzione `let ch = {}` creiamo una memoria cache in cui salveremo un numero da sommare in seguito. La nostra cache è una proprietà della funzione stessa, quindi è mantenuta in vita finché la funzione è viva.

Questo approccio appena visto presenta indubbiamente alcuni vantaggi:

L'utente finale godrà di benefici inerenti alle prestazioni e il tutto funzionerà senza che se ne accorga, non servirà che qualcuno debba eseguire qualche richiesta particolare o effettuare ulteriori azioni.

C'è un rovescio della medaglia: qualsiasi tipo di memorizzazione nella cache sacrificherà sicuramente la memoria a favore delle prestazioni le quali dipenderanno anche da altri fattori non immediatamente definibili.

Sicuramente con l'esperienza saremo in grado di valutare tutte queste problematiche, e potremo sempre rivolgerci alla immensa community di Javascript, sempre attiva e pronta ad aiutare.

Andiamo avanti con le funzioni e la differenza tra le varie tipologie di esse.

Differenze tra le varie tipologie di funzioni

Come abbiamo già scritto la funzione non è altro che un blocco di codice che dovrà poi eseguire una determinata attività o che restituirà un valore e, se impiegate come oggetti tra le varie possibilità, sono riutilizzabili, proprio come un valore o un numero. In definitiva JavaScript fornisce diversi modi per definire le funzioni, vediamoli tutti qui:

- *Funzione dichiarativa*, il modo più comune per utilizzarla, si identifica con la parola chiave "*function*" seguito da un nome da attribuire:

```
function prova() {  
    alert('Buongiorno');  
}
```

- *Funzione espressione*, molto usata, come la prima, qui il nome non è obbligatorio ma necessita della parola chiave "*function*":

```
let prova = function() {  
    alert("Buonasera");  
};
```

La variabile *prova* ottiene come valore la funzione che ci esporrà il saluto.

- *Funzione IIFE* (Immediately Invoked Function Expression, tradotto funzione anonima autoeseguita o funzione immediata), usata poco dopo che l'ES6 ha introdotto i nuovi tipi di variabile (come *const* e *let*) e i moduli, è composta da una funzione anonima, dove le variabili non sono accessibili dall'esterno di essa, e da una chiusura:

```
(function() {})(45); oppure (function() {} (45))
```

- *Funzione freccia*, nuova sintassi introdotta dall'ECMAScript 6 in cui non serve né il nome né la parola chiave "*function*", sostituita appunto dalla freccia `=>` e dai parametri racchiusi nelle parentesi:

```
x => {return x + x}
```

- *Funzione costruttore*, tramite il costruttore *new function* da una stringa verrà creata una funzione seguita dai classici parametri nelle parentesi tonde:

```
new function('x', 'y', 'return x - y')
```

Nell'esempio viene creata una funzione la quale fornisce la differenza tra i due valori *x* e *y*.

- *Funzione come metodo di un oggetto*, quando una proprietà di un'oggetto è una funzione prende il nome di metodo:

```
let oggetto = {  
  nome: "Mario",  
  cognome: "Rossi",  
  nomeCognome: function () {  
    return this.nome+ " " + this.cognome;  
  }  
}  
oggetto.nomeCognome(); // Mario Rossi
```

L'oggetto "*oggetto*" ha due proprietà (*nome* e *cognome*) e un metodo (*nomeCognome*) che è una funzione.

- *Funzione asincrona*, introdotta con l'ECMAScript 8, tramite *async* e *await* potremo gestire funzioni asincrone tramite un approccio sincrono, cioè verranno eseguite dopo che un evento predeterminato, affiancandosi alle *promesse* (che vedremo nel capitolo 8).

- *Funzione generatrice*, funzione molto particolare introdotta con l'ECMAScript 6 le quali, al bisogno attraverso un loop (ciclo o iterazione) possono restituire una serie di valori. Si usano assieme a *yield* e *next* con la sintassi *function **. Possono generare dichiarazioni di funzioni, espressioni di funzioni e costruttori di funzioni (ma non è un costruttore):

```
function* prova(){
  yield 0;
  yield 3167;
}

let x = prova();
console.log(x.next().value); // 0
console.log(x.next().value); // 3167
console.log(x.next().value); // undefined
```

Dedichiamo adesso una breve parentesi al costruttore di funzioni *Function*. Non impedisce chiusure ai contesti di creazione e viene sempre creato nell'ambito globale. Durante l'esecuzione, potrà accedere solo alle variabili locali e globali, non a quelle dell'ambito in cui è stato creato il costruttore, vediamo qui sotto un esempio proprio di questa specifica:

```
let a = 1;
function creaFunzione() {
  let a = 5;
  return new Function('return a;');
}
let b = creaFunzione();

console.log(b()); // Il risultato è 1, non 5
```

Non importa se ancora non è chiarissimo questo esempio, più avanti ci servirà.

Dopo aver notato le diverse tipologie delle funzioni insistiamo su come sia molto importante comprenderle e, forse non c'è neanche bisogno di sottolineare che le differenze non sono solo formali ma bensì *sostanziali*.

In un esempio di due pagine precedenti, abbiamo mostrato la classica funzione, nel caso specifico era una funzione immediata, la quale era racchiusa tra parentesi. Per quale strano motivo è stato fatto ciò? In realtà è molto semplice, se tralasciassimo le parentesi attorno all'espressione della funzione e mettessimo la nostra chiamata immediata come un'istruzione separata: *function() {}* (1), il parser JavaScript inizierà a elaborarlo e

concluderà, perché è un'istruzione separata che inizia con la parola chiave `funzione`, che si tratta di una dichiarazione di funzione. Poiché ogni dichiarazione di funzione deve avere un nome (e qui non ne abbiamo specificato uno), verrà generato un errore. Per evitare ciò, inseriamo l'espressione della funzione tra parentesi, segnalando al parser JavaScript che si tratta di un'espressione e non di un'istruzione.

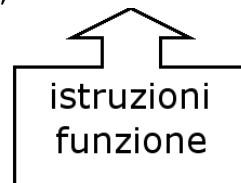
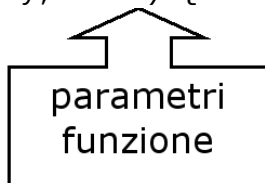
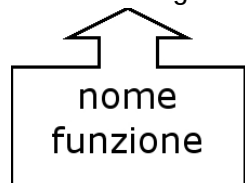
Per raggiungere il medesimo obiettivo esiste un modo alternativo ancora più semplice, anche se usato di rado: `(function() {} (1))`. Racchiudendo la definizione della funzione immediata e la chiamata tra parentesi, è anche possibile notificare al parser JavaScript che ha a che fare con un'espressione. Andiamo avanti, sperimentando le due modalità generalmente utilizzate per definire le funzioni in JavaScript che sono l'uso di dichiarazioni di funzione e le espressioni di funzione.

La funzione dichiarativa

Come abbiamo affermato poco fa la funzione dichiarativa o la dichiarazione di funzione è il modo più semplice ed utilizzato per definire una funzione in JavaScript. Dopo la parola chiave `function` vi sarà un nome di funzione obbligatorio e un elenco di nomi di parametri separati da virgole opzionali racchiusi tra parentesi (obbligatorie). Il corpo della funzione, che è un elenco di istruzioni potenzialmente vuoto, deve essere racchiuso tra parentesi graffe di apertura e chiusura. Oltre a questa forma, che ogni dichiarazione di funzione deve soddisfare, c'è un'altra condizione: una dichiarazione di funzione in JavaScript deve essere inserita da sola, come un'istruzione separata contenuta in un'altra funzione o un blocco di codice.

Vediamo degli esempi chiarificatori:

```
function gatti(Kitty, Coda) {alert("I gatti osservano");
```



Potremo anche dichiarare una funzione all'interno di un'altra:

```
function gatti(Kitty, Coda) {function topoNascosto(Squit) {
```

```
alert("I gatti hanno sentito qualcosa");  
}  
  
return topoNascosto(Squit);  
}
```

Approfondiremo nel capitolo 7 questo argomento leggermente più complesso di quello che potrebbe sembrare a prima vista.

La funzione espressione

Oltre alla sintassi appena indicata esiste un'altra modalità denominata “funzione espressione”, essenzialmente perché, come vedremo, è una vera e propria operazione basica di uguaglianza, e quindi un’espressione:

```
let gatti = function(Kitty, Coda) {alert("I gatti osservano");  
}
```

Qualcuno potrebbe chiedersi il perché di questa differenziazione. Esistono diverse motivazioni che vedremo passo dopo passo nel proseguio dello studio, intanto possiamo sicuramente affermare che le funzioni espressione ci permetteranno di definirne l’esecuzione esattamente dove ne avremo bisogno, rendendo il nostro codice anche più facile da capire.

Guardiamo un altro esempio che mostra le differenze tra le funzioni dichiarate e le espressioni:

```
function salutiamo() {  
    alert("Buongiorno"); // funzione dichiarata  
}  
let espressione = function() {  
    alert("Arrivederci"); // funzione espressione  
}  
;  
let x = salutiamo; // possiamo anche copiare la funzione  
  
x();  
salutiamo();  
espressione();
```

Ritorniamo sulle differenze, abbiamo già visto come le dichiarazioni di funzione sono istruzioni separate di codice, e può essere anche contenuto nel corpo di altre funzioni.

Invece le funzioni espressione fanno sempre parte di un'altra affermazione. Sono posizionate sempre sul lato destro di un incarico o una dichiarazione di variabile:

```
let prova = function() {};
```

O come argomento per un'altra chiamata di funzione o come valore di ritorno di una funzione:

```
prova(function() {return function() {};  
});
```

Ma la differenza più sostanziale riguarda un parametro, il nome della funzione è obbligatorio per le dichiarazioni di funzione, mentre al contrario, per le espressioni di funzione è opzionale, questo perché una delle ragioni dell'esistenza stessa di una funzione è il fatto che deve essere invocabile, e quindi l'unico modo per fare riferimento ad essa è attraverso il suo nome.

Invece Le espressioni di funzione, sono parte di altre espressioni, quindi abbiamo modi alternativi per invocarle. Ad esempio, se un'espressione di funzione è assegnata a una variabile, possiamo usare quella variabile per invocarla:

```
let leggereLibro = function(sfogliare, pagine) {};  
leggereLibro(sfogliare, pagine);
```

O, se è un argomento di un'altra funzione, potremo invocarlo all'interno di quella funzione tramite il nome del parametro corrispondente:

```
function leggereLibro(qualcosa) {qualcosa();  
}
```

Le funzioni immediate

Nel nostro listato potrebbe capitare di avere diverse variabili con scope globale assieme a tante funzioni. In questo caso, per evitare di intasare il nostro codice, Javascript ci mette a disposizione le cosiddette *funzioni IIFE* (ossia funzioni immediatamente invocate, funzioni immediate).

Questa diversa tipologia di funzione ha alcune peculiarità: non avremo bisogno di assegnarle alcun nome e potrà essere racchiusa all'interno delle parentesi tonde, nel caso avremo bisogno di mantenere private le variabili. Inoltre ci consentiranno di imitare i moduli. Vediamo le sintassi:

```
(function() {})( );  
(function() {} )( );  
(( ) => {})( );
```

Oppure potremo anche invocare in questi modi:

```
+function() {} ( );  
-function() {} ( );  
!function() {} ( );  
~function() {} ( );
```

In tale caso, invece di usare le parentesi attorno alle espressioni di funzione, utilizzeremo i cosiddetti “*operatori unari*”: +, -, ! E ~, per differenziarle dalle dichiarazioni di funzione standard. Lo faremo proprio per segnalare al motore JavaScript che si tratterà di espressioni e non di istruzioni. Vediamone un esempio:

```
const depositare = (saldo) => ((copiaSaldo) => {  
  let saldo = copiaSaldo; // questa IIFE sarà privata  
  const vediSaldo = () => {  
    console.log('troppi soldi nel conto, spendiamoli!');  
  };  
  vediSaldo();  
  return {  
    prelievo(cifra) {  
      if (saldo >= cifra) {  
        saldo -= cifra;  
        return saldo;  
      }  
      return 'Non c'è abbastanza denaro sul conto';  
    },  
  };  
})(saldo);
```

```
const conto1 = depositare(500); // depositiamo 500  
console.log(conto1.saldo); // il saldo è privato  
console.log(conto1.vediSaldo); // non possiamo sbirciare  
console.log(conto1.prelievo(100)); // restano 400  
console.log(conto1.prelievo(350)); // restano 50
```

```
const conto2 = depositare(800); // qui depositiamo 800  
console.log(conto2.prelievo(900)); // richiesta troppo alta!
```

Dopo aver imparato i dettagli delle due modalità per definire le funzioni in JavaScript esploriamo un'altra possibilità per lavorarci che abbiamo già notato in alcuni esempi già visti, tra cui quello in alto: le *funzioni freccia*.

Le funzioni freccia

Poiché in JavaScript useremo moltissimo le funzioni, in questo paragrafo vedremo un metodo che ci consentirà di crearle più velocemente e oltretutto anche con una sintassi molto ristretta. Tutto questo ci sarà di aiuto per la stesura del codice in genere. Stiamo parlando delle cosiddette funzioni freccia, così nominate per la loro esplicitazione simbolica uguale ad una freccia, cioè `=>`.

Esemplificando al massimo, potremo affermare che le funzioni freccia sono una sorta di clone delle espressioni di funzione.

Riproponiamo un esempio di ordinamento già visto precedentemente per reinterpretarlo con questa nuova metodologia:

```
let numeri = [0, 2, 3, 4, 9, 1];
numeri.sort(function (n1, n2) {
  return n1 - n2;
});
```

Questo esempio utilizza un'espressione di funzione di callback inviata al metodo di ordinamento dell'oggetto array; questo callback verrà invocato dal motore JavaScript per ordinare i valori di matrice in ordine decrescente.

Ora vediamo come fare esattamente la stessa cosa con le funzioni freccia:

```
let numeri = [0, 2, 3, 4, 9, 1];
numeri.sort((n1, n2) => n1 - n2);
```

Indubbiamente più breve. Non sussistono eventuali distrazioni causate dalla parola chiave della funzione, dalle parentesi graffe o dall'istruzione *return*. In un modo molto più semplice di quanto possa fare un'espressione di funzione, la funzione freccia afferma: *ecco una funzione che accetta due argomenti e restituisce la loro differenza*. Come abbiamo introdotto all'inizio, per utilizzarla ci serviremo di un nuovo operatore, `=>`, il cosiddetto “*operatore freccia*” (un segno uguale immediatamente seguito da un segno maggiore), che è alla base della definizione di una funzione freccia, la sintassi precisa prevederà prima un parametro e dopo un'espressione, in questo modo: `parametro => espressione`

Vediamola di nuovo all'opera:

```
let chiamare = nome => "Dove sei" + nome;
assert(chiamare("Briciola") === "Dove sei Briciola"); // true
let chiamareAltri = function(nome) {return "Dove sei" + nome;
};
```

```
assert(chiamareAltri("Kitty") === "Dove sei Kitty"); // true
```

Si nota subito che il codice è più ristretto rispetto alla seconda parte. Questa è la versione più semplice della sintassi della funzione freccia, ma in generale, la funzione freccia può essere definita in due modi.

La definizione della funzione freccia inizia con un elenco opzionale di nomi di parametri separati da virgole. Se non ci sono parametri, o più di un parametro, questo elenco deve essere racchiuso tra parentesi. Ma se abbiamo un solo parametro, le parentesi sono opzionali. Questo elenco di parametri è seguito da un codice freccia obbligatorio, che indica a noi e al motore JavaScript che abbiamo a che fare con una funzione freccia.

Dopo l'operatore freccia, abbiamo due opzioni. Se è una funzione semplice, inseriamo un'espressione (un'operazione matematica, un'altra invocazione di funzione, qualunque cosa), e il risultato dell'invocazione di funzione sarà il valore di quell'espressione. Ad esempio:

```
let guadagno = denaro => "Oggi hai guadagnato" + denaro;
```

Se il corpo della funzione freccia è un'espressione, il valore restituito dalla funzione sarà il valore di quell'espressione.

```
(parametro1, parametro2) =>
```

In altri casi, quando le nostre funzioni freccia non sono così semplici e richiedono più codice, possiamo includere un blocco di questo dopo l'operatore freccia. Per esempio:

```
let mangiare = cibo => {  
  let cosaMangiare = 'Mangeremo';  
  return cosaMangiare + cibo;  
};
```

Qui, il valore di ritorno della funzione freccia si comporta come in una funzione standard. Se non c'è un'istruzione *return*, il risultato dell'evocazione della funzione sarà indefinito, e, se c'è, il risultato sarà il valore dell'espressione di ritorno.

Le funzioni freccia, come tutte le altre funzioni, possono ricevere argomenti da utilizzare per eseguire il loro compito.

Vediamo adesso cosa succede con questi valori che passiamo alla funzione.

Distinguiamo tra parametri ed argomenti

In questa sezione ci occuperemo di definire precisamente la differenza tra i *parametri* e gli *argomenti* della funzione, cioè quei valori che saranno inseriti rispettivamente all'interno delle parentesi tonde e di quelle graffe subito dopo. Il “parametro” sarà una variabile che elencheremo come parte della definizione di una funzione all'interno delle parentesi tonde:

```
function sfamare(gatto) {  
  return prepCena(gatto, "cibo");  
}
```

L'argomento è un valore che passeremo alla funzione quando la invocheremo, come *gatto* e *cibo* qui sopra.

```
let cosaMangiare = function(animale, mangiare) {  
  return animale + " - " + mangiare;  
};  
let pranzo = ciotola => cosaMangiare(ciotola, "quantità");  
fame("Briciola");  
pranzo("Croccantini");
```

Gli esempi appena osservati ci mostrano come la definizione della funzione includa la specifica dei parametri, cosa che ovviamente tutti i tipi di funzioni potranno avere. Andando nello specifico, abbiamo notato che i parametri si trovavano:

Nella *dichiarazione di funzione* (il parametro *gatto* e la funzione *sfamare*), nell'*espressione di funzione* (i parametri *animale* e *mangiare* e la funzione *cosaMangiare*), e, infine nella *funzione freccia* (il parametro *ciotola*).

Gli argomenti invece sono collegati all'invocazione di una funzione, essenzialmente sono quindi valori passati ad essa al momento della sua invocazione, anche questi li abbiamo visti:

Nella stringa *Briciola* che viene passata come argomento della funzione *fame*, nella stringa *Croccantini* che viene passata come argomento della funzione *pranzo*, ed infine nel parametro *gatto* della funzione *fame* che viene passata come argomento della funzione *cosaMangiare*.

Ricordiamoci sempre che quando un elenco di argomenti viene fornito come parte di una chiamata di funzione, questo verrà attribuito ai parametri nell'ordine specificato, quindi il primo argomento sarà assegnato al primo parametro, il secondo al secondo parametro e così via. In caso sussista un numero di argomenti superiore rispetto ai parametri, quelli in eccesso

semplicemente non saranno assegnati. Testiamolo sul campo, come appena detto una funzione può essere chiamata con qualunque numero di argomenti:

```
function somma(a, b) {  
    return a + b;  
}  
  
alert(somma(1, 2, 3, 4, 5) );
```

In questo caso non ci saranno errori a causa di troppi argomenti, ma ovviamente nel risultato verranno conteggiati solo i primi due.

Abbiamo assodato questo concetto, e, già che ci siamo, possiamo ricollegarci al discorso inerente agli argomenti utilizzando lo stesso esempio (con qualche piccola variazione) per parlare del parametro *rest*. Ci servirà per includere in un vettore tutti gli argomenti designati. La sintassi prevede l'inclusione nella definizione della funzione utilizzando tre punti ... seguiti dal nome dell'array che li conterrà. I punti significano letteralmente "raccolgi i parametri rimanenti in un insieme":

```
function sommaTutto(...argomenti) {  
    let somma = 0;  
    for (let arg of argomenti) somma += arg;  
    return somma;  
}  
  
alert(sommaTutto(1) );  
alert(sommaTutto(1, 5, 8) );
```

Potremmo anche scegliere di ottenere i primi parametri come variabili e raccogliere solo il resto. Qui i primi due argomenti vanno nelle variabili e il resto nell'array dei titoli:

```
function mostraRazze(tipo, nome, ...razza) {  
    alert(tipo + ' ' + nome );  
    alert(razza[0] );  
    alert(razza[1] );  
    alert(razza.length );  
}  
  
mostraRazze("Gatto", "Briciola", "Persiano", "Pelo lungo");
```

Utilizzare l'argomento della funzione

Abbiamo appena visto che l'argomento è un valore che passiamo alla funzione quando la invochiamo. Utilizzando il parametro *arguments* avremo a che fare con l'insieme (vettore o array) di tutti gli argomenti passati in una funzione, vediamo come scrivere una funzione nella maniera standard, indicando i parametri:

```
function contoRistorante(primi, secondi, dolce) {  
    alert("Conto: " + primi + ", " + secondi + ", " + dolce);  
}
```

oppure usando *arguments*, e si comporterà allo stesso modo anche se in questo caso non definiremo nessun argomento:

```
function contoRistorante() {  
    alert("Conto: " + arguments[0] + ", " + arguments[1] + ", " +  
arguments[2]);  
}
```

arguments è un oggetto simile ad un insieme (vettore o array) anche se gli argomenti avranno una lunghezza e le proprietà indicizzate da zero, ma non i metodi integrati dei vettori come *forEach()* o *map()*.

Arguments sarà accessibile all'interno di funzioni che contengono i valori degli argomenti passati a quella funzione, è una variabile locale disponibile all'interno di tutte le funzioni non freccia. Possiamo fare riferimento agli argomenti all'interno di una funzione usando il suo oggetto *arguments*. Ha voci per ogni argomento con cui è stata chiamata la funzione, con l'indice della prima uguale a 0.

Ad esempio, se a una funzione vengono passati 3 argomenti, è possibile accedervi come segue:

```
arguments[0] // primo argomento  
arguments[1] // secondo argomento
```

Ogni argomento può anche essere impostato o riassegnato:

```
arguments[1] = 'nuovo valore';
```

Come detto l'oggetto argomenti non è un vettore (array). È simile, ma manca di tutte le proprietà dell'array tranne la lunghezza. Tuttavia, può essere convertito in un vettore:

```
let argomenti = Array.prototype.slice.call(blabla);
```

```
let argomenti = [].slice.call(blabla);
```

L'uso di un array letterale è più breve di quanto sopra ma alloca un array vuoto. Come si può fare con qualsiasi oggetto simile ad un array, possiamo utilizzare il metodo *Array.from()* o diffondere la sintassi per convertire gli argomenti in un array reale:

```
let argomenti = Array.from(blabla);
```

```
let argomenti = [...blabla];
```

L'oggetto *arguments* è utile per le chiamate di funzioni con più argomenti. Ha una proprietà denominata *length* che ne indica il numero esatto:

```
x = sommaTutti(5, 10, 15, 20, 25, 30);
```

```
function sommaTutti() {  
  let i;  
  let somma = 0;  
  for (i = 0; i < arguments.length; i++) {  
    somma += arguments[i];  
  }  
  return somma;  
}
```

Qui sopra vediamo delle chiamate legate ad argomenti. Come abbiamo già visto, si potrebbe pensare che questi argomenti siano degli array, infatti, hanno un parametro di lunghezza e le sue voci possono essere recuperate usando la notazione dell'array. Ma come abbiamo detto non è così.

È possibile utilizzare *arguments.length* per contare con quanti argomenti è stata chiamata la funzione. Se invece vogliamo contare quanti parametri una funzione dichiara di accettare, controlleremo la proprietà *length* di quella funzione. Possiamo anche utilizzare l'operatore *typeof* che restituirà 'object' se utilizzato con *arguments*, e potremo anche vedere anche il singolo argomento:

```
console.log(typeof arguments); // 'oggetto'
```

```
console.log(typeof arguments[0]); // restituisce il primo argomento
```

[Usare la sintassi spread con gli argomenti](#)

Come è possibile fare con qualsiasi oggetto array, si può utilizzare il metodo *Array.from()* conosciuto come *spread operator*, ossia un operatore che distribuisce i valori contenuti in un array sugli argomenti della funzione, per convertire argomenti in un array:

```
let argomenti = Array.from(argomenti);  
let argomenti = [...argomenti];
```

Di seguito le proprietà che potremo eseguire:

arguments.callee: riferimento alla funzione attualmente in esecuzione a cui appartengono gli argomenti. Vietato in modalità rigorosa.

arguments.length: il numero di argomenti passati alla funzione.

arguments[@@iterator]: restituisce un nuovo oggetto *array iterator* che contiene i valori per ogni indice negli argomenti.

Prima di *Array.from* si utilizzava *Array.prototype.slice.call* per trasformare qualunque oggetto con le sembianze di un vettore in un insieme.

Parametri rest, default e destructured

L'oggetto *arguments* potrà essere usato assieme ai parametri *rest*, *default*, e *destructured*. Ma a cosa ci riferiamo precisamente? vediamo subito, il parametro *rest* l'abbiamo già studiato, permette ad una funzione di accettare un numero indefinito di argomenti (nota anche come funzione variadica), eccone un esempio:

```
function variadica(...argomenti) {  
  return argomenti;  
}  
variadica(1, 2, 3);
```

L'uso di *...* può essere utilizzato una sola volta e alla fine dell'ultimo argomento. Serve per ridurre il codice utilizzato per la conversione di un insieme di argomenti in un vettore (array).

Per quanto riguarda il parametro di default (o predefinito) normalmente è *undefined*, tuttavia è possibile impostare un valore differente:

```
function moltiplica(x, y) {
    return x * y
}
moltiplica(3)      // NaN !
```

potrà essere cambiato in:

```
function moltiplica(x, y) {
    y = (typeof y !== 'undefined') ? y : 2
    return x * y
}
moltiplica(3) // 6
```

Infine, la destrutturazione (*destructured*) è un'espressione che consente di separare i valori da vettori della variabile di origine o proprietà da oggetti in diverse variabili:

```
let x, y, z;
[x, y] = [1, 2];
console.log(x); // 1
console.log(y); // 2
[x, y, ...z] = [1, 2, 4, 8, 16];
console.log(z); // array [4,8,16]
```

La presenza dei parametri *rest*, *default*, o *destructured* non altera il comportamento dell'oggetto *arguments* nel codice scritto in *strict mode*, comunque c'è una sottile differenza tra modalità *strict* e *non-strict*.

Quando una funzione *non-strict* contiene parametri *rest*, *default*, o *destructured*, allora i valori nell'oggetto *arguments* non tracciano il valore degli argomenti (e vice versa). Al contrario, riflettono gli argomenti forniti al momento dell'invocazione:

```
function prova(z) {
    arguments[0] = 9; // aggiorna arguments[0] e anche z
    console.log(z);
}
prova(10); // 9
```

```
function prova(z) {
    z = 9;
    console.log(arguments[0]);
}
prova(10);
```

arguments è un oggetto simile a un array accessibile all'interno di funzioni che contiene i valori degli argomenti passati a quella funzione.

```
function prova(z = 2) {  
    arguments[0] = 9; // aggiorna arguments[0] ma non z  
    console.log(a);  
}  
prova(10); // 10
```

```
function prova(z = 2) {  
    z = 9;  
    console.log(arguments[0]);  
}  
prova(10); // 10
```

```
function prova(z = 2) {  
    console.log(arguments[0]);  
}  
prova(); // undefined
```

Abbiamo accennato in precedenza che in molti casi possiamo usare il parametro *rest* invece del parametro *arguments*. *Rest* è un vero array, il che significa che possiamo usare tutti i nostri metodi di array preferiti su di esso. Ciò gli conferisce un certo vantaggio rispetto all'oggetto *arguments*. Come esercizio, prova a riscrivere l'elenco sopra utilizzando *rest* invece di *arguments*.

Quindi *arguments* si può utilizzare invocando una funzione con più argomenti di quanti essa dichiara formalmente di accettare.

Si usa *arguments.length* per determinare il numero di argomenti passati alla funzione, e quindi si processa ogni argomento usando l'oggetto *arguments*. Per determinare il numero di parametri presenti nella dichiarazione di una funzione, si usa la proprietà *function.length*.

Assodati questi concetti è ora di procedere oltre, concentriamoci sull'invocazione delle funzioni e ciò che ne concerne.

Evocare le funzioni avvalendosi di this

Quando invocheremo una funzione, oltre ai parametri che conosciamo ne avremo a disposizione un altro denominato *this*. Questo parametro, molto importante in tutti i linguaggi orientati agli oggetti e quindi anche in Javascript, si riferisce a un oggetto associato alla chiamata della funzione, o *contesto della funzione*. Potrebbe capitare per i programmatori che provengono da linguaggi come Java di avere una visione diversa, infatti in

quel contesto *this* di solito punta a un'istanza della classe all'interno della quale è definito il metodo.

Tuttavia, in JavaScript, invocare una funzione come metodo è solo un modo in cui essa può essere invocata. E, come abbiamo detto, ciò a cui punta il paramentro *this* non è definito, come in Java o C #, solo da come e dove la funzione è determinata; può anche essere fortemente influenzato da come viene invocata la funzione. Vedremo quindi, che una delle differenze principali tra i vari modi di invocare le funzioni è come viene determinato il valore di *this*.

Abbiamo anche visto praticamente tutte le possibili modalità per chiamare una funzione, ma finora non sapevamo che il modo in cui una funzione viene invocata ha un enorme impatto sul modo in cui opera il codice al suo interno, principalmente nel modo in cui viene stabilito il contesto della funzione. Questa differenza è molto più importante di quanto potrebbe sembrare a prima vista. Lo esamineremo in questa sezione e lo sfrutteremo nel resto dei prossimi capitoli, ma andiamo con ordine, proviamo a richiamare una funzione con le varie possibilità, precisamente con i quattro diversi modi possibili:

- 1) Direttamente: *function abc()*
- 2) Tramite il costruttore: *new gatto()*
- 3) Come metodo di un oggetto: *function gatto.abc()*
- 4) Tramite la funzione: *abc.call(gatto)* o il metodo *abc.apply(gatto)*

In tutti i casi l'operatore di chiamata della funzione è un insieme di parentesi che seguono qualsiasi espressione che restituisce un riferimento a una funzione, a parte la funzione *call* e il metodo *apply* ovviamente.

Evocare una funzione direttamente

Moltissimi esempi su questo libro invocano una funzione in maniera diretta. Questo serve anche per avere immediatamente un risultato o un valore.

Definiamo che una funzione viene invocata direttamente per distinguerla dagli altri meccanismi di invocazione, cioè: metodi, costruttori, *apply* e *call*. Questo tipo di invocazione si verifica quando una funzione viene invocata utilizzando l'operatore *()* e l'espressione a cui viene applicato l'operatore *()* non fa riferimento alla funzione come proprietà di un oggetto:

```
function moltiplica(a, z) {  
    return a * z;  
}
```



```
}  
moltiplica(5, 4);
```

Potremo anche fare in questo modo, per fare un esempio estremo:

```
let a = function prova(b) {if(b<=2) return 3; else return  
b*prova(b+4);};
```

Qui sopra definiamo una funzione senza nome assegnandola alla variabile *a*. Di fatto non creiamo davvero una funzione *prova*, ma all'interno della stessa possiamo utilizzare il suo nome.

Evocare una funzione utilizzando il costruttore

Le funzioni costruttrici sono dichiarate nello stesso modo di qualsiasi altra funzione, e possiamo utilizzare le dichiarazioni di funzioni e le espressioni di funzioni *per creare nuovi oggetti*. L'unica eccezione riguarda le funzioni freccia, che affronteremo più avanti nel capitolo. Torniamo ora alla chiamata di una funzione tramite il costruttore, che utilizzeremo con la parola chiave *new*:

```
function prova(a1, z8) {  
  this.nome = a1;  
  this.titolo = z8;  
}  
let x = new prova("Rossi","Sig.");  
x.nome;
```

Sembra che *new* crei una nuova funzione, ma dal momento che le funzioni di JavaScript sono oggetti, in realtà, *new* creerà un nuovo oggetto (*prova*). Il nuovo oggetto eredita le proprietà e i metodi dal suo costruttore.

Lo scopo dei costruttori è inizializzare il nuovo oggetto che verrà creato dalla chiamata della funzione alle condizioni iniziali. E sebbene tali funzioni possano essere chiamate come funzioni "normali", o anche assegnate alle proprietà degli oggetti per essere invocate come metodi, generalmente non sono utili in quanto tali.

Poiché i costruttori sono generalmente codificati e utilizzati in un modo diverso dalle altre funzioni e non sono poi così utili a meno che non vengano richiamati come costruttori, è nata una convenzione di denominazione per distinguere i costruttori dalle funzioni e dai metodi comuni.

Le funzioni e i metodi sono generalmente denominati che iniziano con un verbo che descrive ciò che fanno (*moltiplicare*, *specificare*, etc.) e iniziano

con una lettera minuscola. I costruttori, d'altra parte, sono solitamente chiamati come un nome che descrive l'oggetto che viene costruito e inizia con un carattere maiuscolo: *Gatto*, *Topo*, etc. e anche se qualche volta negli esempi non è stato mostrato sarà bene utilizzare questa convenzione quando andremo a scrivere il nostro listato.

Evocare una funzione come metodo di un oggetto

Questa modalità ci permetterà di sfruttare i vantaggi derivati dall'essere un oggetto, nello specifico, quando useremo il metodo faremo riferimento all'oggetto proprietario di esso. Procediamo per gradi: quando una funzione viene assegnata a una proprietà di un oggetto e l'invocazione avviene utilizzando quella proprietà, la funzione verrà richiamata come metodo di quell'oggetto:

```
let specifichiamo = {
  tipo:"Gatto",
  colore: "Rosso",
  insieme: function () {
    return this.tipo + " " + this.colore;
  }
}

specifichiamo.insieme();
```

Di conseguenza, quell'oggetto diventerà il contesto della funzione e sarà disponibile all'interno. In questo caso specifico il metodo “*insieme*” è una funzione che appartiene all'oggetto “*specifichiamo*” e *this* è l'oggetto lo possiede.

Evocare una funzione con i metodi *apply* e *call*

Abbiamo appena visto che In JavaScript, le funzioni sono oggetti e che hanno proprietà e metodi.

call() e *apply()* sono metodi della funzione predefiniti di JavaScript. Entrambi possono essere utilizzati per invocare una funzione, ed entrambi i metodi devono avere l'oggetto proprietario come primo parametro:

```
let mioOggetto
function moltiplicaFunz(a, z) {
  return a * z;
}

mioOggetto = moltiplicaFunz.call(mioOggetto, 6, 3);
```

Vediamo il metodo *apply*:

```
let mioOggetto, mioArray;  
function miaFunzione(a, z) {  
    return a * z;  
}  
  
mioArray = [6, 3];  
mioOggetto = miaFunzione.apply(mioOggetto, mioArray);
```

Entrambi i metodi hanno un oggetto proprietario come primo argomento. L'unica differenza è che *call()* prende gli argomenti della funzione separatamente e *apply()* prende gli argomenti della funzione in un vettore (array).

In modalità rigorosa JavaScript, il primo argomento diventa il valore di *this* nella funzione invocata, anche se l'argomento non è un oggetto.

In modalità "non-strict", se il valore del primo argomento è *null* o *undefined*, viene sostituito con l'oggetto globale.

Con *call()* o *apply()* è possibile impostare il valore di *this*, e di richiamare una funzione come un nuovo metodo di un oggetto esistente.

Arrivati a questo punto potremo chiederci: come decidiamo quale usare? La risposta è la stessa di molte di queste domande: *utilizziamo quella che migliora la chiarezza del codice*. Una risposta più pratica è servirsi di quella che meglio corrisponde agli argomenti che abbiamo a portata di mano. Se abbiamo un mucchio di valori non correlati nelle variabili o specificati come letterali, *call* ci consente di elencarli direttamente nella sua lista di argomenti. Ma se abbiamo già i valori degli argomenti in un insieme (vettore o array), o se è conveniente raccogliarli come tali, *apply* potrebbe essere la soluzione migliore.

A questo punto andremo ad approfondire un concetto inerente al contesto della funzione, per questo ci orienteremo verso il cosiddetto *callback*, tradotto *funzione di richiamo*. In poche parole, possiamo affermare che vi è un callback dove una funzione ne accetta un'altra come parametro. Nelle funzioni di callback (come nei gestori di eventi), il contesto della funzione potrebbe non essere esattamente quello che ci aspettiamo, ma possiamo usare la chiamata e applicare metodi per aggirare il problema. Oltre a ciò, nella sezione seguente vedremo anche le *funzioni freccia* e il metodo *bind*.

Funzioni freccia, callback e this

Ne abbiamo già parlato precedentemente, ma adesso ci interessa che le funzioni freccia hanno anche una caratteristica che le rende particolarmente utili come funzioni di callback. Sappiamo già che hanno una sintassi breve e non hanno un proprio valore:

```
[5,6,7].map(x => 1 + x) // [6, 7, 8]
```

e ricordano il valore del parametro *this* al momento della loro definizione. Solitamente *this* chiamava la funzione, mentre con le funzioni freccia rappresenta l'oggetto che l'ha definito. Vediamo un esempio che ci schiarirà le idee:

```
<html>
<body>
<button id="btn">Cliccami</button>
<p id="b"></p>
<script>
// Funzione tradizionale
let a = function() {
  document.getElementById("b").innerHTML += this;
}

// Aggiungiamo un listener per l'evento "load" della finestra
window.addEventListener("load", a);

// Aggiungiamo un listener per l'evento "click" del pulsante
document.getElementById("btn").addEventListener("click", a);
</script>
</body>
</html>
```

Qui vediamo che, in una funzione normale, *this* può rappresentare oggetti diversi. Quando cliccheremo sul bottone si eseguirà la funzione *a* dove *this* è il bottone: con l'istruzione in basso l'oggetto *Window*, cioè la nostra finestra, chiamerà la funzione *a*:

```
window.addEventListener("load", a);
```

Poi anche l'oggetto bottone chiamerà la funzione *a*:

```
document.getElementById("btn").addEventListener("click", a);
```

Se invece il codice fosse impostato in questo modo:

```
<html>
```

```

<body>
<button id="btn">Cliccami</button>
<p id="b"></p>
<script>
// Funzione freccia
let a = () => {
  document.getElementById("b").innerHTML += "Cliccato da " + this;
}

// Aggiungiamo un listener per l'evento "load" della finestra
window.addEventListener("load", a);

// Aggiungiamo un listener per l'evento "click" del pulsante
document.getElementById("btn").addEventListener("click", a);
</script>
</body>
</html>
</html>

```

Quando il pulsante viene cliccato, il testo *"Cliccato da [object Window]"* verrà visualizzato nel paragrafo. Questo dimostra che con le funzioni freccia, *this* rappresenta l'oggetto globale che ha definito la funzione, che nel caso del browser è l'oggetto *Window*. In pratica le funzioni freccia offrono un modo coerente per gestire il valore di *this* nei *callback*, semplificando il comportamento rispetto alle funzioni tradizionali.

Tuttavia, le funzioni freccia ereditano il valore di *this* dal contesto circostante al momento della loro definizione. In questo caso, la funzione freccia è stata creata nel contesto globale, quindi *this* all'interno della funzione freccia si riferisce all'oggetto globale (*Window*).

Se desideri accedere all'oggetto pulsante all'interno della funzione freccia, dovresti passare esplicitamente l'oggetto pulsante come argomento alla funzione o utilizzare una funzione tradizionale che mantenga il contesto corretto.

Perciò dobbiamo anche ricordarci di fare attenzione agli ambiti e ai contesti di esecuzione, ad esempio:

```

const esempio = {
  nome: 'Mario',
  altriNomi: ['Gino', 'Paolo'],
  prop:()=> this.altriNomi.forEach(a => console.log(this.nome + ' '
+ a))
}; // errore!

```

Infatti qui abbiamo passato una funzione anonima a *forEach* alla quale non è legata e non ve ne è un'altra esterna da cui ricavare *this*.

Ricordiamoci sempre che in strict-mode (modalità rigorosa) *this* è *undefined*, mentre normalmente è il contesto globale (la finestra del browser).

Infine, non lo ripeteremo mai abbastanza: le funzioni freccia raccolgono il valore del parametro *this* al momento della loro creazione. Poichè il cliclo funzione freccia viene creato come valore di proprietà su un oggetto letterale e l'oggetto letterale viene creato nel codice globale, il valore della funzione freccia sarà un valore del codice globale.

Ora che abbiamo esplorato come utilizzare le funzioni freccia per aggirare il problema di contesti di funzione, continuiamo con un altro metodo per risolvere lo stesso problema.

Il metodo bind

In questo libro capiterà di dover tornare su alcuni argomenti e forse anche di dover ripetere determinati concetti. Verrà fatto con lo scopo di inculcare le possibili differenze o peculiarità degli specifici argomenti. Ma torniamo a noi, come da premessa ritorneremo su due metodi a cui ogni funzione ha accesso, cioè *call* e *apply*. Abbiamo visto come usarli per ottenere un maggiore controllo sul contesto e sugli argomenti delle nostre invocazioni di funzioni. Il preambolo vuole sottolineare la diversità tra *call* e *apply*, e il metodo *bind* a cui ogni funzione ha accesso, e che, praticamente creerà una nuova funzione che manterrà il suo contesto legato a un determinato oggetto. Ecco in un listato:

```
const provabind = {
  a: 10,
  prendiLaa: function() {
    return this.a;
  }
};

const prendiAerr = provabind.prendiLaa;
prendiAerr(); // funzione chiamata nel contesto globale, darà
undefined

const prendiAok = prendiAerr.bind(provabind);
prendiAok(); // 10
```

Il metodo *bind* è disponibile per tutte le funzioni ed è progettato per creare e restituire una nuova funzione associata all'oggetto passato (in questo caso,

l'oggetto *a*). Il valore del parametro *this* è sempre impostato su quell'oggetto, indipendentemente dal modo in cui è stata invocata la funzione associata, il valore di *this* sarà sempre impostato su quell'oggetto. Nel nostro esempio queste righe di codice invocano la funzione nel contesto globale restituendo *undefined*:

```
const prendiAerr = provabind.prendiLaa;  
prendiAerr(); // undefined
```

Quando si chiama *prendiAok()*, la funzione è stata "legata" all'oggetto *provabind* utilizzando *bind*, il che fa sì che *this* all'interno della funzione si riferisca sempre a quell'oggetto, *quindi restituisce 10*:

```
const prendiAok = prendiAerr.bind(provabind);  
prendiAok(); // 10
```

Quindi, riassumendo, e come abbiamo appena visto nel codice che ci ha preceduto Il metodo *bind()* crea una nuova funzione associata che, quando invocata tramite la parola chiave *this* impostata sul valore fornito, esegue una determinata sequenza di argomenti prima di qualsiasi altro argomento passato alla funzione. Guardiamo questo:

```
this.a = 1;  
const module = {  
  a: 5,  
  prendiA: function() {return this.a;}  
};  
module.prendiA(); // il risultato sarà 5  
  
const prendiA = module.prendiA;  
prendiA(); // il risultato sarà 1
```

Creiamo un'altra funzione con *this* associato al modulo, i neofiti potrebbero confondere la variabile globale 'a' con la proprietà 'a' del modulo:

```
const limitePrendiA = prendiA.bind(module);  
limitePrendiA(); // il risultato sarà 5
```

Spieghiamo bene cosa accade: quando si chiama *module.prendiA()*, il risultato è 5 perché *this* all'interno della funzione è legato all'oggetto *module*. Quando si chiama *prendiA()*, il risultato è 1 perché la funzione è stata assegnata a una variabile globale e viene chiamata nel contesto globale dove

this rappresenta l'oggetto globale.

Quando si chiama *limitePrendiA()*, la funzione è stata legata all'oggetto *module* utilizzando *bind*, quindi *this* all'interno della funzione è legato a *module* e il risultato è 5.

Con questo, terminiamo la nostra esplorazione sulle funzioni. Ma non del tutto, infatti nel prossimo capitolo avremo a che fare con le chiusure, argomento che riguarderà gli ambiti in cui le funzioni potranno interagire tra di loro.

Quiz & esercizi

1) Creare un listato con una funzione dichiarativa e due operatori la quale produca il risultato numerico di 300.

2) Correggere il listato di seguito in modo riporti correttamente la somma totale dei tre pasti del gatto Briciola:

```
function totpasti (a,b,c,x){
  return a + b;
}
function pasti(x){
  let colaz = parseInt(prompt('Quanti gamberi mangia Briciola a colazione?'));
  let pranzo = parseInt(prompt('Quanti gamberi mangia Briciola a pranzo?'));
  let cena = parseInt(prompt('Quanti gamberi mangia Briciola a cena?'));
  alert(x(pranzo,cena));
}

pasti(totpasti);
```

3) Scrivere una funzione che in una sola riga di codice riporti il risultato di un'operazione (es. a+b).

4) Utilizzare una funzione immediata per scrivere “buongiorno” in una finestra.

5) Ordinare questi numeri: 5, 8, 2, 6 utilizzando *sort* e la funzione freccia.

6) Correggere il codice per sommare tutti i numeri:

```
function somma(a, b) {
  return a + b;
}
alert(somma(1, 2, 3, 4, 5));
```

7) Convertire gli argomenti della funzione in un array:


```
function f(a, b, c) {}
```

8) Invocare la funzione come metodo nella console:

```
let metodi = {  
  tipo: "Cane",  
  colore: "Bianco",  
  evocati: function () {  
    return this.tipo + " " + this.colore;  
  }  
}
```

9) Utilizzare i metodi *call* e *apply* per ottenere lo stesso risultato del listato precedente.

10) Ottenere il numero 7 tramite il metodo *bind*:

```
this.x = 1;  
const module = {  
  x: 7,  
  ris: function() {return this.x;}  
};
```

11) Crea una funzione chiamata *saluta* che accetti un nome come argomento e restituisca una stringa di saluto che includa il nome.

12) Crea quattro funzioni separate per eseguire operazioni matematiche di base (addizione, sottrazione, moltiplicazione e divisione).

13) Crea una funzione chiamata *isPari* che accetti un numero come argomento e restituisca *true* se il numero è pari, altrimenti restituisca *false*.

14) Crea una funzione chiamata *conteggioVocali* che accetti una stringa come argomento e restituisca il numero di vocali (a, e, i, o, u) presenti nella stringa.

15) Crea una funzione chiamata *isPalindromo* che accetti una stringa come argomento e restituisca *true* se la stringa è un palindromo (legge lo stesso sia da sinistra a destra che da destra a sinistra), altrimenti restituisca *false*.

16) Crea una funzione chiamata *fattoriale* che accetti un numero intero positivo come argomento e restituisca il fattoriale di quel numero.

17) Crea una funzione chiamata *sommaArray* che accetti un array di numeri come argomento e restituisca la somma di tutti gli elementi nell'array.

18) Crea una funzione chiamata *trovaMax* che accetti un array di numeri come argomento e restituisca il numero più grande presente nell'array.

19) Crea una funzione chiamata *calcolaPotenza* che accetti due argomenti, la base e l'esponente, e restituisca il risultato della base elevata all'esponente.

20) Crea una funzione chiamata *conteggioParole* che accetti una stringa come argomento e restituisca il numero di parole presenti nella frase. Le parole sono separate da spazi.

21) Crea una funzione chiamata *generaNumeriCasuali* che accetti un numero intero minimo e un numero intero massimo come argomenti e restituisca un numero casuale compreso tra minimo e massimo.

22) Crea una funzione chiamata *fibonacci* che accetti un numero n come argomento e restituisca il valore della sequenza di Fibonacci corrispondente all'indice n. Utilizza la ricorsione per calcolare il valore.

23) Crea una funzione *memoize* che accetti una funzione *funzioneOriginale* come argomento e restituisca una nuova funzione che memorizza i risultati della funzione originale per gli stessi input, evitando di ricalcolarli.

24) Crea una funzione chiamata *ordina* che accetti un array e una funzione di confronto personalizzata come argomenti e restituisca una copia dell'array ordinata in base alla funzione di confronto.

25) Crea una funzione *eseguiInSequenza* che accetti un array di funzioni asincrone e le esegua una dopo l'altra in ordine. Assicurati che la funzione successiva venga chiamata solo quando la funzione precedente ha completato la sua esecuzione.

Riassunto

Abbiamo visto che il nostro caro JavaScript è un linguaggio di programmazione con significative caratteristiche orientate alla funzionalità. Abbiamo esplorato le differenze tra gli argomenti della chiamata e i parametri della funzione e il modo in cui i valori vengono trasferiti dagli argomenti ai parametri della funzione. È subito stato chiaro che il cuore del codice di JS è stato pensato per essere estremamente versatile, ed eccoci alle funzioni come oggetti.

In pratica di funzioni ce ne sono diverse tipologie, le abbiamo studiate tutte insieme ai modi per richiamarle.

Dopo aver imparato bene tutto questo saremo prontissimi per gli ambiti e le loro regole.

7 - Le Chiusure

Concetto di Scope

Utilizzo delle chiusure

Monitorare l'esecuzione di programmi JavaScript

Comprensione dei tipi di variabili

Il funzionamento delle chiusure

Il concetto di chiusura inizialmente potrebbe risultare un po' ostico e, per questo motivo, spesso chi è alle prime armi (ma non solo) tende a non usufruire di questa funzione utilissima che ci mette a disposizione Javascript, soprattutto in determinati ambiti come ad esempio le animazioni la gestione degli eventi.

Cercheremo di studiare le chiusure in modo da capirle per bene e per trarne tutti i vantaggi che ne deriveranno dal suo utilizzo nei nostri programmi. Iniziamo con il dire che la chiusura è, semplificando al massimo, una funzione o una procedura che delimita e chiude un ambiente al quale però si potrà accedere anche dopo che l'esecuzione stessa della funzione è stata completata, dato che normalmente, come sappiamo, questo non accade. Ovviamente vedremo anche l'abbinamento ai callback che sono l'ovvia conseguenza per accedere a determinati valori confinati. Prima però analizziamo il concetto inerente allo *scope*, ossia il contesto di esecuzione.

Concetto di scope o contesto di esecuzione

È molto importante comprendere la differenza tra contesto globale e locale. Sintetizzando il concetto di scope si riferisce proprio a questo, ossia l'ambito di validità di una variabile. Quando parliamo di una variabile con scope globale intendiamo quindi che essa ha validità nell'intero listato di codice (script). Di contro l'ambito locale, quindi una variabile con scope locale, avrà valore solo all'interno della funzione o in una porzione di codice.

```
let valore = 1;  
let prova = function() {
```

```
        console.log(valore);
    }
    prova();

console.log(valore);
```

Abbiamo appena visto come una variabile dichiarata prima della funzione abbia uno scope globale, quindi il valore 1 viene visto sia all'interno della funzione che all'esterno. Come risultato verrà mostrato due volte il valore 1.

```
let valore = 1;
let prova = function() {
    let valore2 = 2;
    console.log(valore);
    console.log(valore2);
}
prova();

console.log(valore);
console.log(valore2); // errore!
```

Ed ecco che il *valore2*, che in questo caso ha uno scope locale dato che è stato dichiarato all'interno della funzione, viene visto solamente all'interno di *prova*, mentre all'esterno di essa non è considerato. Ne consegue l'ovvio errore di mancata definizione della variabile.

Utilizzo delle chiusure

Per permettere ad una funzione di avere l'accesso e di manipolare variabili esterne a quella funzione, o ad altre, avremo certamente bisogno del supporto delle cosiddette *chiusure*, tenendo bene a mente le differenze tra tutti i vari ambiti o *scope*.

Avremo anche a che fare con funzioni dichiarate, le quali potranno essere chiamate in qualsiasi momento, anche dopo che l'ambito (*scope*) in cui era stata dichiarata non esiste più.

Iniziamo con gli esempi che ci aiutano sempre a comprendere:

```
function abc() {
    let animale = "Gatto coccoloso";
    function mostralo() {
        alert(animale);
    }
    mostralo();
}
```

```
abc();
```

Si parte con *abc()* che crea una variabile “*animale*” e poi una funzione *mostralo()*. *Animale()* è una funzione interna, definita dentro *abc()* ed è disponibile solo all'interno del corpo di quella funzione. *Mostralo()* non ha proprie variabili locali. Tuttavia, poiché le funzioni interne hanno accesso alle variabili di quelle esterne, *mostralo()* può accedere alla variabile *animale* dichiarata nella funzione genitore *abc()*.

```
function cde() {  
  let animale = "Gatto coccoloso";  
  function mostralo() {  
    alert(animale);  
  }  
  return mostralo;  
}  
  
let fgh = cde();  
fgh();
```

L'esecuzione di questo codice è praticamente identica alla funzione *abc()*: la stringa “*Gatto coccoloso*” verrà visualizzata in una casella di avviso JavaScript. La cosa differente è che la funzione interna *mostralo()* viene restituita dalla funzione esterna prima di essere eseguita.

A un primo sguardo sembrerebbe controintuitivo che il codice funzioni ancora. In alcuni linguaggi di programmazione le variabili locali interne a una funzione esistono solo per la durata dell'esecuzione della funzione stessa. Una volta che *cde()* ha terminato la propria esecuzione, ci si potrebbe aspettare che la variabile *animale* non sia più accessibile. Tuttavia, poiché il codice funziona ancora, è ovvio che in JavaScript non è il caso.

La ragione, come abbiamo accennato all'inizio, è che le funzioni in JavaScript formano *closures* (chiusure). Esse sono la combinazione di una funzione e dell'ambito lessicale in cui questa funzione è stata dichiarata. In questo caso, *fgh* è un riferimento all'istanza della funzione *mostralo* creata quando *cde* è eseguita.

La soluzione di questo rompicampo è che *fgh* è diventata una chiusura, cioè uno speciale tipo di oggetto che combina due cose: una funzione e l'ambito in cui questa è stata creata. L'ambito consiste in qualsiasi variabile locale che era nel suo *scope* (contesto di esecuzione) nel momento in cui la chiusura è stata creata. In questo caso, *fgh* è una chiusura che incorpora sia la

funzione *mostralo* che la stringa "Gatto coccoloso", già esistente quando la chiusura è stata creata.

Un altro esempio per capire le chiusure è con i moduli:

```
function ModuloYeah() {
  let gatto = "Voglio le coccole";
  let quantita = [1, 2, 3];

  function azione() {
    console.log(gatto);
  }
  function altraAzione() {
    console.log(quantita.join("!"));
  }
  return {
    azione: azione,
    altraAzione: altraAzione
  };
}
let vai = ModuloYeah(); // Creo il modulo
vai.azione(); // Coccole
vai.altraAzione(); // 1 ! 2 ! 3
```

Qui la funzione *ModuloYeah* espone due funzioni definite internamente (azione, altraAzione) perché ritorna un oggetto JavaScript che fa riferimento ad esse, le quali hanno accesso allo scope interno alla funzione (gatto, quantita), anche se vengono eseguite al di fuori di esso! così infatti che possiamo definire una API pubblica: *vai* infatti può avere accesso alle funzioni *azione* e *altraAzione*, ma non alle variabili dentro a *ModuloYeah*. Ciò ci permette di evitare collisioni tra nomi e non sovraffollare lo scope di variabili, nascondendo ciò che vogliamo mantenere privato.

Chiusure e Callback

Potremo vedere spesso le chiusure anche quando avremo che fare con i callback, in sostanza quando una funzione verrà chiamata in un secondo momento non specificato rispetto alla normale sequenza di eventi, ecco un esempio:

```
function sottrazione (a,z){
  return a - z;
}

function eccoilCallBack(callback){
  let n1 = parseInt(prompt('Digita un numero'));
  callback(n1);
}
```

```
    let n2 = parseInt(prompt('Digita un numero'));
    alert(callback(n1,n2));
}
```

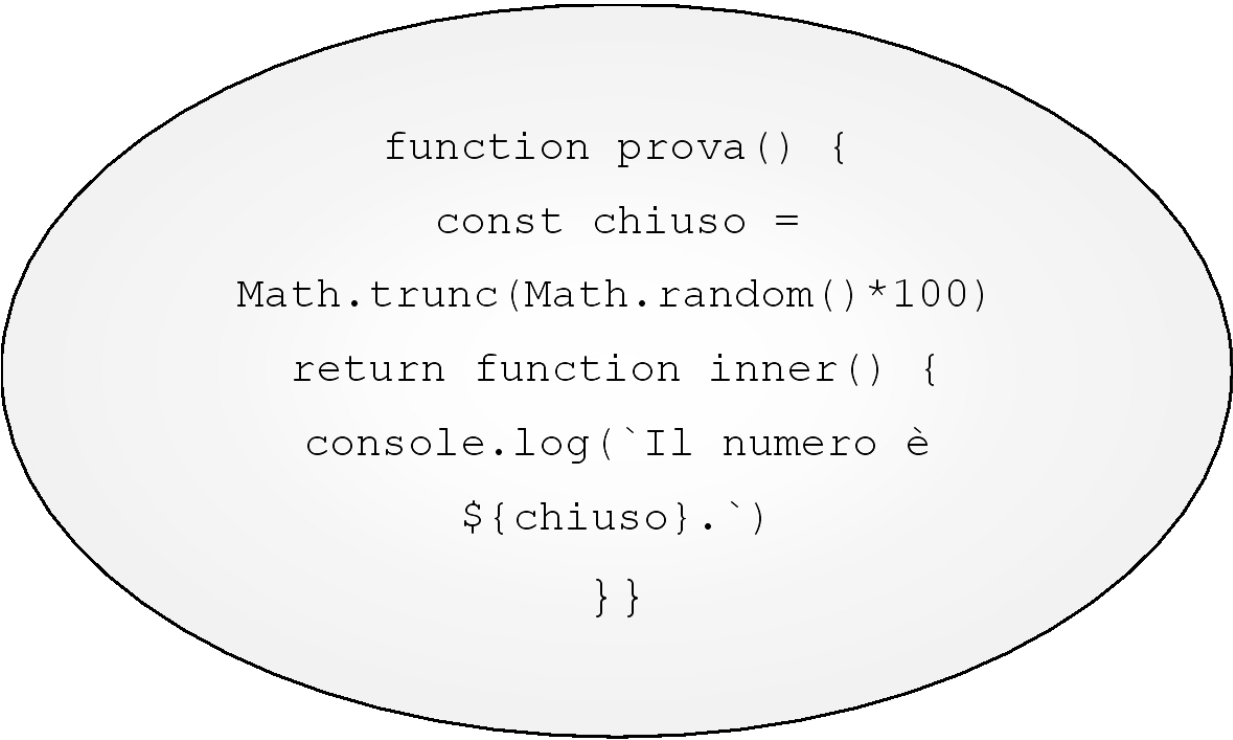
eccoilCallBack(sottrazione);

Vediamo quest'altro esempio:

```
function prova() {
  const chiuso = Math.trunc(Math.random()*25)
  return function inner() {
    console.log(`Il numero è ${chiuso}.`)
  }
}
```

const a = prova() // non potremo vedere "*chiuso*" dall'esterno
a() // potremo farlo solamente invocando a

Se tentassimo di estrarre il numero *chiuso* avremo un errore, perché avremo tentato di forzare la chiusura della funzione *prova*, rappresentiamolo meglio graficamente:



```
function prova() {
  const chiuso =
Math.trunc(Math.random()*100)
  return function inner() {
    console.log(`Il numero è
    ${chiuso}.`)
  }}
}
```



```
alert(chiuso); // errore!
```

Dalla grafica si evince che la funzione *prova* ha un suo ambito chiuso (chiusura) non accessibile dall'esterno, quindi il valore di *chiuso* non è estrapolabile da fuori.

Qui sotto invece vediamo il callback assieme alla funzione *setTimeout* per aiutarci anche a rallentare le chiamate di funzione:

```
function prova(callback, fattore, tempo){  
  let callInt = function(tick, contatore) {  
    return function() {if (--tick >= 0) {  
      window.setTimeout(callInt, ++contatore * fattore);  
      callback();  
    }}(tempo, 0);  
    window.setTimeout(callInt, fattore);  
  };  
}
```

```
prova(function(){console.log('Ciao'); }, 10, 10);  
prova(function(){console.log('Addio'); }, 100, 10);
```

Le chiusure hanno uno stretto legame con il contesto di esecuzione, quindi dedicheremo una buona parte di questo capitolo all'esplorazione delle sue regole.

JavaScript si basa sull'ambito per stabilire quali parti di codice sono accessibili in un dato punto dell'esecuzione del programma. L'ambito può essere considerato come un recinto attorno a particolari parti di codice in un programma. JavaScript utilizza l'ambito lessicale, il che significa che l'ambito è definito dal codice sorgente al momento dell'autore, piuttosto che cambiare dinamicamente al momento dell'esecuzione. L'ambito esiste a livello globale (il livello più alto del codice) e a livello locale (all'interno di singole funzioni e, in alcuni casi, blocchi). L'ambito è anche annidato in natura, il che significa che il codice a livelli inferiori di ambito annidato può accedere a variabili definite a livelli superiori, ma non viceversa. Le singole funzioni mostrano anche un comportamento di chiusura dell'ambito, il che significa che si chiudono su variabili che rientrano nell'ambito nel momento in cui una funzione viene definita. Le chiusure consentono alle funzioni di

mantenere l'accesso a una determinata variabile anche se richiamate da un ambito diverso.

```
let gelato = "fragola e limone";
function mangiareGelato() {
  console.log("Sto mangiando il gelato gusto " + gelato + "!");
}

function merenda() {
  let gelato = "A";
  mangiareGelato ();
}

merenda();
```

In questo esempio, abbiamo una funzione *mangiareGelato* e una variabile *gelato*, entrambe parte dell'ambito globale, *mangiareGelato* accede alla variabile *gelato* che registra un testo nella console. Definita anche in ambito globale abbiamo una funzione *merenda*, che definisce la propria variabile *gelato* nel proprio ambito locale e quindi invoca *mangiareGelato*, a cui può accedere dallo *scope* globale.

Approfondiamo le variabili

Adesso approfondiremo i vari modi in cui una variabile può essere dichiarata: in JavaScript, come abbiamo già visto, possiamo utilizzare tre parole chiave per definire le variabili: *var*, *let* e *const*. Differiscono in due aspetti: la mutabilità e la loro relazione con l'ambiente lessicale.

Se dovessimo dividere le parole chiave di dichiarazione delle variabili per mutabilità, metteremmo *const* da un lato e *var* e *let* dall'altro, ricordando che *var* è considerata obsoleta ed è sconsigliato il suo utilizzo nelle applicazioni attuali, tuttavia è bene conoscere le sue peculiarità. Tutte le variabili definite con *const* sono immutabili, il che significa che il loro valore può essere impostato una sola volta, mentre, le variabili definite con le parole chiave *var* e *let* potrebbero cambiare nel corso dell'esecuzione del programma, anche più volte.

Tutte le variabili assegnate a *const* saranno simili a una variabile normale, con l'eccezione che dovremmo fornire un valore di inizializzazione quando verranno dichiarate e non potremo assegnarne un altro in seguito. Effettivamente sembrerebbe un pò limitata, ma andiamo oltre.

Le variabili *const* vengono spesso utilizzate per due scopi leggermente diversi: come detto per specificare delle variabili fisse o per fare riferimento a un valore fisso per nome, ad esempio il numero massimo di calciatori in una squadra: `const calciatoriMax = "players"` , invece di usare un numero letterale come 11. Questo rende i nostri programmi più facili da capire e mantenere. Vediamo il comportamento delle variabili *const*:

```
const primo = "cane";
assert(primo === "cane", "primo è un cane"); // true

try {primo = "gatto"; fail ("primo non è un gatto");
} catch(e) {pass ("Qui abbiamo un'eccezione");
}

assert(primo === "cane", "primo è ancora un cane!");
const seconda = {};
```

Il tentativo di attribuire un nuovo valore a una variabile assegnata con *const* genera un'eccezione. Ora creiamo un'altra variabile con *const* e gli destiniamo un oggetto:

```
seconda.nome = "Pussy";
assert(seconda.nome === "Pussy"); // true
```

Non possiamo assegnare un oggetto completamente nuovo alla variabile *seconda*, ma nulla ci impedisce di modificare quello che abbiamo già.

```
const terza = [];
assert(terza.length === 0, "Nessun elemento nel nostro array");
terza.push("Mario");
assert(terza.length === 1, "L'array è cambiato");
```

La stessa identica cosa vale per i vettori (array).

Qui definiamo prima una variabile servendoci di *const* chiamata *prima* con un suo valore, e testiamo che la variabile è stata inizializzata:

```
const prima = "elefante";
assert(prima === "elefante", "prima è un elefante");
```

Continuiamo cercando di assegnare un valore completamente nuovo, alla variabile *prima*:

```
try {prima = "gatto"; fail ("Il valore non cambia");
```

```
} catch (e) {pass ("Eccezione");  
}
```

Poiché la variabile *prima* è una costante, non possiamo assegnarle un nuovo valore, quindi il motore JavaScript genera un'eccezione senza modificare il valore della variabile.

Nel frattempo stiamo utilizzando due funzioni che non abbiamo visto finora: *fail* e *pass*. Questi due metodi si comportano in modo simile al metodo *assert*, tranne che *fail* fallisce sempre e *pass* passa sempre. Qui li usiamo per verificare se si è verificata un'eccezione, in tal caso viene attivata l'istruzione *catch* e viene eseguito il metodo *pass*. Se non ci sono eccezioni, viene eseguito il metodo *fail* e ci verrà notificato l'errore.

Proviamo adesso a definire un'altra variabile con *const*, questa volta inizializzandola su un oggetto vuoto:

```
const oggVuoto = {};
```

Ora discuteremo un'importante caratteristica delle variabili assegnate con *const*. Come abbiamo già visto, non potremo attribuire a una variabile un valore completamente nuovo, ma niente ci impedisce di modificare quello attuale. Ad esempio, possiamo aggiungere nuove proprietà all'oggetto corrente:

```
oggVuoto.nome = "Pussy";  
assert(oggVuoto.nome === "Pussy"); // true
```

O, se la nostra variabile *const* si riferisce a un array, possiamo modificarlo rimpendolo o ingrandendolo:

```
const prova = [];  
assert(prova.length === 0); // true  
  
prova.push ("Mario");  
assert(prova.length === 1); // true
```

Per adesso possiamo dire che le regole dietro le variabili evocate tramite *const* non sono così complicate all'inizio. L'importante è tenere a mente che *const corrisponde a una costante, un valore immutabile*, anche se, abbiamo visto che trattando la costante come un oggetto avremo la possibilità di effettuare dei cambiamenti sulle sue proprietà e sul suo contenuto.

Variabili metasintattiche

Durante l'apprendimento della programmazione, è probabile che incontrerete il termine "foo" o altre variabili simili come "bar" o "baz" nei vari esempi e esercizi. Queste variabili sono comunemente utilizzate dai programmatori come segnaposto per rappresentare un valore generico che può essere modificato in base alle condizioni o alle informazioni specifiche del programma. In linguaggio tecnico, queste variabili sono conosciute come "variabili metasintattiche".

Ecco un semplice esempio che utilizza la variabile "foo":

```
let foo = {  
  testo: function() {  
    alert("Esempio riuscito!");  
  }  
};
```

```
foo.testo();
```

In questo caso, abbiamo dichiarato una variabile "foo" che è un oggetto contenente un metodo chiamato "testo". Quando chiamiamo "foo.testo()", verrà visualizzato un messaggio di avviso che indica "Esempio riuscito!".

Le variabili metasintattiche come "foo" sono spesso utilizzate nei tutorial e negli esempi per semplificare la comprensione del codice, ma nella pratica, dovreste utilizzare nomi di variabili più descrittivi e significativi per rendere il vostro codice più chiaro e leggibile.

Gli ambiti in Javascript

Prima di proseguire è importante distinguere bene e conoscere ciò a cui facciamo riferimento quando parliamo di ambiti globali o lessicali.

Essenzialmente in Javascript esiste *l'ambito lessicale* con *l'ambito di funzione*. Ogni funzione potrà creare un nuovo ambito.

Ogni nuovo ambito (o volgarmente area, zona per capirci) potrà avere al suo interno delle variabili le quali non saranno accessibili da fuori. Quindi avremo avremo due aree contraddistinte, locale e globale.

La prima è quella parte al di fuori delle funzioni, la seconda riguarda quella all'interno di esse.

L'ambito locale è a sua volta diviso in ambito di blocco e di funzione. Il primo è quello all'interno di un loop, l'altro è ovviamente quello della funzione.

Le variabili e la loro validità all'interno delle varie "zone" viene definita dalla posizione in cui sono dichiarate. Javascript le cercherà a partire dalla

zona più interna proseguendo verso l'esterno finchè non le troverà. Quindi queste variabili saranno accessibili a partire dall'interno verso l'esterno.

Questo è il principio di base, ci sono molte altre sfumature, torneremo su questo argomento poco più avanti e sapremo di cosa stiamo parlando.

La variabile var (obsoleta)

Fatto nostro il concetto sulle costanti adesso è il turno delle variabili, quindi eccoci con *var*. Ora che abbiamo esaminato le costanti, è il momento di parlare delle variabili, in particolare della parola chiave "var". Come abbiamo già menzionato, viene utilizzata per dichiarare una variabile il cui valore potrebbe cambiare nel corso dell'esecuzione del programma. Tuttavia, è importante sottolineare che le variabili *var* sono definite nell'ambiente lessicale più vicino, che può essere la funzione corrente o l'ambiente lessicale globale. Ciò significa che le strutture di blocco (come gli "if" o i cicli "for") non impediscono l'accesso alle variabili definite con *var*. Ecco un esempio:

```
var globale = "Kitty";

function cosaFa() {
    var funzFare = "mangia";

    for (var x = 1; x < 3; x++) {
        var osserviamo = globale + " " + funzFare;
        assert(osserviamo === "Kitty mangia", "Condizione soddisfatta!");
        assert(x, "Contatore loop corrente: " + x);
    }

    assert(x === 3 && osserviamo === "Kitty mangia", "Variabili di
loop accessibili al di fuori del loop");
}

cosaFa();

assert(typeof funzFare === "undefined" && typeof x === "undefined"
&& typeof osserviamo === "undefined", "Variabili locali non
accessibili al di fuori della funzione");
```

In questo esempio, definiamo una variabile globale chiamata *globale* all'inizio del nostro codice. All'interno della funzione *cosaFa*, creiamo una variabile *funzFare* e un ciclo "for" con una variabile "x".

Sorprendentemente, tutte queste variabili sono accessibili sia all'interno del blocco del ciclo che al di fuori di esso.

Questo comportamento è dovuto al fatto che le variabili dichiarate con *var* vengono sempre registrate nell'ambiente lessicale più vicino, senza prestare attenzione ai blocchi. In questo esempio, ci sono tre ambienti lessicali: l'ambiente globale che contiene la variabile "globale", l'ambiente della funzione "cosaFa" che contiene "funzFare", "x" e "osserviamo", e l'ambiente di blocco del ciclo "for".

È importante notare che questo comportamento può essere fonte di confusione, ed è uno dei motivi per cui, a partire dal 2015, JavaScript ha introdotto nuove parole chiave per dichiarare variabili, come "let" e "const". Queste nuove parole chiave risolvono molti dei problemi associati all'uso di "var" e dovrebbero essere preferite per scrivere codice più prevedibile e comprensibile.

Le variabili let e const

Niente panico, anche se le parole chiave sono diverse il concetto è più o meno lo stesso, infatti *var*, come abbiamo imparato, definisce la variabile nella funzione o nell'ambiente lessicale globale più vicino, mentre le parole chiave *let* e *const* si comportano in maniera più esplicita, infatti definiscono le variabili nell'ambiente lessicale più vicino, come un ambiente relativo ad un ciclo, un ambiente di una funzione o anche l'ambiente globale. Potremmo dunque usare *let* e *const* per definire variabili con ambito di blocco, con ambito di funzione e con ambito globale.

Proviamo adesso il nostro esempio precedente usando *const* e *let*. Le variabili *const* globali sono solitamente scritte in maiuscolo.

```
const GLOB_1 = "Kitty";
function cosaFa() {
  const funzFare = "mangia";

  for (let x = 1; x < 3; x++) {
```

All'interno del ciclo *for*, possiamo accedere alle variabili di blocco, alle variabili di funzione e alle variabili globali. Adesso definiamo due variabili nel ciclo *for*, usando *let*:

```
let osserviamo = GLOB_1 + "" + funzFare;
assert(osserviamo === "Kitty mangia"); // true
assert(x, "Contatore loop corrente:" + x);
```

```

}

assert(typeof x === "undefined" && typeof osserviamo ===
"undefined", "Variabili del ciclo non accessibili al di fuori del
ciclo");
}

cosaFa();

assert(typeof funzFare === "undefined" && typeof x === "undefined"
&& typeof osserviamo === "undefined", "Non possiamo vedere le
variabili di funzione al di fuori di una funzione");

```

Normalmente, nessuna delle variabili della funzione è accessibile al di fuori della funzione.

Anche in questo caso abbiamo ancora tre ambienti lessicali: l'ambiente globale (per il codice globale al di fuori di tutte le funzioni e blocchi), l'ambiente *cosaFa* associato alla funzione *funzFare* e l'ambiente a blocchi per il corpo del loop. Ma poiché stiamo usando *let* e *const*, le variabili sono definite nel loro ambiente lessicale più vicino; la variabile *GLOB_1* è definita nell'ambiente globale, la variabile *funzFare* nell'ambiente *cosaFa* e le variabili *x* e *osserviamo* nell'ambiente del loop.

Ovviamente, associati questi principi, e, per i più tecnici, compreso che JavaScript da oramai sette anni supporta le stesse regole di scoping degli altri linguaggi sulla falsariga di C, da qui in avanti cercheremo di usare solamente *const* e *let* al posto di *var*.

Riassumendo, abbiamo capito come le mappature degli identificatori vengono mantenute all'interno degli ambienti lessicali e come questi siano collegati all'esecuzione del programma. La logica conseguenza adesso sarà quella di osservare il processo esatto mediante il quale gli identificatori vengono definiti all'interno degli ambienti lessicali. Questo ci aiuterà anche a capire meglio alcuni bug che si verificano comunemente.

Ora che abbiamo esplorato la mutabilità delle variabili, considereremo i dettagli della relazione tra vari tipi di variabili e gli ambienti lessicali.

Comprendere le regole degli ambiti

Abbiamo già parlato a grandi linee degli ambiti e delle regole che li governano. Vediamo in dettaglio un esempio e lo commenteremo:

```

const alano = "Thor";
razza(alano);

```

```
function razza(ingombrante) {  
    console.log(ingombrante === "Thor", "Come è possibile?");  
}
```

Quindi, abbiamo assegnato il valore *Thor* all'identificatore *alano*, poi chiamiamo la funzione *razza* con l'identificatore *alano* come parametro. Ma un attimo: se il codice viene eseguito riga per riga, dovremmo essere in grado di chiamare la funzione *razza*? L'esecuzione del nostro programma non ha raggiunto la sua dichiarazione, quindi il motore JavaScript in teoria non dovrebbe nemmeno saperlo.

In ogni caso lanciando il listato va tutto bene, questo perché JavaScript non è troppo esigente riguardo *a dove definiamo* le nostre funzioni. Possiamo scegliere di inserire le dichiarazioni di funzione prima o anche dopo le rispettive chiamate. Questo non è qualcosa su cui gli sviluppatori dovrebbero preoccuparsi.

Ma a parte la facilità d'uso, se il codice viene eseguito riga per riga, come faceva il motore JavaScript a sapere che una funzione denominata *razza* esiste? Si scopre che il motore JavaScript "imbrogia" un pò e che l'esecuzione del codice JavaScript avviene in due fasi.

La prima fase viene attivata ogni volta che viene creato un nuovo ambiente lessicale. In questa fase, il codice non viene eseguito, ma il motore JavaScript visita e registra tutte le variabili e le funzioni dichiarate all'interno dell'ambiente lessicale corrente. La seconda fase, l'esecuzione di JavaScript, inizia dopo che ciò è stato compiuto; il comportamento esatto dipende dal tipo di variabile (*let*, *var*, *const*, dichiarazione di funzione) e dal tipo di ambiente (globale, funzione o blocco).

Ora esamineremo le implicazioni di queste regole. Vedremo anche come stanare alcuni bug facili da creare ma difficile da capire che potrebbero crearsi se non utilizziamo a dovere le nozioni fin qui imparate. Cominceremo con il motivo per cui siamo in grado di chiamare una funzione prima della sua dichiarazione nella sequenza temporale del codice.

Eseguire una funzione prima della sua effettiva creazione

Abbiamo sicuramente già notato che una delle caratteristiche che rende questo linguaggio di programmazione non troppo schematico nel suo uso è anche che l'ordine delle definizioni delle funzioni non è importante. Chi ha qualche anno di più e proviene dal Pascal potrebbe avere dei ricordi non del tutto positivi sui suoi rigidi requisiti strutturali. In JavaScript, possiamo

chiamare una funzione anche prima che sia dichiarata formalmente, anche se ci sono delle limitazioni, vediamo il prossimo esempio poco più avanti.

Come detto, possiamo accedere a una funzione che non è ancora definita, questo vale per tutte le dichiarazioni di funzione:

```
assert(typeof prova === "function"); // true
```

```
assert(typeof provaEsp === "undefined", "Ma non possiamo accedere  
alle espressioni di funzione");  
assert(typeof fuFr === "undefined", "Né funzioni freccia");
```

Mentre non è possibile accedere a funzioni definite come espressioni o a funzioni freccia. Infatti *provaEsp* è un'espressione di funzione, e *fuFr* è una funzione freccia:

```
function prova() {}  
  
let provaEsp = function() {};  
let fuFr = (a) => a;
```

Possiamo accedere alla funzione *prova* anche prima di averla definita. Abbiamo già spiegato il perché, quindi, anche prima di iniziare a eseguire la nostra chiamata di asserzione, la funzione *prova* è già esistente.

Gli sviluppatori di JavaScript hanno pensato a questo per rendere le cose più facili agli sviluppatori, per non appesantirli con lunghi lavori relativi alla sistemazione delle funzioni con un ordine sequenziale preciso. Le funzioni esisteranno già nel momento in cui il nostro codice inizia l'esecuzione.

Si noti che questo vale solo per le dichiarazioni di funzioni. Le espressioni di funzione e le funzioni freccia non fanno parte di questo processo e vengono create quando l'esecuzione del programma raggiunge le loro definizioni, quindi di conseguenza, come visto, non potremo accedere in anticipo a *provaEsp* e *fuFr*.

Sovraccarico degli identificatori di funzione

Con i concetti studiati precedentemente arriviamo ad un problema il quale potrebbe riguardarci quando funzioni e variabili omonime incrociano il loro cammino:

```
let prova = 1;  
function prova(){}  

```

prova // 1

In questo esempio, una dichiarazione di variabile e una dichiarazione di funzione hanno lo stesso nome: *prova*. Se esegui questo codice, vedrai che eseguendo la funzione avremo come risultato 1. Eppure solo inizialmente l'identificatore *prova* si riferisce a un numero, dalla seconda riga, *prova* si riferisce a una funzione, quindi il risultato avrebbe dovuto essere un errore.

Questo comportamento è derivato dalla conseguenza dei passaggi effettuati durante la registrazione degli identificatori. Nella seconda fase del processo descritto, le funzioni definite con le dichiarazioni di funzione vengono create e associate ai loro identificatori prima che qualsiasi codice venga valutato; e nella terza fase, le dichiarazioni di variabili vengono elaborate e il valore *undefined* è associato agli identificatori che non verranno trovati. In questo caso, poiché l'identificatore *prova* è stato rilevato nel secondo passaggio quando vengono registrate le dichiarazioni di funzione, il valore *undefined* non viene assegnato alla variabile *prova*. Dopodiché, abbiamo un'istruzione di assegnazione, `let prova = 1`, che assegna il numero 1 all'identificatore *prova*. In questo modo, perdiamo il riferimento alla funzione, l'identificatore si riferisce a un numero.

Durante l'esecuzione effettiva del programma, le dichiarazioni di funzione vengono saltate, quindi la definizione della funzione *prova* non ha alcun impatto sul valore dell'identificatore. Teniamolo bene a mente.

Sollevamento in Javascript

Rimanendo in questo argomento inerente i meccanismi e le sequenze con cui Javascript assorbe le informazioni, parliamo del **sollevamento** (hoisting). Cosa sarà mai? Andiamo con ordine: il sollevamento di JavaScript avviene durante la fase di creazione del contesto di esecuzione che sposta le dichiarazioni di variabili all'inizio dello script.

Nello specifico significa che quando il motore javascript inizializza il codice, tutte le dichiarazioni di variabili che utilizzano *var* vengono portate all'inizio del loro ambito locale, se dichiarate all'interno di una funzione, o all'inizio del loro ambito globale, se dichiarate all'esterno di una funzione, a prescindere da dove la dichiarazione è stata fatta.

Durante l'utilizzo *var*, il tentativo di utilizzare variabili non dichiarate porterà all'assegnazione alla variabile di un valore *indefinito* al momento del sollevamento, quindi, le dichiarazioni di variabili vengono sollevate, ma non le inizializzazioni. Ciò significa che anche se dichiari una variabile in un

punto successivo nel codice, sembrerà come se fosse stata dichiarata all'inizio:

```
console.log(miaVariabile); // Output: undefined
var miaVariabile = 42;
console.log(miaVariabile); // Output: 42
```

L'output della prima riga è *undefined* perché la dichiarazione di *miaVariabile* viene sollevata, ma l'inizializzazione avviene successivamente.

Anche con *let* e *const*, l'uso di variabili non dichiarate porterà a un errore perché rimangono non inizializzate durante l'esecuzione, in pratica le dichiarazioni di variabili vengono anch'esse sollevate, ma a differenza di *var*, non vengono inizializzate con *undefined*. Pertanto, tentare di accedere a tali variabili prima della loro dichiarazione causerà un errore "ReferenceError":

```
console.log(miaVariabile); // Errore: miaVariabile non è definita
let miaVariabile = 42;
console.log(miaVariabile); // Output: 42
```

Anche le dichiarazioni di funzioni vengono sollevate. Questo significa che puoi chiamare una funzione prima della sua dichiarazione nel codice:

```
miaFunzione(); // Output: "Ciao, Mondo!"
function miaFunzione() {
  console.log("Ciao, Mondo!");
}
```

Per evitare confusione e comportamenti inaspettati, è una buona pratica dichiarare e inizializzare le variabili e le funzioni prima del loro utilizzo. Questo rende il tuo codice più leggibile e prevedibile:

```
let variabileInizializzata = "Valore Iniziale";
console.log(variabileInizializzata); // Output: "Valore Iniziale"
```

C'è da sottolineare che le variabili e le dichiarazioni di funzione non vengono tecnicamente "spostate" da nessuna parte, semplicemente bisogna tenere a mente che c'è una precisa sequenza di eventi che dovremmo conoscere.

In questo caso dovremmo prendere l'abitudine di dichiarare e inizializzare le variabili JavaScript prima dell'uso.

Variabili private e chiusure

Arrivati a questo punto dovremmo sapere come funzionano le regole di *scoping* in JavaScript (tradotto: ciò che abbiamo visto fin'ora relativo agli ambiti), quindi ci concentreremo sui contesti di esecuzione e sull'ambiente lessicale. Procediamo come di solito con un esempio.

Dichiariamo una variabile all'interno del costruttore. Poiché l'ambito della variabile è limitato all'interno del costruttore, questa sarà una variabile "privata":

```
function Banca() {  
  let depositi = 0;  
  this.prendiSoldi = function() {  
    return depositi;  
  };  
  this.deposito = function() {  
    depositi++;  
  };  
}
```

```
let banca2 = new Banca();
```

```
assert(banca2.depositi === undefined, "Dati privati non  
accessibili.");  
banca2.deposito();  
assert(banca2.prendiSoldi() === 1, " Siamo in grado di accedere al  
conteggio dei depositi interni.");
```

Test che l'incremento è stato eseguito.

```
let banca3 = new Banca();
```

```
assert(banca3.prendiSoldi() === 0, "Il secondo oggetto banca ottiene  
la propria variabile depositi.");
```

Quando creiamo un nuovo oggetto *banca3* con il costruttore *Banca*, l'oggetto *banca3* ottiene la propria variabile *depositi*.

Durante l'esecuzione del costruttore, vengono create due funzioni e assegnate come proprietà dell'oggetto appena creato (*prendiSoldi* e *deposito*). Come con qualsiasi funzione, queste due funzioni mantengono un riferimento all'ambiente in cui sono state create (l'ambiente *Banca*).

Analizzeremo ora lo stato dell'applicazione dopo che il primo oggetto *Banca* è stato creato. Attraverso la parola chiave *new* evocheremo il costruttore, e ogni volta che invochiamo una funzione di costruzione, creiamo un nuovo ambiente lessicale, che tiene traccia delle variabili locali.

In questo esempio, viene creato un nuovo ambiente *Banca* che tiene traccia della variabile *depositi*.

Inoltre, ogni volta che viene creata una funzione, mantiene un riferimento all'ambiente lessicale in cui è stata creata (tramite una proprietà interna `[[Environment]]`, tradotto: ambiente, area o zona). In questo caso, all'interno della funzione costruttore *Banca*, creiamo due nuove funzioni: *prendiSoldi* e *depositi*, che ottengono un riferimento all'ambiente *Banca*, perché è quello in cui sono stati creati.

Le funzioni *prendiSoldi* e *deposito* vengono assegnate come metodi del nuovo “file oggetto” banca (che, se ricordate dal capitolo precedente, è accessibile tramite la parola chiave *this*). Pertanto, *prendiSoldi* e *deposito* saranno accessibili dall'esterno della funzione di costruzione *Banca*, il che a sua volta porta al fatto che hai effettivamente creato una chiusura attorno alla variabile *depositi*.

I metodi mantengono vivi gli ambienti in cui sono stati creati, mantenendo vive le variabili "private" di ogni istanza.

```
let banca2 = new Banca();  
banca2.deposito()
```

```
let banca3 = new Banca();
```

I metodi di ogni istanza creano chiusure attorno alle variabili di istanza "private".

Quando creiamo un altro oggetto *Banca*, l'oggetto *banca3*, l'intero processo viene ripetuto. Ogni oggetto creato con il costruttore *Banca* ottiene i propri metodi (il metodo *banca2.prendiSoldi* è diverso dal metodo *banca3.prendiSoldi*) che si chiudono attorno alle variabili definite quando è stato invocato il costruttore. Queste variabili "private" sono accessibili solo tramite metodi oggetto creati all'interno del costruttore e non direttamente!

Ora vediamo come vanno le cose quando si effettua la chiamata *banca3.prendiSoldi()*.

Prima di effettuare la chiamata *banca3.prendiSoldi()*, il nostro motore JavaScript sta eseguendo il codice globale, che è anche l'unico contesto nello stack di esecuzione. Allo stesso tempo, l'unico ambiente lessicale attivo è l'ambiente globale, l'ambiente associato all'esecuzione globale nel contesto di attenzione.

Quando si effettua la chiamata *banca3.prendiSoldi()*, si chiama il metodo *prendiSoldi* dell'oggetto *banca3*. Poiché ogni chiamata di funzione causa la

creazione di un nuovo contesto di esecuzione, un nuovo contesto di esecuzione *prendiSoldi* viene creato e inviato allo stack di esecuzione. Questo porta anche alla creazione di un nuovo ambiente lessicale *prendiSoldi*, che viene normalmente utilizzato per tenere traccia delle variabili definite in questa funzione. Inoltre, l'ambiente lessicale *prendiSoldi*, come ambiente esterno, ottiene l'ambiente in cui è stata creata la funzione *prendiSoldi*, l'ambiente *Banca* che era attivo quando l'oggetto *banca3* è stato costruito.

Ora vediamo come vanno le cose quando proviamo a ottenere il valore della variabile *depositi*. Per prima cosa, viene consultato l'ambiente lessicale *prendiSoldi* attualmente attivo. Poiché non abbiamo definito alcuna variabile nella funzione *prendiSoldi*, questo ambiente lessicale è vuoto e la nostra variabile non verrà trovata lì. Successivamente, la ricerca continua nell'ambiente esterno dell'attuale ambiente lessicale: nel nostro caso, l'ambiente *Banca* è attivo durante la costruzione dell'oggetto *banca3*. Questa volta, l'ambiente *Banca* ha un riferimento alla variabile *depositi* e la ricerca è fatta. Un po' complicato nel seguirlo testualmente, guardando il codice sarà tutto più chiaro.

Adesso focalizziamo la nostra attenzione alle variabili cosiddette "private". Cosiddette perché queste variabili non sono proprietà private dell'oggetto, ma sono variabili mantenute in vita dai metodi dell'oggetto creati nel costruttore. Diamo un'occhiata a un interessante effetto collaterale che deriva da tutto ciò.

Da sapere sulle variabili private

Oltre a tutto quello che abbiamo visto finora, dobbiamo ricordarci (e se non ce lo ricordiamo allora lo sapremo adesso) che con questo linguaggio di programmazione le proprietà create e attribuite ad un certo oggetto le possiamo anche assegnare ad un altro oggetto. Per capirci meglio riproponiamo il codice visto in precedenza corretto in maniera che potremmo accedere alle variabili private tramite funzioni, non tramite oggetti come in precedenza:

```
function Banca() {let depositi = 0;
  this.prendiSoldi = function() {return depositi;
  };
  this.depositi = function() {depositi ++;
  };
```

```
}  
let banca1 = new Banca(); banca1.depositi();  
let ladro = {};  
ladro.prendiSoldi = banca1.prendiSoldi;
```

Rende la funzione *prendiSoldi* di *banca1* accessibile tramite il *ladro* e dopo, nella prossima riga, verifica che possiamo accedere alla “presunta” variabile privata di *banca1*:

```
assert(ladro.prendiSoldi() === 1, "Il ladro ha rubato i soldi!");
```

Questo listato modifica il codice sorgente in modo da assegnare al metodo *banca1.prendiSoldi* un oggetto *ladro* completamente nuovo. Quindi, quando chiameremo la funzione *prendiSoldi* sull'oggetto *ladro*, testeremo che potremmo accedere al valore della variabile creata quando *banca1* è stato istanziato. Stiamo parlando del valore *depositi*.

Questo esempio dimostra che non ci sono variabili oggetto private in JavaScript, e che possiamo servirci delle chiusure create dai metodi dell'oggetto per superare queste barriere. E, nonostante molti sviluppatori trovano utile questo modo di nascondere le informazioni potremmo comunque accedere alle variabili "private" tramite le funzioni, anche se quella funzione fosse collegata ad un altro oggetto.

Chiusure e callback per animare oggetti?

Andiamo oltre, osserviamo adesso un semplice esempio di animazione con i timer di richiamata. Faremo animare un oggetto, usando una chiusura in un file *setInterval* richiamato finquando l'oggetto animato non raggiungerà la posizione designata:

```
<!DOCTYPE html>  
<html>  
<style>  
  #scatola {  
    width: 300px;  
    height: 300px;  
    position: relative;  
    background: green;  
  }  
  
  #scatolino {  
    width: 80px;  
    height: 80px;
```

```

    position: absolute;
    background-color: black;
}
</style>
<body>
  <p>
    <button onclick="animAz()">Ciccami</button>
  </p>
  <div id="scatola">
    <div id="scatolino"></div>
  </div>
  <script>
    let id = null;
    function animAz() {
      let cubo = document.getElementById("scatolino");
      let muovi = 0;
      clearInterval(id);
      id = setInterval(cornice, 15);
      function cornice() {
        if (muovi == 220) {
          clearInterval(id);
        } else {
          muovi++;
          cubo.style.top = muovi + 'px';
          cubo.style.left = muovi + 'px';
        }
      }
    }
  </script>
</body>
</html>

```

Forse non ne abbiamo parlato a sufficienza nel capitolo, ma, anche analizzando l'esempio appena esposto, noteremo che potremo utilizzare le chiusure per semplificare l'animazione di più oggetti sulle nostre pagine.

Ma ora considereremo gli ambienti lessicali, ogni volta che chiamiamo la funzione *cornice*, viene creato un nuovo ambiente lessicale della funzione creata che tiene traccia dell'insieme di variabili importanti per quell'animazione (*elementId*, *cubo*, l'elemento che viene animato, l'ID del timer che esegue l'animazione). Quell'ambiente lo manterrà in vita finché c'è almeno una funzione che lavora con le sue variabili attraverso le chiusure. Questo ci consente di evitare il problema di mappare manualmente il callback e le variabili attive, quindi semplificheremo significativamente il nostro codice.

Tuttavia si pone un altro problema, Il listato appena visto violerebbe una policy di sicurezza conosciuta come "Content Security Policy" (CSP) o

"*Política de seguridad del contenido*" in italiano. Una CSP è una serie di regole che un'applicazione web può implementare per mitigare potenziali vulnerabilità legate all'inclusione di risorse esterne o all'esecuzione di script non attendibili all'interno di una pagina web. Questo serve a proteggere gli utenti da attacchi come l'inclusione di script dannosi (inclusi gli attacchi XSS) o l'accesso a risorse non autorizzate.

Nel codice stiamo utilizzando l'attributo *onclick* direttamente nell'HTML per chiamare una funzione JavaScript quando il pulsante "Cliccami" viene premuto. Questo può essere considerato un potenziale rischio di sicurezza, poiché permette l'esecuzione di script arbitrari all'interno della pagina web. Una CSP ben configurata potrebbe impedire l'esecuzione di script esterni non fidati o addirittura l'uso dell'attributo *onclick* direttamente nell'HTML.

Per mitigare questa violazione potenziale della CSP, dovremmo spostare la gestione degli eventi e l'inclusione di script all'interno del nostro file JavaScript invece di utilizzare direttamente gli attributi HTML come *onclick*. Inoltre, dovremmo assicurarci di avere una CSP adeguata configurata per il nostro sito web, specificando quali risorse e origini sono autorizzate a essere caricate ed eseguite all'interno della pagina.

Vediamo quindi un approccio diverso per ottenere lo stesso effetto precedente, ma in sicurezza:

```
<!DOCTYPE html>
<html>
<head>
<style>
  #scatola {
    width: 300px;
    height: 300px;
    position: relative;
    background: green;
  }

  #scatolino {
    width: 80px;
    height: 80px;
    position: absolute;
    background-color: black;
    transition: all 2s ease; /* Aggiungiamo una transizione CSS */
  }
</style>
</head>
<body>
```

```

<p>
<button onclick="avviaAnimazione()">Cliccami</button>
</p>
<div id="scatola">
<div id="scatolino"></div>
</div>
<script>
function avviaAnimazione() {
  let cubo = document.getElementById("scatolino");
  cubo.style.top = "220px"; // Impostiamo la posizione desiderata
  cubo.style.left = "220px";
}
</script>
</body>
</html>

```

In questo esempio, utilizziamo la proprietà CSS *transition* per creare un'animazione fluida quando la posizione di "scatolino" viene modificata. L'animazione è controllata semplicemente impostando direttamente la posizione desiderata quando viene cliccato il pulsante "Cliccami". Questo approccio è più sicuro e consigliato rispetto all'uso di *setInterval* e *clearInterval*.

Tuttavia, se desideri animare più oggetti contemporaneamente o avere animazioni più complesse, puoi utilizzare librerie JavaScript come jQuery o un framework di animazione come *GreenSock (GSAP)* che semplificano notevolmente la gestione delle animazioni, comprese le chiusure per oggetti multipli.

Ecco un esempio di come potresti utilizzare jQuery per animare più oggetti:

```

<!DOCTYPE html>
<html>
<head>
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<style>
  .scatola {
    width: 300px;
    height: 300px;
    position: relative;
    background: green;
    margin: 10px;
  }

  .scatolino {
    width: 80px;
    height: 80px;
    position: absolute;
    background-color: black;
  }

```

```

    transition: all 2s ease;
  }
</style>
</head>
<body>
  <div class="scatola">
    <div class="scatolino"></div>
  </div>
  <div class="scatola">
    <div class="scatolino"></div>
  </div>
  <div class="scatola">
    <div class="scatolino"></div>
  </div>
<script>
$(".scatolino").click(function() {
  $(this).animate({ top: "220px", left: "220px" }, 2000);
});
</script>
</body>
</html>

```

Utilizziamo jQuery per selezionare tutti gli elementi con classe "scatolino" e applicare un'animazione quando vengono cliccati. Questo consentirebbe di gestire facilmente più oggetti in una pagina.

Questo è tutto ciò che abbiamo da dire su chiusure e ambiti. Nel prossimo capitolo esploreremo due concetti: generatori e promesse che ci saranno d'aiuto durante la scrittura di codice asincrono.

Quiz & esercizi

1) Completare il listato in modo che il saluto sia mostrato:

```

function testo() {
  let saluto = "Buongiorno";
  function chiusa() {alert(saluto);}
}

```

2) Indicare la chiusura e lo scope del listato qui sopra.

3) Troviamo il modo di far aprire una finestra che ci faccia vedere il numero calcolato nel listato, bypassando la chiusura:

```

function numero() {
  const chiuso = Math.trunc(Math.random()*200)
  return function inner() {
    console.log(`Il numero è ${chiuso}.`)
  }
}

```

```
}}
```

4) Completare questo programma con il callback:

```
function m(x,y){  
    return x*y;  
}  
function prova(op){  
    let nm = 3;  
    let nn = 3;  
    alert(op(nm,nn));  
}
```

5) *var numero = 10* , *const numero = 10* , *let numero = 10* , quali tra queste dichiarazioni dovrei adottare se il numero 10 nel mio programma probabilmente sarà soggetto a modifiche?

6) Dopo aver dichiarato tre numeri, dal più basso al più alto, quale sarà assegnato ad un'eventuale funzione con la dichiarazione del nome uguale al nome della variabile?

7) Quando Javascript attiva il sollevamento dovrò alzare il mio computer? (se la risposta è: sì ...ripassa il capitolo :D)

8) Per evitare il sovraccarico degli identificatori della funzione dovrò: 1) controllare bene i valori 2) evitare funzioni dichiarative 3) utilizzare var al posto di const 4) controllare i nomi assegnati 5) utilizzare un callback

9) Crea una funzione *creaContatore* che restituisca una funzione interna che, ogni volta che viene chiamata, incrementa un contatore e lo restituisce.

10) Crea una funzione *sommaParziale* che restituisca una funzione interna che, ogni volta che viene chiamata con un numero, restituisce la somma parziale dei numeri passati finora.

11) Crea una funzione *memoizza* che accetti una funzione e la memorizzi in modo che, se viene chiamata con gli stessi argomenti, restituisca il risultato memorizzato invece di calcolarlo nuovamente.

12) Crea una funzione *creaPersona* che accetti un nome e una età e restituisca un oggetto con metodi per ottenere il nome e l'età, ma che siano accessibili solo tramite i metodi.

13) Crea una funzione *eseguiInSequenza* che accetti un array di funzioni e le esegua una dopo l'altra in ordine. Assicurati che ogni funzione venga chiamata solo quando la funzione precedente ha completato la sua esecuzione.

14) Crea una funzione *creaTimer* che accetti un intervallo di tempo in millisecondi e restituisca una funzione interna che, quando chiamata, attenda l'intervallo di tempo specificato prima di eseguire una determinata azione.

15) Crea una funzione chiamata *calcola* che accetti un numero variabile di argomenti e restituisca la somma di tutti gli argomenti.

16) Spiega il comportamento di sollevamento (hoisting) delle variabili in JavaScript e mostra come il seguente codice funziona correttamente:

```
console.log(messaggio);  
let messaggio = "Ciao, mondo!";  
console.log(messaggio);
```

17) Crea un modulo (una funzione che restituisce un oggetto) che utilizzi variabili private per conservare un contatore. Fornisci metodi pubblici per aumentare e ottenere il valore del contatore.

18) Crea una funzione *eseguiOperazione* che accetti due numeri e una funzione di callback e utilizzi la callback per eseguire un'operazione tra i due numeri.

19) Crea una funzione *leggiFile* che accetti il nome di un file e una funzione di callback. Simula la lettura di un file in modo asincrono utilizzando *setTimeout* e poi chiama la callback con il contenuto del file.

20) Crea un oggetto calcolatrice con metodi per eseguire operazioni matematiche di somma, sottrazione, moltiplicazione e divisione tra due numeri.

Riassunto

Siamo partiti dal concetto di chiusura e dalla differenza tra i contesti di esecuzione locali e globali, molto importanti da capire per evitare fastidiosi errori o bug durante la stesura del codice. Abbiamo anche imparato che quando una funzione viene chiamata in un momento diverso dalla sequenza degli eventi legati al programma avremo un callback (sostanzialmente un richiamo). Importante è anche la distinzione tra le varie dichiarazioni di variabile possibili, tra *var*, *let*, *const*, cioè, rispettivamente per un valore non fisso, per un valore valido in un certo ambito e per un valore fisso. Con il sollevamento abbiamo capito che Javascript conosce in anticipo le variabili dichiarate e che le variabili private hanno un loro ambiente di riferimento specifico normalmente inaccessibile dall'esterno.

8 - Codice Asincrono

Funzioni e generatori
Le promesse in Javascript
Combinare generatori e promesse
Async e await

Codice asincrono, indubbiamente il titolo non promette nulla di buono, ma forse potrà sollevarci il pensiero che per ora non parleremo delle espressioni regolari. La maggior parte di noi ha iniziato a programmare scrivendo codice procedurale, cioè la classica impostazione dove le procedure e le funzioni sono separate e definite in subrutine, ottenendo anche una certa chiarezza schematica nella struttura del programma. Di contro bisogna affermare che, come già accennato, Javascript è un linguaggio a thread singolo, cioè in estrema sintesi esegue il codice seguendo le istruzioni riga per riga. Ora, distinguiamo per bene tra multi-thread, dove un elenco di attività viene smistato su più processori, e codice asincrono, dove diverse liste di istruzioni vengono processate in base alle esigenze, bypassando di fatto il single thread. Questo è proprio quello che andremo ad imparare con i generatori e le promesse.

Partiamo dai primi i quali sono un tipo speciale di funzione. Mentre una funzione standard produce al massimo un singolo valore quando esegue il suo codice dall'inizio alla fine, i generatori producono più valori, in base alla richiesta, sospendendo la loro esecuzione tra queste richieste.

I generatori esistono anche in PHP, Python e C#, quindi è doppiamente importante conoscerli bene. Nonostante queste ottime premesse potrebbe capitare che alcuni li considerino una caratteristica del linguaggio marginale, non molto utilizzata dal programmatore medio. Al contrario invece ci serviranno per semplificare i loop contorti e anche per sfruttare la loro capacità di sospendere e riprendere l'esecuzione, che, tra l'altro potrà anche aiutarci a scrivere codice asincrono di ottima qualità.

Stesso discorso si può fare con le promesse, difatti, anche esse sono un tipo di oggetto che ci aiuterà a lavorare bene anche con il codice asincrono. Essenzialmente sono come un segnaposto per un valore che ancora non abbiamo se non in un momento successivo. Sono particolarmente utili per lavorare con più passaggi asincroni.

Il passaggio successivo sarà quello di vedere come combinarli assieme per semplificare notevolmente le nostre operazioni, difatti entrambi le tecniche coesistono senza problemi. Praticamente con le promesse otterremo il risultato di un'operazione asincrona non ancora disponibile, anche se dovremo scrivere il codice in modo asincrono, per listati di grandi dimensioni potrebbe essere un problema, quindi l'esigenza di scrivere codice asincrono in modo sincrono viene ovviata proprio dai generatori. Iniziamo vedendo come tutto questo agisce in concreto, immaginiamo di essere uno sviluppatore che lavora su di un sito da cui dobbiamo estrapolare alcuni semplici dati che sono memorizzati su un server remoto, codificato in JSON (la libreria più popolare di Javascript, vedi il capitolo 11 per i dettagli), vediamo un esempio dove utilizzeremo *try catch* per visualizzare l'errore prodotto:

```
let json = "{prova}";
try {
  let utente = JSON.parse(json);
  alert(utente.nome); // non esiste
} catch (errore) {
  alert("C'è un errore.");
  alert(errore.nome);
}
```

La sintassi appena esposta è abbastanza chiara, in caso ci fosse un errore in uno dei passaggi, potremmo facilmente individuarlo nel blocco *catch*. Oltre a questo impareremo che ottenere dati da un server è un'operazione di lunga durata e poiché JavaScript si basa su un modello di esecuzione a thread singolo, bloccheremo la nostra interfaccia utente fino al termine dell'operazione. Per risolvere questo problema, potremmo riscriverlo con i callback, che verranno richiamati quando un'attività finisce, senza bloccare l'interfaccia utente. Tuttavia se dovessimo gestire tutti i diversi errori possibili il listato diventerebbe disordinato e confusionario, e poi esiste una soluzione. È qui che i generatori e le promesse possono venirci incontro risolvendo i nostri problemi. Combinandoli, possiamo trasformare il codice in qualcosa di molto più funzionale:

Una funzione generatore viene definita inserendo un file asterisco subito dopo la parola chiave della funzione. Possiamo usare la nuova parola chiave *yield* nelle funzioni del generatore:

```
async(function*() {try {  
    const prova = produce getJSON("prova.json");  
    const utenti = yield getJSON(prova[0].utentiUrl);  
    const eta = yield getJSON(utenti[0].datiUrl);  
}  
catch(err) {  
}  
});
```

Messo così, su due piedi, non è affatto chiaro, e la grammatica non è per niente familiare, difatti * di solito è un operatore matematico e *yield* risulta del tutto nuovo. Di contro il codice sembra chiaro e conciso, l'ovvia conclusione è che ci toccherà studiare ancora questa nuova modalità. Partiamo dalla funzione generatore.

La funzione generatore

Questo tipo di funzione, che, come abbiamo scoperto si attiva con la sintassi *function**, ci fa capire subito che siamo di fronte a qualcosa di completamente nuovo e che queste cosiddette *funzioni generatore* sono significativamente diverse da tutte le altre che conosciamo. Ma andiamo con ordine, un generatore è una funzione che genera una sequenza di valori, ma non tutti in una volta come farebbe una normale funzione, al contrario, agirà in base alla richiesta. Dobbiamo chiedere esplicitamente al generatore un nuovo valore ed esso risponderà con un valore o ci informerà che non ne ha nessuno da produrre. Una cosa molto importante e utile è che dopo che un valore è stato prodotto, una funzione generatore non termina la sua esecuzione come le altre, è semplicemente sospesa. Quindi, quando arriva una richiesta di un altro valore, il generatore riprende da dove era stato interrotto. L'istruzione *yield* definirà il momento o il punto in cui l'esecuzione può ripartire o stopparsi.

Nell'esempio di seguito vedremo un esempio basico di utilizzo di un generatore per creare un elenco di numeri:

```
function* elenco() {  
    yield 1;  
    yield 2;  
    yield 3;  
}
```



```
}  
  
let gen = elenco();  
for(let valore of gen) {  
    alert(valore);  
}
```

Osserviamo che attraverso un generatore elenchiamo una piccola sequenza di numeri. Per farlo useremo la parola chiave *function* seguita da un asterisco. A questo punto saremo in grado di utilizzare la nuova parola chiave *yield* all'interno del corpo del generatore per promuovere dei valori individuali, infatti creiamo un generatore chiamato *elenco* che produce una sequenza di numeri fino a tre che poi elencheremo con un ciclo *for-of*:

```
for(let valore of gen) {alert(valore);
```

Evocare un generatore non esegue la funzione del generatore; invece crea un oggetto chiamato iteratore, perciò, essenzialmente effettuare una chiamata a un generatore non significa che il corpo della funzione del generatore verrà eseguito. Viene invece creato un oggetto iteratore, attraverso il quale potremmo comunicare con il generatore. Ad esempio, possiamo utilizzare l'iteratore per richiedere valori aggiuntivi. Assodato che il generatore è un iteratore potremmo ottenere dei vantaggi da questo, come la sintassi *Spread* (...), già vista nel capitolo 6.

Ma osserviamo un'altra possibilità:

```
function* elenco() {  
    yield 1;  
    yield 2;  
    yield 3;  
}  
  
let seq = [0, ...elenco()];  
alert(seq);
```

Come possiamo vedere, quando chiamiamo un generatore, viene creato un nuovo iteratore:

```
let lista = {  
    from: 5,  
    to: 10,  
    [Symbol.iterator]: function*() {  
        for (let valore = this.from; valore <= this.to; valore++) {
```

```

        yield valore;
    }
},
};

```

```

alert([...lista]);

```

L'iteratore viene utilizzato per controllare l'esecuzione del generatore. In questo caso utilizzeremo *Symbol.iterator*, il quale specifica l'iteratore predefinito per un oggetto in un ciclo *for of* con tutti i suoi vantaggi, lo vedremo nel prossimo paragrafo.

```

function* lista1() {
    yield 7;
    yield 8;
    yield 9;
}

```

```

function* lista2() {
    yield 6;
    yield* lista1();
    yield 10;
}

```

```

let x = lista2();

```

```

console.log(x.next());
console.log(x.next());
console.log(x.next());
console.log(x.next());
console.log(x.next());
console.log(x.next());

```

Qui vediamo che potremo anche delegare il controllo ad un'altra funzione generatrice dall'interno della funzione principale.

Dopodichè noteremo che il metodo *next* invia il valore al generatore:

```

console.log(x.next());

```

Questo risveglia il generatore dalla sospensione, esso continua da dove era stato interrotto, eseguendo il suo codice fino a quando non viene raggiunto un altro valore intermedio. Questo sospende il generatore e produce un numero che viene visualizzato nella nostra finestra.

Finalmente, quando chiamiamo il metodo successivo per la sesta volta, il generatore riprende la sua esecuzione. Ma questa volta non c'è più codice da

eseguire, quindi il generatore restituisce un oggetto con valore impostato su *undefined* e impostato su *true*, segnalando che ha terminato il suo lavoro.

Abbiamo visto che un iteratore ha il metodo *next()*, che ritorna il seguente elemento dell'iterazione. Questo elemento ottenuto, è un oggetto con due proprietà: *done* e *value*:

```
function generaIteratore(array) {let vaiAvanti = 0;
    return { next: function() { return vaiAvanti < array.length ?
        {value: array[vaiAvanti++], done: false} : {done: true};
    }
};
}
```

```
let testo = generaIteratore(['Eccomi qui', 'sono il', 'tuo
gatto!']);
console.log(testo.next().value);    console.log(testo.next().value);
console.log(testo.next().value);
```

Creare manualmente un iteratore, come mostrato sopra, potrebbe essere complicato, l'abbiamo già detto, in alternativa si possono utilizzare i generatori. Essi permettono di scrivere un generatore *function** che tiene traccia della propria posizione e possiede il metodo *next()*. I generatori ritornano un oggetto “*generatore*”. La parola chiave *yield* viene usata per sospendere e riprendere l'esecuzione di una funzione generatore:

```
function* numeri() {
let lista = 5;
yield lista++;
yield lista++;
yield lista++;
yield lista++; }
let num = numeri();
console.log(num.next().value);
console.log(num.next().value);
console.log(num.next().value);
console.log(num.next().value);
```

Parliamo di oggetti iterabili, un oggetto lo è se può ripetere i suoi elementi e contemporaneamente attuare un metodo iteratore. Tutto ciò verrà eseguito con una proprietà *Symbol.iterator*:

```
let oggettoIter = {};
oggettoIter[Symbol.iterator] = function* () {
    yield 3;
    yield 4;
```

```

    yield 5;
  };
  for (let numeri of oggettoIter) {
    console.log(numeri);
  }

```

Gli iterabili predefiniti di JavaScript sono: *String*, *Array*, *TypedArray*, *Map*, *Set*. Tutti hanno il metodo *Symbol.iterator* che specifica l'iteratore predefinito usato dall'operatore per un ciclo *for-of*:

```

let testoNomegatto = 'Briciola';
for (let lettere of testoNomegatto) {
  console.log(lettere);
}

```

Si può utilizzare *Iterable* come *Iterator* così:

```

let txtMiagola = 'Miao';
let itCat = txtMiagola[Symbol.iterator]();
console.log(itCat.next().value);
console.log(itCat.next().value);
console.log(itCat.next().value);
console.log(itCat.next().value);

```

Abbiamo anche gli operatori JavaScript da usare, come l'operatore *for* (*.. of ..*), il quale crea un ciclo su tutti gli oggetti iterabili come (*Array*, *Map*, *Set*, *String*, ...), invocando delle istruzioni da noi definite:

```

let ripetere = 'Topolino';
for (let txt of ripetere) {
  console.log(txt);
}

```

Spread permette ad un'espressione di espandersi lì dove avremo più argomenti, è definito da tre puntini come abbiamo già visto:

```

function prodottoNumeri(primo, secondo, terzo) {
  console.log(primo * secondo * terzo);
}
let trenta = [3, 2, 5];
prodottoNumeri(...trenta);

```

Per finire vediamo l'operatore *yield**, diverso da *yield*, è usato per delegare ad un altro generatore oppure ad un altro oggetto iterabile, come abbiamo già visto in precedenza, ma è sempre bene ribadirlo:

```
function* numeri() {
  yield 6;
  yield 9;
}

function* altrinum() {
  yield 3;
  yield* numeri();
  yield 12;
}

let iter = altrinum();

console.log(iter.next().value);
console.log(iter.next().value);
console.log(iter.next().value);
console.log(iter.next().value);
```

Adesso scopriamo le altre possibilità offerte dai generatori.

Sfruttare i generatori

Impareremo che, per avere la situazione del nostro programma sotto controllo dovremo mantenere l'ordine. Ad esempio, quando si creano determinati oggetti spesso è necessario assegnare un ID univoco a ciascuno di essi e, proprio per questo potremo utilizzare un generatore:

```
let id = 0;

function* generaID() {
  while(true) {
    yield id++;
  }
}

const idGenerator = generaID();

const obj1 = { name: "oggetto 1" };
const obj2 = { name: "oggetto 2" };
const obj3 = { name: "oggetto 3" };

obj1.id = idGenerator.next().value;
obj2.id = idGenerator.next().value;
obj3.id = idGenerator.next().value;

console.log(obj1); // { name: "oggetto 1", id: 0 }
console.log(obj2); // { name: "oggetto 2", id: 1 }
console.log(obj3); // { name: "oggetto 3", id: 2 }
```

In questo listato, definiamo una funzione generatore chiamata "generaID" che crea un nuovo ID ogni volta che viene invocata. Iniziamo con un ID impostato su 0 e il generatore restituisce il valore dell'ID e lo incrementa ogni volta che viene chiamato. La funzione generatore è creata utilizzando la sintassi *function**, e all'interno del ciclo *while(true)* viene mantenuta in esecuzione in modo indefinito. Ogni volta che chiamiamo il generatore utilizzando *next()*, otteniamo il prossimo valore dell'ID.

Creiamo quindi una variabile *idGenerator* che rappresenta la funzione generatore e tre oggetti *obj1*, *obj2*, e *obj3*. Assegniamo a ciascun oggetto un valore ID unico ottenuto chiamando *idGenerator.next().value*. Come risultato, ogni oggetto ha un ID univoco.

È possibile anche inviare dati al generatore passando argomenti al metodo *next()*. Questo risveglia il generatore dalla sospensione e riprende la sua esecuzione, utilizzando il dato passato come valore dell'intera espressione *yield*. Ad esempio:

```
function* generatore() {  
  while(true) {  
    let valore = yield null;  
    console.log(valore);  
  }  
}
```

```
let z = generatore();  
z.next(1);  
z.next(2);
```

In questo esempio, la prima chiamata a *next(1)* non registra alcun valore, poiché il generatore non aveva istruzioni per raccogliere dati inizialmente.

Adesso vedremo qualcosa che ci aiuterà nella stesura del codice quando ci saranno tante funzioni con compiti da eseguire e da attendere per procedere con il programma.

Problemi con i callback

Utilizziamo codice asincrono perché non vogliamo bloccare l'esecuzione della nostra applicazione durante lo svolgimento di attività a lungo adempimento, parliamo di tutto quello che potrebbe richiedere dei tempi relativamente lunghi, come ad esempio 20 secondi, a dispetto della maggior parte di ciò che succede sul nostro schermo il quale si esplicita nel momento stesso in cui eseguiamo un'azione (come il click su un link). Normalmente

risolviamo questo problema con i callback: per un'attività di lunga durata utilizzeremo una richiamata che viene invocata quando l'attività è finalmente completata.

Ad esempio, il recupero di un file JSON (tratteremo di JSON nel capitolo 11, dai un'occhiata se vuoi approfondire) da un server è un'attività di lunga durata, durante la quale non vogliamo che l'applicazione non risponda ai nostri utenti. Pertanto, forniamo una richiamata che verrà avviata al termine dell'attività:

```
getJSON("data/gatto.json", function() {});
```

Naturalmente, durante questa attività di lunga durata, possono verificarsi errori. E il problema con i callback è che non puoi usare costrutti di linguaggio incorporati, come *try-catch* visto in precedenza:

```
try {getJSON("data/gatto.json", function() {  
});  
} catch(a) {/ Qui arriveranno gli errori /}
```

Ciò accade perché il codice che richiama il callback di solito non viene eseguito nella stessa fase del ciclo di eventi del codice che avvia l'attività di lunga durata. Di conseguenza, i bug di solito si perdono. E questo è uno dei problemi principali con i callback assieme alla gestione degli errori.

Purtroppo ve ne sono anche altri: eseguire sequenze di passaggi o passaggi in parallelo è complicato, sconsigliato e fonte di problemi:

```
function stupida() {  
  setTimeout(() => {  
    setTimeout(() => {  
      setTimeout(() => {  
        setTimeout(() => {  
          //qui si fa qualcosa  
        }, 5000);  
      }, 3000);  
    }, 1500);  
  }, 8000);  
}
```

Quindi l'introduzione delle promesse è fondamentale e ci servirà in tutti quei casi che abbiamo appena menzionato.

Abbasso i callback W le Promesse

Le promesse in JavaScript sono state pensate per delineare operazioni non completate inizialmente, ma che lo saranno in futuro. Nella stesura del codice ci affidiamo molto a calcoli asincroni, di cui non abbiamo ancora i risultati, ma lo saranno in un momento successivo. In quel caso ci affideremo a una promessa, che è un segnaposto per un valore che non abbiamo ora ma che avremo in seguito; è una garanzia che prima o poi conosceremo il risultato di un calcolo asincrono. Se manteniamo la promessa, il risultato sarà un valore. Se si verifica un problema, il nostro risultato sarà un errore. Un ottimo esempio di utilizzo delle promesse è il recupero dei dati da un server; promettiamo che alla fine avremo i dati, ma c'è sempre la possibilità che si verifichino problemi. Creare una nuova promessa è facile:

```
let promessa = new Promise(function(resolve,reject){
  let stato = false;
  if(stato)
    resolve("ci siamo");
  else
    reject("non ci siamo");
});
promessa.then(function(sitIncarichi){
  console.log('Situazione attuale: ' + sitIncarichi);
}).catch(function(sitIncarichi){
  console.log('Situazione attuale: ' + sitIncarichi);
});
```

Ogni promessa ha una funzione *then()* che sarà eseguita al completamento della promessa. Nell'esempio sopra si utilizza la funzione *then* sulla *promessa* per agganciare il testo derivato da *resolve*.

Promesse concatenate

Abbiamo già visto come gestire una sequenza di passaggi interdipendenti porta a una sequenza di richiami profondamente annidata e difficile da mantenere. Le promesse sono un passo verso la risoluzione di questo problema, perché hanno la capacità di essere concatenate.

In precedenza nel capitolo, abbiamo visto come, utilizzando il metodo *then* su una promessa, possiamo registrare un callback che verrà eseguito se una promessa viene risolta con successo. Quello che non non abbiamo approfondito è che la chiamata al metodo *then* restituisce anche una nuova promessa. Così non c'è nulla che ci impedisca di concatenare tutti i metodi che vogliamo:


```

let obiettivo = 6;

function processElement(a) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log(a);
      if (a === obiettivo) {
        resolve(a);
      } else {
        reject();
      }
    }, 1100);
  });
}

[1, 2, 4, 6, 9]
  .map(processElement)
  .reduce((chain, promise) => chain.then(() => promise),
Promise.resolve())
  .then((a) => console.log('L'obiettivo è: ' + a))
  .catch(() => console.log('Problema!'));

```

Definiamo una variabile `obiettivo` con un valore di 6, che rappresenta l'obiettivo che vogliamo raggiungere nell'array, poi creiamo una funzione *processElement(a)* che prende un parametro *a* e restituisce una promessa. Questa funzione è responsabile di effettuare il lavoro su ciascun elemento dell'array e restituire una promessa che verrà risolta o rigettata in base al risultato del lavoro.

All'interno di *processElement(a)*, creiamo una nuova promessa usando *new Promise*. All'interno del costruttore di promesse, eseguiamo un'operazione asincrona utilizzando *setTimeout*. Questa operazione rappresenta il lavoro da svolgere su ciascun elemento dell'array.

Nel blocco di codice all'interno di *setTimeout*, controlliamo se *a* è uguale all'obiettivo che stiamo cercando (6). Se è uguale, risolviamo la promessa con *resolve(a)*, altrimenti la rigettiamo con *reject()*.

Ogni promessa restituita da *processElement* rappresenta il risultato del lavoro su un elemento dell'array. Quindi, avremo un array di promesse corrispondenti agli elementi dell'array originale [1, 2, 4, 6, 9]. Usiamo il metodo *map* sull'array originale per applicare la funzione *processElement* a ciascun elemento. Questo crea un nuovo array che contiene le promesse risultanti. Utilizziamo poi il metodo *reduce* sull'array di promesse per concatenarle in una catena di promesse. La catena inizia con *Promise.resolve()*, che è una promessa risolta.

Nella funzione di riduzione (*chain, promise*) => *chain.then()* => *promise*), stiamo dicendo che vogliamo concatenare ciascuna promessa successivamente usando il metodo *then*. Mentre *chain* rappresenta la catena di promesse fino a quel punto, e *promise* è la promessa successiva nell'array. Alla fine, chiamiamo *then* sulla catena di promesse per gestire il risultato finale. Se tutte le promesse vengono risolte con successo, la funzione *success* viene chiamata con il valore risultante. In caso contrario, la funzione *catch* viene chiamata se una qualsiasi delle promesse viene rigettata.

In questo modo, il codice esegue il lavoro su ciascun elemento dell'array in sequenza, attendendo la risoluzione o il rigetto di ciascuna promessa prima di passare all'elemento successivo. Alla fine, otteniamo il risultato dell'obiettivo o gestiamo un problema se una delle promesse viene rigettata. La scrittura di tale codice per mezzo di callback standard comporterebbe una loro sequenza profondamente annidata. Identificare l'esatta serie di passaggi non sarebbe facile e avremmo problemi se dovessimo aggiungere uno step in più da qualche parte nel codice.

Quando si tratta di sequenze di passaggi asincroni, può verificarsi un errore in uno qualsiasi di questi. Sappiamo già che possiamo fornire un secondo callback di errore alla chiamata *then* oppure possiamo concatenare una chiamata *catch* che accetta un callback di errore. Quando ci preoccupiamo solo del successo o fallimento dell'intera sequenza di passaggi, fornire ad ognuno di essi una speciale gestione degli errori potrebbe essere un po' troppo lungo. Tuttavia, come abbiamo visto, esiste una soluzione: possiamo sfruttare il metodo di cattura con il metodo *catch*:

```
unaPromessa.then(risultato => {  
  // Fa qualcosa con il risultato  
}).catch(errore => {  
  // Gestisce l'errore  
  fail("Errore: " + errore);  
});
```

Se si verifica un errore in una delle promesse precedenti, *catch* lo rileva. Se non si verifica, il flusso del programma continua attraverso di esso, senza ostacoli.

In questo modo, avere sotto controllo una sequenza di passaggi nel codice è molto più piacevole e gestibile con le promesse rispetto ai normali callback.

In attesa delle promesse

Promise oltre ad aiutarci a gestire sequenze di passaggi interdipendenti e asincroni, permette di ridurre in modo significativo il carico di attesa per diverse attività asincrone indipendenti:

```
let z1 = Promise.resolve(3);
let z2 = 6;
let z3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, "Miao");
});

Promise.all([z1, z2, z3]).then(values => {
  console.log(values);
});
```

Come puoi vedere, non dobbiamo preoccuparci dell'ordine in cui vengono eseguite le attività e se alcune di esse sono terminate e altre no. Affermiamo di voler attendere un certo numero di promesse utilizzando il metodo *Promise.all*. Questo metodo accetta una serie di promesse e crea una nuova promessa che si risolve con successo quando tutte quelle passate si risolvono. Il callback di successo riceve un array di valori di successo, uno per ciascuna delle promesse passate.

Promise.all attende tutte le promesse in un elenco. Ma a volte abbiamo numerose promesse, ma ci preoccupiamo solo della prima che riesce (o fallisce). Scopriamo il metodo *Promise.race* il quale ci viene incontro in questi casi e tutti gli altri disponibili.

I metodi delle promesse

Le nostre promesse hanno a disposizione sei metodi, vediamoli tutti con le loro peculiarità:

- *Promise.all*, lo abbiamo visto poco fa.
- *Promise.allSettled*, a differenza del primo, anche se una promessa viene respinta attende tutte le altre.
- *Promise.race*, accetta una serie di promesse e restituisce una promessa completamente nuova che si risolve o viene rifiutata non appena la prima delle promesse viene risolta o rifiutata. In pratica il primo risultato ottenuto vince la gara (race). Vediamolo in questo codice:

```
const prom1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 50, 'primo');
});
const prom2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 5, 'secondo');
```

```
});  
Promise.race([prom1, prom2]).then((value) => {  
  console.log(value);  
});
```

- *Promise.any*, con la quale, se la condizione si attua, essa ritorna una singola promessa, altrimenti ritorna un errore, quindi a differenza del precedente vince il primo che risolve la promessa, non il più veloce:

```
const promessaA = Promise.reject(0);  
const promessaB = new Promise((resolve) => setTimeout(resolve, 120,  
'vinco sempre io'));  
const promessaC = new Promise((resolve) => setTimeout(resolve, 600,  
'non lo vedremo mai'));  
const promessaD = new Promise((resolve) => setTimeout(resolve, 300,  
'e anche questo non lo vedremo'));  
const promesse = [promessaA, promessaB, promessaC, promessaD];  
Promise.any(promesse).then((value) => console.log(value));
```

- *Promise.resolve/reject*, con cui si ottiene una promessa risolta o respinta associata ad un valore. Non vengono più utilizzati perché sostituiti da *async* e *await* che vedremo poco più avanti.

Finora abbiamo visto come funzionano le promesse e come possiamo usarle per semplificare notevolmente la gestione di una serie di passaggi asincroni, in serie o in parallelo. Nella sezione successiva, i due grandi concetti che abbiamo introdotto in questo capitolo, generatori e promesse, si uniscono per fornire la semplicità di codice sincrono con la natura non bloccante del codice asincrono.

Generatori e promesse uniti per la gloria!

Arrivati a questo punto siamo certamente in grado di passare al livello successivo e, quindi combineremo i generatori, con la loro capacità di sospendere e riprendere l'esecuzione, con le promesse.

Come sappiamo, l'uso di *yield* in un generatore sospende la sua esecuzione senza bloccarlo, infatti il metodo *next* sull'iteratore del generatore lo riattiverà. Le promesse, invece, ci permettono di specificare una richiamata che verrà attivata nel caso fossimo in grado di ottenere la promessa del valore richiesto e un callback che verrà attivato nel caso si sia verificato un errore.

L'idea, quindi, è di combinare generatori e promesse senza scrivere codice asincrono, al contrario, scrivendo in maniera sincrona utilizzando *spawn* nel

modo seguente:

```
function spawn(genFunzione) {
  function prova(verb, arg) {
    let result;
    try {result = generator[verb](arg);
    } catch(err) {return Promise.reject(err);
    }
    if (result.done) {return result.value;
    } else {return Promise.resolve(result.value).then(risult1,
    risult2);
    }
  }
  let generator = genFunzione();
  let risult1 = prova.bind(continuer, "avanti");
  let risult2 = prova.bind(continuer, "ook");
  return onFulfilled();
}
```

Questo è sicuramente un discreto risultato, ma sarà molto meglio guardare a *async* e *await*, sicuramente più funzionali.

Async e await

Da pochi anni possiamo creare ancora un altro tipo di funzione: una *async function*. Potremo inviare promesse mentre la *async function* le collega automaticamente per riprenderle al termine (non c'è bisogno nemmeno di utilizzare gli iteratori!). Iniziamo vedendo come funziona a livello di sintassi:

```
async function prova() {
  return Promise.resolve(33);
}
prova().then(console.log);
```

Tramite *async* attiviamo una funzione che restituirà sempre una promessa. Procediamo anche con *await*, il quale potrà funzionare solamente assieme ad *async*:

```
async function prova() {
  let promessa = new Promise((ok, no) =>
  {
    setTimeout(() => ok("ecco il risultato"), 2000)
  });
  let attesa = await promessa;
  console.log(attesa);
}
```

```
prova();
```

Ed ecco i due metodi in funzione, qui attenderemo *await* e la sua promessa per due secondi per poi ottenere il risultato. Sicuramente una migliore opzione rispetto a *promise.then*. Ma proviamo qualcosa di più complesso:

```
async function prova() {  
  let risult1 = await request("http://sito.internet");  
  let dati = JSON.parse(risult1);  
  
  let risult2 = await request("http://sit.int2" + dati.id);  
  let resp = JSON.parse(risult2);  
  console.log("I nostri dati che cercavamo: " + resp.value);  
}  
prova();
```

Come abbiamo già visto, un'*async function* può essere chiamata direttamente, come *prova()*, e all'interno, invece di usare *yield*, utilizzeremo *await* il quale dice alla *async function* di attendere il completamento della promessa prima di procedere. Nello specifico attendiamo di recuperare dei dati da due siti prima di procedere oltre.

Non si direbbe, ma questi esempi osservati finora sono dei ristretti, dei concentrati di molte cose che abbiamo visto finora: infatti, quando inviamo una funzione come argomento ad un'altra asincrona (tramite *async*) utilizzeremo i vantaggi del servirsene come oggetti. Con i generatori usiamo la loro capacità di sospendere e riprendere l'esecuzione, mentre le promesse ci servono per gestire il codice asincrono, e assieme ai callback sapremo se sono andate a buon fine o meno. I callback verranno sfruttati dalle funzioni freccia a causa della loro facilità d'uso, e infine l'iteratore, attraverso il quale controlliamo il generatore, viene creato nella funzione asincrona, e vi accediamo, tramite chiusure, nei callback della promessa.

Insomma, un ottimo modo di ripassare, ma anche di utilizzare diverse tecniche per far funzionare meglio il nostro codice e allo stesso tempo per renderlo breve e coinciso.

Quiz & esercizi

1) Correggere il listato utilizzando *yield* e ordinando i valori:

```
function* elenco() {  
  let a = 9;  
  let b = 5;  
  let c = 10;
```

```
}  
let gen = elenco();  
for(let valore of gen) {  
    console.log(valore);  
}
```

2) È vero che evocare un generatore equivale ad eseguire la funzione del generatore?

3) Fare in modo che il risultato sia 14 SENZA MODIFICARE i numeri nel vettore *risultato*:

```
function somma(primo, secondo, terzo, quarto, quinto) {  
    console.log(primo + secondo);  
}  
let risultato = [3, 2, 6, 4, 1];  
somma(...risultato);
```

4) Se ho dei dubbi sul risultato delle promesse userò *Promise.allSettled* o *Promise.all*?

5) Che tipo di funzione dovrò usare per ottenere una promessa?

6) Quali sono i vantaggi di un codice asincrono?

7) Crea una funzione *ritardaMessaggio* che accetti un messaggio e un tempo in millisecondi come argomenti. La funzione dovrebbe ritardare la visualizzazione del messaggio per il tempo specificato utilizzando *setTimeout*. Implementa una callback per gestire il completamento del ritardo.

8) Crea una funzione *ottieniDatiUtente* che utilizzi la Fetch API per effettuare una richiesta a un endpoint API (ad esempio, un endpoint che restituisce dati utente) e restituisca una promessa. Gestisci la promessa in modo che restituisca i dati dell'utente quando la richiesta è completata con successo.

9) Crea una funzione *ottieniDatiConcorrenti* che accetti un array di URL e utilizzi *Promise.all* per effettuare richieste parallele a tutti gli URL. La funzione dovrebbe restituire un array con i dati delle risposte quando tutte le richieste sono state completate con successo.

10) Crea una funzione asincrona *ottieniDatiUtente* che utilizzi l'operatore *await* con la Fetch API per effettuare una richiesta a un endpoint API (ad esempio, un endpoint che restituisce dati utente) e restituisca i dati dell'utente. Utilizza *try/catch* per gestire eventuali errori.

Riassunto

Il breve ripasso che abbiamo affrontato in questo capitolo servirà a tutti quelli che sono alle prime armi, in questo caso suggerisco di approfondire ulteriormente questi temi in quanto proseguendo ci serviranno delle basi solide per proseguire nello studio. Difatti il discorso verrà ripreso in vari capitoli e soprattutto per quanto riguarda le stringhe quando andremo a trattare il temibile argomento delle espressioni regolari. Adesso, prima di tuffarci a pesce verso uno degli argomenti cardine di Javascript, cioè le funzioni, avremo un'infarinatura sugli oggetti. Altro fondamento di questo linguaggio di programmazione.

9 - *Oggetti software*

I prototipi di oggetti
La creazione di oggetti
Le classi e l'ereditarietà
Utilizzo di get & set
Utilizzo del proxy

Oramai a questo punto sappiamo tutti che, come Java e C++, Javascript è un linguaggio orientato agli oggetti. Vedremo in realtà che esso si *basa* sugli oggetti (software). Ma andiamo con ordine: abbiamo imparato che le funzioni sono oggetti di prima classe, che le chiusure le rendono incredibilmente versatili e utili e che puoi combinare funzioni del generatore con le promesse per affrontare il problema del codice asincrono. Adesso siamo pronti ad affrontare un altro aspetto importante di JavaScript: i prototipi di oggetti.

Attraverso i prototipi possiamo delegare la ricerca di una particolare proprietà. Sono dei veri e propri oggetti, con tutti i vantaggi del caso, e sono anche un mezzo utile per definire proprietà e funzionalità che saranno automaticamente accessibili ad altri oggetti. I prototipi hanno uno scopo simile a quello delle classi nei linguaggi orientati agli oggetti classici.

Vedremo anche come, con i metodi *get* e *set*, riusciremo a controllare l'accesso a proprietà di oggetti privati, resi tali attraverso il cosiddetto incapsulamento.

I prototipi

Il prototipo possiamo disegnarlo come una sorta di archivio dove una quantità di proprietà e metodi possono essere condivise a chi lo richiede. Non ci suona nuovo, infatti stiamo parlando delle caratteristiche degli oggetti. Sappiamo già che in JavaScript, gli oggetti sono raccolte di proprietà

denominate con valori, e abbiamo imparato in diverse occasioni come sia semplice creare nuovi oggetti con le loro specifiche proprietà:

```
let oggetto = {  
  alfa: 5000,  
  beta: "Giovanni",  
  gamma: function(qualcosa){x,y},  
  delta: {}  
}
```

Come possiamo vedere, le proprietà degli oggetti possono essere valori semplici (come numeri o stringhe), funzioni e altri oggetti. Inoltre, JavaScript è un linguaggio altamente dinamico e le proprietà assegnate a un oggetto possono essere facilmente modificate o aggiunte da zero:

```
oggetto.alfa = 5000;  
oggetto.beta = "Mario";  
delete oggetto.gamma;  
oggetto.omega = "Luisa";
```

Quando sviluppiamo software, ci sforziamo di non reiterare parti di codice. Una forma di riutilizzo che aiuta anche a organizzare i nostri programmi è l'*ereditarietà*, che estende le funzionalità di un oggetto in un altro. In JavaScript questa funzionalità è implementata con la prototipazione.

Ma come funziona esattamente? abbiamo già detto che ogni oggetto può avere un riferimento al suo prototipo, e gli si può delegare la ricerca di una particolare proprietà se esso non la possiede, utilizzando il metodo *Object.setPrototypeOf* potremo estendere queste proprietà ad altri oggetti:

```
const mario ={basso:true};  
const nino ={calvo:true};  
Object.setPrototypeOf(mario, nino);
```

In questo caso *nino* è diventato il prototipo di *mario*, e quindi quest'ultimo potrà essere *calvo*.

Con una funzione di costruzione, usando la parola chiave *new* si potranno creare nuovi oggetti dello stesso prototipo:

```
function animale(colore, tipologia, nome, peso) {  
  this.coloreAnimale = colore;  
  this.tipoAnimale = tipologia;  
  this.nomeAnimale = nome;  
}
```

```
    this.pesoAnimale = peso;  
}
```

```
let mioCane = new animale("Marrone", "Setter", "Billy", 10);  
let mioGatto = new animale("Grigio", "Certosino", "Kitty", 5);
```

È importante sottolineare che ogni oggetto può avere un prototipo e il prototipo di un oggetto può anche avere un prototipo e così via, formandone così una catena. La delega di ricerca per una particolare proprietà si verifica lungo l'intera catena e si interrompe solo quando non ci sono più prototipi da esplorare.

Vediamo ora le implicazioni che ci possono essere assieme alla funzione di costruzione.

Prototipi, oggetti e costruttori

Sappiamo già che attraverso il costruttore avremo modo di creare un nuovo oggetto, con un'istruzione come questa:

```
const gatto = {};
```

Questo crea un oggetto che potremo implementare aggiungendo delle proprietà tramite le istruzioni di assegnazione:

```
const gatto = {};  
gatto.nome = 'Kitty';
```

Tutto chiaro e semplice, finchè non dovremo creare più istanze dello stesso tipo di oggetto e assegnargli le proprietà individualmente. Tutto ciò diventerebbe tremendamente noioso ma probabilmente anche soggetto a errori.

A questo punto ci piacerebbe poter consolidare l'insieme di proprietà e metodi per una classe di oggetti in un unico posto.

Per fortuna il nostro linguaggio di programmazione preferito ci fornisce un tale meccanismo, sebbene in una forma diversa rispetto alla maggior parte degli altri, quali Java e C++ ad esempio.

JavaScript utilizza l'operatore *new* per istanziare un nuovo oggetto tramite il costruttore, ma di contro non potremo definirne una classe. Invece, *new* applicato assieme a una funzione di costruzione attiva la creazione di un oggetto appena allocato, come abbiamo già visto nel capitolo 6, e adesso scopriremo e impareremo che ogni funzione ha un oggetto di tipo *prototipo*

assegnato automaticamente ad ogni nuova creazione, osserviamolo subito sul campo:

```
function Prova() {}  
Prova.prototype.qualcosa = function() {  
    return true;  
};  
  
const p2 = Prova();  
p2; // undefined  
  
const p3 = new Prova();  
console.log(p3 && p3.qualcosa && p3.qualcosa()); // true
```

Vediamo che abbiamo lanciato una funzione chiamata *Prova* poi invocata in due modi: come una funzione "normale", *const p2 = Prova()*, e dal costruttore *new*: *const p3 = new Prova()*;

Quando la funzione viene creata, ottiene immediatamente un nuovo oggetto assegnato al suo prototipo, in questo caso, abbiamo aggiunto un metodo *qualcosa*.

Appena eseguito il codice chiamiamo la funzione normalmente e memorizziamo il suo risultato nella variabile *p2*. Guardando il corpo della funzione, vediamo che *p2* verrà segnalato come indefinito. Quindi la prima invocazione di fatto non sembrerebbe molto utile.

Tuttavia più avanti invochiamo la funzione tramite il costruttore *new*, e questa volta è stato creato un oggetto appena allocato e impostato come contesto della funzione. Il risultato restituito dal nuovo operatore è un riferimento a questo nuovo oggetto. Testiamo quindi che *p3* abbia effettivamente un riferimento all'oggetto appena creato e che quell'oggetto abbia un metodo *qualcosa* che possiamo chiamare.

Ne consegue che, quando usiamo una funzione come costruttore (come abbiamo fatto con *new Prova()*), il prototipo dell'oggetto di nuova costruzione è impostato sull'oggetto a cui fa riferimento il prototipo della funzione di costruzione.

In poche parole, *Prova.prototype* e il suo metodo *qualcosa* saranno le stesse per l'oggetto *p3*. Pertanto, quando proveremo ad accedere al file *qualcosa* di *p3*, la ricerca di quella proprietà è delegata all'oggetto prototipo *Prova*. Concludendo, per tutti gli oggetti creati con l'estensione *Prova* il costruttore avrà accesso al file *qualcosa*. Questo è un ottimo esempio di riutilizzo del codice e ci sarà sicuramente molto utile.

Il metodo *qualcosa* è una proprietà del prototipo di *Prova* e non una proprietà delle istanze *p2* o *p3*. Dobbiamo imparare bene questa differenza tra proprietà di istanza e proprietà del prototipo.

Differenze di concetto

In pratica, quando si crea un nuovo oggetto, come sappiamo, questo avrà i suoi metodi e attributi. Quando vorremmo condividerli con altri nuovi oggetti dovremo per forza di cose fare riferimento ad una classe, la quale non è altro che un modello, un contenitore, che verrà utilizzato proprio per questo scopo. Le nuove creazioni saranno istanze della classe.

Quindi, una proprietà tipica dichiarata in una classe è una proprietà di istanza, il che significa che hai una copia del valore di quella proprietà per ogni istanza della classe:

```
class PadroneCane {  
  constructor() {  
    this.Nome = "Mario Rossi";  
  }  
}  
  
const pad = new PadroneCane();  
console.log(pad.Nome);
```

Con questo codice qui sopra creiamo semplicemente una classe (*PadroneCane*) e definiamo un campo istanza chiamato *Nome*, e questo campo è accessibile solo tramite istanza, la variabile *pad*. Possiamo riscrivere questo codice rimuovendo il costruttore:

```
class PadroneCane {  
  Nome = "Mario Rossi";  
}  
  
const pad = new PadroneCane();  
console.log(pad.Nome);
```

La differenza fondamentale tra i due esempi è la modalità di dichiarazione delle proprietà all'interno di una classe: nel primo, la proprietà *Nome* è dichiarata come una proprietà di istanza e viene inizializzata nel costruttore. Questo significa che ogni istanza della classe *PadroneCane* avrà una copia separata della proprietà *Nome*. Questo approccio è tradizionalmente noto come dichiarazione delle proprietà di istanza.

```
class PadroneCane {  
  constructor() {  
    this.Nome = "Mario Rossi";  
  }  
}
```

Vantaggi: la proprietà di istanza è specifica per ogni oggetto istanza, il che significa che ogni oggetto può avere un valore diverso per *Nome*.

È possibile personalizzare il valore della proprietà quando si crea un'istanza passando un valore diverso al costruttore.

Svantaggi: è necessario un costruttore separato per inizializzare la proprietà. Se non si specifica un valore nel costruttore, la proprietà avrà un valore predefinito.

Nel secondo esempio, la proprietà *Nome* è dichiarata utilizzando la sintassi delle proprietà di classe, che è una funzionalità introdotta nelle versioni più recenti di JavaScript. Questo approccio rende la proprietà *Nome* una proprietà statica della classe *PadroneCane*, condivisa da tutte le istanze.

```
class PadroneCane {  
  Nome = "Mario Rossi";  
}
```

Vantaggi: non è necessario un costruttore separato per inizializzare la proprietà.

La proprietà è condivisa tra tutte le istanze della classe, il che significa che tutte le istanze avranno lo stesso valore per *Nome*.

Svantaggi: la proprietà è condivisa tra tutte le istanze, quindi non è possibile personalizzare il valore per ciascuna istanza.

Non è possibile passare un valore diverso al costruttore per inizializzare la proprietà in modo dinamico. Scegliere tra i due approcci dipende dalle esigenze specifiche del tuo programma:

- Se hai bisogno che ciascuna istanza abbia un valore separato per la proprietà, utilizza la dichiarazione di proprietà di istanza con un costruttore.
- Se la proprietà deve essere condivisa tra tutte le istanze e non deve essere personalizzata, utilizza la sintassi delle proprietà di classe.

Entrambi gli approcci sono validi e possono essere utili in diverse situazioni. Quindi, in poche parole, per creare una proprietà di classe, avremo bisogno di *static*:

```
class PadroneCane {  
    static Nome = "Mario Rossi";  
}  
const pad = new PadroneCane();  
console.log(pad.Nome);  
console.log(PadroneCane.Nome);
```

Come si può vedere nel codice in alto, adesso il nome di campo è accessibile direttamente tramite classe e non è possibile accedervi utilizzando l'istanza *pad*.

Ne consegue che ogni istanza ottiene la propria versione delle proprietà che sono state create all'interno del costruttore, mentre tutte hanno accesso alle proprietà dello stesso prototipo. Questo va bene per le proprietà del valore che sono specifiche per ciascuna istanza dell'oggetto. Ma in alcuni casi potrebbe essere problematico per i metodi.

Questo non è un problema se creiamo un paio di oggetti, ma è tutt'altra cosa se intendiamo crearne un gran numero. Poiché ogni copia del metodo si comporta allo stesso modo, la creazione di più copie spesso non ha senso, perché consuma solo più memoria. Certo, in generale, il motore JavaScript potrebbe eseguire alcune ottimizzazioni, ma non ci possiamo basare su questo. Ha senso posizionare i metodi dell'oggetto solo sul prototipo della funzione? Sì, perché in questo modo abbiamo un unico metodo condiviso da tutte le istanze dell'oggetto.

Prototipi e proprietà hanno le stesse peculiarità

Nel corso dei capitoli precedenti, abbiamo appreso che le proprietà possono essere facilmente aggiunte, rimosse e modificate a piacimento. Questa regola si applica anche ai prototipi, che possono essere sia di funzioni che di oggetti. Per comprendere meglio questo concetto, esaminiamo un esempio esplicativo:

```
function Prova() {  
    this.qualcosa = true;  
}  
  
const p2 = new Prova();  
  
Prova.prototype.qualcosAltro = function() {  
    return this.qualcosa;  
};
```

```
assert(p2.qualcosAltro()); // Restituisce true
```

In questo esempio, definiamo una funzione costruttrice *Prova* e creiamo un'istanza chiamata *p2*. Successivamente, aggiungiamo un metodo *qualcosAltro* al prototipo della funzione *Prova*. Anche se abbiamo aggiunto questo metodo dopo aver creato l'istanza *p2*, l'istanza può ancora accedere al metodo *qualcosAltro* attraverso la catena del prototipo.

```
Prova.prototype = {  
  niente: function() {  
    return true;  
  }  
};
```

```
assert(p2.qualcosAltro()); // Continua a restituire true, nonostante  
la sovrascrittura del prototipo con "niente"
```

```
const p3 = new Prova();  
assert(p3.niente()); // Restituisce true  
assert(!p3.qualcosAltro()); // Errore! Il metodo non è definito su  
questa istanza
```

In questo secondo frammento di codice, definiamo nuovamente la funzione costruttrice *Prova*. Successivamente, sovrascriviamo il prototipo della funzione *Prova* assegnandolo a un nuovo oggetto che contiene un metodo *niente*. Nonostante questa sovrascrittura del prototipo, l'istanza *p2*, creata in precedenza, può ancora accedere al metodo *qualcosAltro*. Tuttavia, l'istanza *p3*, creata dopo la sovrascrittura del prototipo, non ha accesso al metodo *qualcosAltro* ma può utilizzare il nuovo metodo *niente*.

La chiave per comprendere questo comportamento è che il riferimento tra un oggetto e il prototipo della funzione viene stabilito al momento dell'istanziamento dell'oggetto. Gli oggetti di nuova creazione avranno un riferimento al nuovo prototipo, mentre gli oggetti esistenti manterranno il riferimento al prototipo originale. Questo può sembrare complesso, ma seguendo la logica si può comprendere chiaramente come funziona questo meccanismo.

Un'altra funzione interessante è la possibilità di definire un costruttore e di creare un'istanza di oggetto con esso.

Sebbene sia fantastico sapere come JavaScript utilizza il prototipo per trovare i riferimenti di proprietà corretti, è anche utile sapere quale funzione ha costruito un'istanza di oggetto. Come si è visto in precedenza, il costruttore di un oggetto è disponibile tramite la proprietà *constructor* del

prototipo della funzione costruttore. Ad esempio, qui sotto vediamo lo stato dell'applicazione quando istanziamo un oggetto:

```
function Gatto() {}  
const gatto = new Gatto();  
  
assert(typeof gatto === "object"); // true  
assert(gatto.constructor === Gatto); // true  
assert(gatto instanceof Gatto); // true
```

L'oggetto prototipo di ogni funzione ha una proprietà costruttore che fa riferimento alla funzione. Attraverso le asserzioni possiamo sapere tutto quello che ci interessa.

```
function Gatto() {  
  
  const gatto = new Gatto();  
  const gatto2 = new gatto.constructor();  
  
  assert(gatto2 instanceof Gatto); // true
```

Costruisce un secondo *Gatto* dal primo, ma non sarà lo stesso oggetto:

```
assert(gatto !== gatto2); // errore!  
}
```

Le asserzioni mostrano che sì, è stato costruito un secondo *Gatto*, ma che la variabile non indica la stessa istanza.

Ciò che è particolarmente interessante è che possiamo farlo senza nemmeno avere accesso alla funzione originale; possiamo usare il riferimento completamente dietro le quinte, anche se il costruttore originale non è più nell'ambito.

Ricordiamoci che, nonostante la proprietà del costruttore di un oggetto possa essere modificata, in realtà non ci sono poi tutti questi motivi per farlo. La ragione di essere della proprietà è indicare da dove è stato costruito l'oggetto. Se la proprietà del costruttore viene sovrascritta, il valore originale andrà perso.

L'ereditarietà delle proprietà

Attraverso la cosiddetta ereditarietà degli oggetti potremo crearne di nuovi i quali avranno accesso alle proprietà di quelli esistenti. Questa possibilità ci aiuterà a evitare la ridondanza del codice e dei dati mentre progettiamo e

scriviamo il nostro listato. Dobbiamo anche sapere che in JavaScript l'ereditarietà funziona in modo leggermente diverso rispetto ad altri popolari linguaggi orientati agli oggetti:

```
function Aereo() {}  
Aereo.prototype.volo = function() {};
```

Definisce un *Aereo* che vola tramite un costruttore e il suo prototipo.

```
function Uomo() {}  
  
Uomo.prototype = {volo: Uomo.prototype.volo};
```

Un *Uomo* tenta di copiare il metodo *volo* dell'*Aereo*.

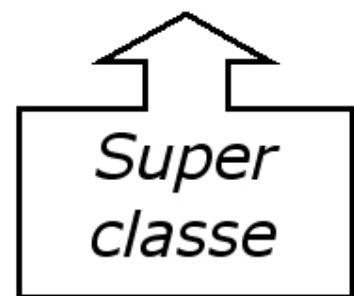
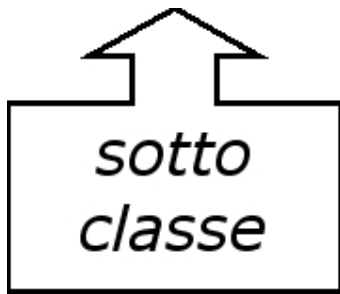
```
const mario = new Uomo();  
  
assert(mario instanceof Uomo); // true  
assert(mario instanceof Aereo); // false!
```

Sappiamo bene che il prototipo di una funzione è un oggetto, di conseguenza esisteranno diversi modi per copiare ad esempio le proprietà o i metodi sfruttando l'ereditarietà. In questo codice definiamo un *Aereo* e poi un *Uomo*. Decidiamo che *mario* ne erediti gli attributi. Nonostante il nostro tentativo non siamo riusciti a fare in modo che *mario* sia un'istanza dell'*Aereo*.

A parte il fatto che questo approccio non funziona correttamente, avremmo anche bisogno di copiare ogni proprietà dell'*Aereo* al prototipo *Uomo* individualmente. E questo non è un buon modo per sfruttare l'ereditarietà.

Quello che vogliamo veramente ottenere è una catena di prototipi in modo che un *Uomo* possa essere un *Aereo*, questo possa essere un'*Astronave* e così via. La tecnica migliore per creare una catena di prototipi di questo tipo consiste nell'usare un'istanza di un oggetto come prototipo dell'altro oggetto:

```
Uomo.prototype = new Aereo();
```



Ciò preserva la catena del prototipo, perché il prototipo dell'istanza della *sottoclasse* sarà un'istanza della *Superclasse*, che ha un prototipo con tutte le proprietà della *Superclasse*, e che a sua volta avrà un prototipo che punta a un'istanza della sua superclasse, e così via.

```
function Aereo() {}  
Aereo.prototype.volo = function() {};  
function Uomo() {}  
Uomo.prototype = new Aereo();
```

Rende un *Uomo* un *Aereo* trasformando il prototipo *Uomo* in un'istanza dell'*Aereo*, ereditando così anche il metodo *volo*.

```
const mario = new Uomo();  
  
assert(mario instanceof Uomo); // true  
assert(mario instanceof Aereo); // true  
assert(mario instanceof Object); // true  
assert(typeof mario.volo === "function"); // true
```

L'unica modifica al codice consiste nell'usare un'istanza dell'*Aereo* come prototipo dell'*Uomo*. L'esecuzione delle asserzioni mostra che siamo riusciti. Quindi, seguendo il ragionamento, all'occorrenza potremo anche eseguire questo codice:

```
Uomo.prototype = Aereo.prototype
```

In realtà, non è per niente una buona idea, infatti qualsiasi modifica al prototipo *Uomo* cambierà di conseguenza anche il prototipo *Aereo* (dato che saranno lo stesso oggetto), e questo è destinato ad avere effetti collaterali indesiderati piuttosto noiosi, come ad esempio il fatto che tutti i prototipi di

funzioni ereditate continueranno a essere aggiornati in tempo reale. Oggetti che ereditano dal prototipo hanno sempre accesso alle loro proprietà.

Questo è un bel problema! E lo risolveremo più avanti, ma prima di poterlo fare, dobbiamo vedere come è possibile configurare le proprietà degli oggetti software.

Le proprietà dell'oggetto software

Soffermiamoci sulle numerose possibilità che abbiamo di modificare le proprietà dell'oggetto e sulle conseguenze di questi cambiamenti. Possiamo iniziare con un gatto a cui assegniamo una proprietà “nome” che è “Briciola”:

```
gatto.nome = "Briciola";
```

Scopriremo che la proprietà *nome* sarà *configurabile*, *enumerabile* e *scrivibile*, il suo valore sarà impostato su *Briciola* e le funzioni *getter* e *setter* saranno non definite. Forse qualcuno potrebbe citare la famosa battuta di Verdone: “..In che senso??”. Facciamo un passo indietro, tutte le proprietà di un oggetto possono essere:

- **Configurabili** (configurable): se impostate su *true*, il descrittore della proprietà potrà essere modificato e il file proprietà potrà essere cancellato. Se impostate su *false*, non si potrà fare nessuna di queste cose.
- **Enumerabili** (enumerable): se *false*, alcune operazioni sulle proprietà non potranno essere eseguite, come un ciclo *for-in*.
- **Scrivibili** (writable): se *true*, il valore della proprietà può essere modificato utilizzando un'assegnazione.

Inoltre potremo impostargli un **valore** (value), il quale sarà *undefined* se non assegnato.

Infine sulle proprietà potremo definire due funzioni:

Get: definisce la funzione *getter*, che verrà chiamata quando accediamo alla proprietà. Non può essere definito insieme a valore e scrivibile.

Set: definisce la funzione *setter*, che verrà chiamata ogni volta che viene eseguita un'assegnazione fatta alla proprietà. Non può essere definita insieme a *value* e *writable*.

Quando vogliamo mettere a punto la nostra configurazione delle proprietà, utilizzeremo il metodo incorporato *Object.defineProperty*, che accetta un oggetto su cui verranno definite come proprietà: il nome, un oggetto utile come cibo e il colore:

```
let gatto = {}; // Definiamo prima l'oggetto gatto

gatto.nome = "Briciola";
gatto.cibo = "Gamberoni";
gatto.colore = "Nero";
```

Come appena visto, prima creiamo un oggetto *gatto* (inizialmente vuoto) e dopo gli aggiungiamo tre proprietà.

Adesso, attraverso il metodo incorporato *Object.defineProperty* ottimizzeremo i dettagli di configurazione della proprietà:

```
Object.defineProperty(gatto, "simpatia", {
  configurable: false,
  enumerable: false,
  value: true,
  writable: true
});
assert("simpatia" in gatto); // true

for (let prop in gatto) {
  if (gatto.hasOwnProperty(prop)) {
    console.log("Proprietà enumerata:" + prop);
  }
}
```

Concludendo, nell'esempio appena visto iniziamo creando un oggetto vuoto *gatto* a cui aggiungiamo tre proprietà: *nome*, *cibo* e *colore*.

Successivamente, utilizziamo il metodo *Object.defineProperty* per definire la proprietà *simpatia*. Abbiamo impostato *configurable* su *false*, quindi questa proprietà non può essere rimossa o modificata in seguito. Abbiamo anche impostato *enumerable* su *false*, il che significa che la proprietà *simpatia* non verrà elencata quando utilizziamo un ciclo *for-in*. Tuttavia, possiamo ancora accedere a questa proprietà utilizzando l'operatore *in*. Infine, utilizziamo un ciclo *for-in* per iterare attraverso tutte le proprietà enumerabili dell'oggetto *gatto*. Poiché *simpatia* non è *enumerabile*, non verrà inclusa nell'elenco delle proprietà durante l'iterazione, quindi, impostando *enumerable* su *false*, possiamo essere sicuri che la proprietà non verrà visualizzata quando si utilizza il ciclo *for-if*. Per capire perché vorremmo fare tutto ciò, torniamo alla questione originale relativa all'*Uomo* e all'*Aereo* vista poco fa.

Possiamo risolvere questo problema utilizzando le nozioni appena studiate. Definiamo un costruttore sul nuovo *Aereo.prototype* utilizzando il metodo *Object.defineProperty*:

```
function Uomo() {}
Uomo.prototype.volo = function() {};
```

```
function Aereo() {}
Aereo.prototype = new Uomo();
```

Adesso definiremo una nuova proprietà costruttore non enumerabile che punta all'*Aereo*:

```
// Modifichiamo il costruttore di Aereo in modo che punti
correttamente a Aereo.
Object.defineProperty(Aereo.prototype, "constructor", {
  enumerable: false,
  value: Aereo,
  writable: true
});
```

```
// Verifichiamo che il costruttore sia stato correttamente
impostato.
let mario = new Aereo();
assert(mario.constructor === Aereo); // true
```

```
// Verifichiamo le proprietà di Aereo.prototype.
for (let prop in Aereo.prototype) {
  assert(prop === "volo"); // true
}
```

Non abbiamo aggiunto alcuna proprietà enumerabile a *Aereo.prototype*.

Ora, se eseguiamo il codice, vedremo che tutto gira bene. Abbiamo ristabilito la connessione tra l'istanza *mario* e la funzione *Aereo*, così possiamo sapere che sono state costruite dalla funzione *Aereo*.

Le applicazioni dell'operatore instanceof

Nella maggior parte dei linguaggi di programmazione, l'approccio diretto per verificare se un oggetto fa parte di una gerarchia di classi consiste nell'usare l'operatore *instanceof*. Ad esempio, in Java, esso funziona controllando se l'oggetto a sinistra è la stessa classe o una sottoclasse del tipo di classe sulla destra, mentre in JavaScript, l'operatore *instanceof* lavora sulla catena di prototipi dell'oggetto:

```
function Uomo() {}
```

```
function Aereo() {}

Aereo.prototype = new Uomo();
const mario = new Aereo();
```

Un'istanza *mario* è sia un *Aereo* che un *Uomo*.

```
assert(mario instanceof Aereo); // true
assert(mario instanceof Uomo); // true
```

L'operatore *instanceof* serve a controllare se il prototipo corrente della funzione *Aereo* è nella catena di prototipi dell'istanza *mario*. Sappiamo che *mario* è composto da un oggetto *new Aereo()*, attraverso il quale abbiamo ottenuto l'ereditarietà, e il Prototipo di *Uomo* attraverso l'espressione: *mario instanceof Aereo*, il motore JavaScript prende il prototipo della funzione *Aereo*, l'oggetto *new Uomo()* e controlla se si trova nella catena del prototipo dell'istanza *mario*. Poiché l'oggetto *new Uomo()* è un prototipo diretto dell'istanza *mario*, il risultato è *true*. Nella seconda asserzione, dove controlliamo *mario instanceof Uomo*, il motore JavaScript prende il prototipo della funzione *Uomo*, e controlla se può essere trovato nella catena di prototipi dell'istanza *mario*. Di nuovo, può, perché è il prototipo del nostro oggetto *new Uomo()*, che, come abbiamo già visto, è il prototipo dell'istanza *mario*.

Forse seguire con attenzione tutte le possibilità è stato un po' noioso, vediamo un esempio ancora più basico in cui poter utilizzare *instanceof*:

```
If (oggetto1 instanceof Classe1) {
    // codice che fa qualcosa
} else if (oggetto2 instanceof Classe2) {
    // codice che fa qualcos'altro
}
```

Forse lo spazio dedicato a questo argomento non è stato sufficiente a comprenderlo appieno. Sebbene il suo utilizzo più comune sia fornire un modo chiaro per determinare se un'istanza è stata creata da un particolare costruttore di funzioni, non funziona esattamente in questo modo. Invece, controlla se il prototipo della funzione sul lato destro è nella catena di prototipi dell'oggetto sulla sinistra. Pertanto, c'è un avvertimento a cui dovremmo prestare attenzione.

Sappiamo bene che JavaScript è un linguaggio dinamico in cui possiamo modificare molte cose durante l'esecuzione del programma. Ad esempio,

nulla ci impedisce di modificare il prototipo di un costruttore:

```
function Aereo() {}  
const mario = new Aereo();  
assert(mario instanceof Aereo); // true
```

Anche se la nostra istanza *mario* è stata creata dal costruttore *Aereo*, l'operatore *instanceof* ci dirà che *mario* non è più un'istanza di *Aereo*:

```
Aereo.prototype = {};  
assert(!(mario instanceof Aereo)); // errore!
```

Cosa succede? eppure stiamo ripetendo di nuovo tutti i passaggi di base per creare un'istanza *mario*, ma se cambiamo il prototipo della funzione costruttore *Aereo* dopo la creazione dell'istanza *mario*, e proviamo a testare se è una sua istanza, otterremo un errore. Se ci pensiamo non dovrebbe essere così strano, infatti il compito di *instanceof* è quello di controllare solo se il prototipo della funzione sul lato destro è nella catena del prototipo dell'oggetto sul lato sinistro e nel caso appena visto *Aereo* è un nuovo oggetto vuoto il quale non potrà essere agganciato alla catena dei prototipi, perché, riassumendo all'osso, siamo usciti fuori dalla sequenza che ci legava alla catena dei prototipi (la quale è "attaccata" al costruttore) osservata nell'esempio all'inizio del paragrafo.

Ora che abbiamo capito come funzionano i prototipi e come utilizzarli insieme alle funzioni del costruttore per implementare l'ereditarietà, passiamo ad analizzare *le classi*, con le quali saliremo ulteriormente di livello.

Le classi

Se volessimo adoperare il modo migliore, nonché più funzionale, di creare oggetti e implementarne l'ereditarietà dovremo assolutamente usare la parola chiave *class*. Vediamo come funziona nel dettaglio nel prossimo esempio dove andremo a creare una classe *moto*, specificando tre proprietà: *modello*, *cavalli* e *velocità*, utilizzeremo anche una funzione *mostra* che richiameremo come metodo sull'oggetto *moto* per visualizzarne le proprietà:

```
class Moto {  
  constructor(modello, cavalli, velocita) {  
    this.modello = modello;  
    this.cavalli = cavalli;  
    this.velocita = velocita;  
  }  
}
```



```

    }
    mostra() {
        return this.modelo + ' ' + this.cavalli + ' ' +
this.velocita;
    }
}

const motoMia = new Moto('Ducati Monster', '160 CV', 200);

console.log(motoMia.mostra());
console.log(motoMia.modelo);

```

Ecco una spiegazione dettagliata del listato e dei suoi punti salienti:

Definizione della classe Moto: iniziamo definendo una classe chiamata *Moto* utilizzando la parola chiave *class*. Questa classe avrà tre attributi: *modelo*, *cavalli*, e *velocita*.

Il costruttore: all'interno della classe *Moto*, c'è un metodo speciale chiamato costruttore *constructor*. Questo metodo viene chiamato quando creiamo una nuova istanza della classe. Prende tre argomenti: *modelo*, *cavalli*, e *velocita*, che vengono utilizzati per inizializzare gli attributi della motocicletta.

Il metodo mostra(): all'interno della classe, c'è anche un metodo chiamato *mostra()*. Questo metodo non richiede argomenti ed è utilizzato per ottenere una rappresentazione testuale delle informazioni sulla motocicletta. Restituisce una stringa che contiene il *modelo*, i *cavalli* e la *velocità* della motocicletta.

Creazione di un'istanza: dopo aver definito la classe *Moto*, creiamo un'istanza chiamata *motoMia* utilizzando il costruttore. Passiamo il modello "Ducati Monster", la potenza "160 CV" e la velocità "200" come argomenti.

Visualizzazione delle informazioni: utilizziamo il metodo *mostra()* per ottenere una rappresentazione testuale delle informazioni sulla motocicletta *motoMia* e la stampiamo sulla console. Inoltre, accediamo direttamente all'attributo *modelo* di *motoMia* e lo stampiamo anche sulla console.

Class extend e super per l'ereditarietà

Come abbiamo scritto e menzionato più volte in diversi paragrafi, l'ereditarietà è una caratteristica fondamentale della programmazione ad oggetti, quindi, si intende la possibilità di estendere una classe madre attraverso una classe figlia che eredita tutte le caratteristiche dalla classe madre.

Tutto questo serve per evitare ridondanze e scrivere del codice più strutturato e leggibile. Ed ecco quindi che *extends* ci consente proprio di sfruttare questa caratteristica, controlliamo insieme:

```
class Automobili {
  constructor(mod, vel, prezzo) {
    this.mod = mod;
    this.vel = vel;
    this.prezzo = prezzo;
  }
}

class Tipi extends Automobili {
  constructor(mod, vel, prezzo, tipo) {
    super(mod, vel, prezzo);
    this.tipo = tipo;
  }
}

let sportiva = new Tipi('Ferrari', 'veloce', 'alto', 'sportiva');
console.log(sportiva.tipo,      sportiva.mod,      sportiva.vel,
sportiva.prezzo);
```

Qui sopra vediamo che dopo aver definito una classe *Automobili* con il costruttore a cui passiamo come parametri il modello, la velocità ed il prezzo, andiamo a costruire la classe *tipi* come estensione di *Automobili*, ne consegue che da quest'ultima eredita le tre proprietà sopraindicate.

All'interno del costruttore evochiamo *super* per richiamare le proprietà del genitore ereditate. Riassumendo in breve, *super* ci permette di fare riferimento alla *superclasse*.

Riproponiamo il nostro esempio precedente con l'uomo e l'aereo adattato per notare le differenze e la maggior semplicità per arrivare allo stesso obiettivo utilizzando *class*:

```
class Uomo{ constructor(nome){
  this.nome = nome;
}
  volo(){ return true;
}
}
class Aereo extends Uomo { constructor(nome){
  super(nome);
}
}
```

```
let uomo = new Uomo("Gianni");  
let aereo = new Aereo("Mario");
```

Notiamo come attraverso *extends* estendiamo il *volo* e altre eventuali proprietà al costruttore *Uomo* in maniera molto diretta e leggibile. Lo stesso vale per i nomi da assegnare. Non ci sono dubbi, questo strumento è molto potente, e, allo stesso tempo relativamente semplice. Detto questo, rimane da approfondire il settore inerente le proprietà degli oggetti (ancora) e, più specificatamente, le modalità per modificarle e controllarle, possibili tramite i metodi “*Get*” e “*Set*”.

Get e set

Abbiamo imparato che gli oggetti sono delle raccolte di proprietà, essenzialmente di due categorie: quella di tipologia “*dati*” con cui ci siamo barcamenati in diversi capitoli che ci hanno preceduto, e le proprietà *accessorie*, cioè praticamente delle funzioni che verranno eseguite quando richiesto o quando un determinato valore sarà restituito. I metodi *get* e *set* (*getter* e *setter*) definiscono questa situazione e controllano anche l’accesso a proprietà di oggetti specifici.

In JavaScript, come già scritto, gli oggetti sono raccolte di proprietà. Tuttavia non abbiamo scritto che una delle modalità con cui riusciremo a tenere traccia dello stato del nostro programma è modificare proprio tali proprietà. Ad esempio:

```
function Gatto(oreDiSonno) {  
  this.oreDiSonno = oreDiSonno;  
}  
const gatto = new Gatto(10);
```

Qui definiamo un file *Gatto* che crea altri oggetti *gatto* con una proprietà basata sulle ore che dorme. In un secondo momento potremo assegnare un nuovo valore a quella proprietà senza alcun problema, salvo che questo valore sia, ad esempio, diverso da un numero (*NaN*, not a number), a questo punto sarebbe una buona idea poter registrare tutte le modifiche inerenti alle ore. Per questo motivo abbiamo parlato di *get* e *set* con i quali possiamo anche imitare le proprietà degli oggetti privati attraverso le chiusure:

```
function Gatto() {  
  let oreDiSonno;  
  this.getSonno = () => oreDiSonno;
```

```
    this.setSonno = value => {  
        orediSonno = value;  
    };  
}
```

Il metodo *get(getSonno)* controlla l'accesso alla nostra variabile privata *orediSonno*, mentre il metodo *set(setSonno)* ne controlla i valori che vi possiamo assegnare.

```
const gatto = new Gatto();  
gatto.setSonno(10);  
assert(gatto.getSonno() === 10, "Il gatto dorme 10 ore!");
```

Ma forse non è tutto oro ciò che brilla: la proprietà *orediSonno* è una proprietà *value*, fa riferimento a dati (il numero 10) e non a una funzione. Sfortunatamente, per sfruttare tutti i vantaggi dell'accesso controllato, tutte le nostre interazioni con la proprietà devono essere effettuate chiamando esplicitamente i metodi associati, il che è, chiaramente un problema.

Fortunatamente, JavaScript ha il supporto integrato per *getter* e *setter*: proprietà a cui si accede come normali proprietà dei dati (ad esempio, *gatto.oreSonno*), ma che sono metodi in grado di calcolare il valore di una proprietà richiesta, convalidarne il valore, o qualsiasi altra cosa abbiamo bisogno che facciano. Lo vedremo poco più avanti, per adesso proviamo un altro esempio per capire bene come funzionano queste istruzioni con un esempio su di un sito web che vende prodotti tecnologici:

```
function Prodotto(tipo, prezzo) {  
    this.tipo = tipo;  
    this.prezzo = prezzo;  
    this.sconto = 0;  
}  
  
let televisore = new Prodotto("Televisore", 500);  
let computer = new Prodotto("Computer", 1000);  
  
console.log(televisore); // Stampa l'oggetto televisore  
console.log(computer); // Stampa l'oggetto computer
```

Sicuramente nel nostro listino avremo bisogno di inserire delle scontistiche diverse per determinate categorie:

```
function distributSconto(ogg) { ogg.sconto += 18; /*... */ }  
function saldiSconto(ogg) { ogg.sconto += 10; /*... */ }
```

```
function fidelizzSconto(ogg) { ogg.sconto += 8; /*... */ }
```

Procedendo probabilmente avremo la necessità di definire una soglia massima di scontistica, ipotizziamo il 30%. Dovremo cercare nel codice ogni riferimento al nostro limite ed aggiungere questa riga:

```
if (ogg.sconto>30) ogg.sconto = 30;
```

Il gestore del nostro sito avrà bisogno di attuare altre politiche di scontistica, ad esempio, per i rivenditori lo sconto massimo può essere del 40% . E si dovrà fare di nuovo la modifica su più punti, inoltre dovremo ricordarci di modificare queste linee ogni volta che la strategia viene cambiata. Tutto questo è evidentemente un problema. Ecco perché l'incapsulamento è il principio di base dei linguaggi orientati agli oggetti. Vediamo:

```
function Prodotto(tipo, prezzo) {  
  let _tipo = tipo, _prezzo = prezzo, _sconto = 0;  
  this.getTipo = function() { return _tipo; }  
  this.setTipo = function(value) { _tipo = value; }  
  this.getPrezzo = function() { return _prezzo; }  
  this.setPrezzo = function(value) { _prezzo = value; }  
  this.getSconto = function() { return _sconto; }  
  this.setSconto = function(value) { _sconto = value; }  
}
```

In questo caso, qui possiamo semplicemente modificare i metodi *getSconto()* e *setSconto()*. Il problema è che la maggior parte dei membri si comporta come variabili comuni, solo lo sconto richiede una cura speciale qui. Ma una buona progettazione richiede l'incapsulamento di ogni membro di dati per mantenere il codice estensibile. Quindi è necessario aggiungere molto codice che praticamente non fa nulla, ottenendo così, utilizzando un termine che ha più a che fare con la moda, un design ridondante e antiestetico. In inglese viene meglio definito con: *boilerplate code*, cioè parti di codice ripetute più volte apparentemente senza cambiamenti. Si potrebbe risolvere con un'aggiustatina, ma, a volte non è possibile eseguire il *refactoring* (cioè la modifica di parti del codice senza cambiarne la struttura principale) dei campi sui metodi in un secondo momento, il programma del negozio online potrebbe aumentare di dimensioni o alcuni codici di terze parti potrebbero dipendere dalla vecchia versione, quindi il *boilerplate* è meno problematico qui. Tuttavia non va assolutamente bene. Ecco perché le proprietà sono state introdotte in molte lingue. Possiamo mantenere il codice originale, basta trasformare il membro dello sconto in una proprietà con *get* e *set*:

```

function Prodotto(tipo, prezzo) {
  this.tipo = tipo;
  this.prezzo = prezzo;
  let _sconto; // scontistica per membri privati
  Object.defineProperty(this, "sconto", {
    get: function() {
      return _sconto;
    },
    set: function(value) {
      _sconto = value;
      if (_sconto > 30) _sconto = 30;
    }
  });
}

let televisore = new Prodotto("Televisore", 500);
let computer = new Prodotto("Computer", 1000);

televisore.sconto = 50; // 50 richiamato con set
televisore.sconto += 20; // 70 richiamato con set
televisore.sconto += 20; // 80
console.log("Sconto televisore:", televisore.sconto); // 80
// richiamato con get

computer.sconto = 40; // 40 richiamato con set
console.log("Sconto computer:", computer.sconto); // 30 (massimo
// scontato) richiamato con get

```

Definiamo una classe *Prodotto* che ha tre proprietà: *tipo*, *prezzo*, e *sconto*. La particolarità qui è che *sconto* è gestita tramite metodi *get* e *set*, che controllano il valore dell'attributo *sconto* e ne limitano il massimo a 30.

Viene dichiarata una variabile locale *_sconto* tramite *let*. Questa variabile sarà utilizzata per memorizzare il valore dello sconto interno all'oggetto *Prodotto*, ma è resa privata e non accessibile direttamente dall'esterno.

Viene utilizzato *Object.defineProperty* per definire la proprietà *sconto* dell'oggetto *Prodotto*. Questo metodo consente di definire metodi personalizzati per il recupero (*get*) e l'impostazione (*set*) del valore della proprietà. Nel metodo *get*, viene restituito il valore di *_sconto*. Nel metodo *set*, viene impostato il valore di *_sconto* e, se il nuovo valore è maggiore di 30, viene forzato a 30.

Ora, quando crei un'istanza di *Prodotto*, come *televisore* o *computer*, puoi accedere alla proprietà *sconto* come se fosse un normale attributo, ma i metodi *get* e *set* vengono eseguiti implicitamente. Ad esempio:

```
televisore.sconto = 50; imposta il valore di _sconto a 50 tramite il
metodo set.
televisore.sconto += 20; aumenta il valore di _sconto a 70 tramite
il metodo set.
televisore.sconto += 20; aumenta ulteriormente il valore di _sconto
a 80 tramite il metodo set.

console.log("Sconto televisore:", televisore.sconto); recupera il
valore di _sconto tramite il metodo get e lo visualizza nella
console. Il risultato sarà 80 per televisore.
```

Tuttavia, quando si imposta *computer.sconto = 40*;, il valore viene impostato a 40 tramite il metodo set, ma poi viene immediatamente limitato a 30 da *if (_sconto > 30) _sconto = 30*;. Quindi, quando si recupera il valore di *computer.sconto*, sarà 30 (massimo sconto).

Abbiamo imparato moltissime cose sulle proprietà degli oggetti, e, i più esperti avranno notato che Javascript è diverso dai linguaggi orientati agli oggetti “puri” come C# e codifica le funzionalità in modo leggermente differente, difatti in C#, la trasformazione dei campi in proprietà è una modifica fondamentale, quindi i campi pubblici devono essere codificati come proprietà implementate automaticamente.

In Javascript, come visto, le proprietà standard (i dati con *get* e *set* descritti sopra) sono definite dal descrittore dell'accessorio.

Ma, a parte questa breve digressione, continuiamo ad approfondire *get* e *set* studiando i loro metodi.

Utilizzare get e set

Per servirci dei metodi di *getter* e *setter* potremo specificarli all'interno di oggetti letterali o all'interno delle classi, altrimenti usando il metodo *Object.defineProperty*. Vediamo come potremo beneficiare di queste peculiarità:

```
let z = {};
Object.defineProperty(z, "prova", {
  value: "a",
  configurable: true
});
console.log(Object.getOwnPropertyDescriptor(z, "prova"));           //
Controlliamo che tutto sia ok

for(let i in z) console.log(z[i]); // z non è enumerabile
console.log(z.prova);
z.prova = "b"; // il risultato sarà sempre "a"
```

```
delete(z.prova);  
z.prova = "b"; // adesso invece è "b"  
for(let i in z) console.log(z[i]); // "b" è enumerabile, come di  
default
```

Tutto ottimo, ma si evidenziano delle problematiche legate alla gestione degli oggetti, se non vogliamo permettere al codice client qualche piccolo “*cheat*”, potremo limitare l'oggetto con tre livelli di confinamento:

Object.preventExtensions(*iltuoOggetto*) impedisce l'aggiunta di nuove proprietà al tuo oggetto. Usa **Object.isExtensible(<iltuoOggetto>)** per verificare se il metodo è stato utilizzato sull'oggetto.

Object.seal(*iltuoOggetto*) come sopra e le proprietà non possono essere rimosse (impostato come *configurable: false* su tutte le proprietà). Usa **Object.isSealed(<iltuoOggetto>)** per rilevare questa caratteristica sull'oggetto.

Object.freeze(*iltuoOggetto*) come sopra e le proprietà non possono essere modificate (impostato come *writable: false* su tutte le proprietà con descrittore di dati). La proprietà scrivibile di *Setter* non è interessata (poiché non ne ha una). Il congelamento è superficiale: significa che se la proprietà è *Object*, le sue proprietà NON SONO congelate. Usa **Object.isFrozen(<iltuoOggetto>)** per rilevarlo.

Come puoi vedere, l'approccio con **Object.defineProperty** è più arzigogolato e complicato rispetto ai metodi *getter* e *setter* quando si utilizzano valori letterali e con le classi di oggetti. Indubbiamente, se avremo bisogno di mantenere le proprietà dei nostri oggetti private, valerà la pena usufruirne.

Indipendentemente dal modo in cui li definiamo, *getter* e *setter* ci consentono di definire proprietà oggetto che vengono utilizzate come *proprietà oggetto standard*, ma sono metodi che possono eseguire codice aggiuntivo ogni volta che leggiamo o scriviamo su una particolare proprietà. Questa è una funzionalità incredibilmente utile che ci consente di eseguire la registrazione, convalidare i valori di assegnazione e persino notificare altre parti del codice quando si verificano determinate modifiche, infatti, come abbiamo imparato un *setter* è un metodo che viene eseguito ogni volta che scriviamo un valore nella proprietà corrispondente. Di conseguenza potremmo sfruttare i *setter* per eseguire un'azione ogni volta che il codice tenta di aggiornare il valore di una proprietà. Ad esempio, convalidare il

valore passato, come nel listato in basso dove *set* garantisce che alla nostra proprietà *pesoPanza* possano essere assegnati solo valori interi:

```
function Gatto() {
  let _pesoPanza = 0;

  Object.defineProperty(this, 'pesoPanza', {
    get: function() {
      return _pesoPanza;
    },
    set: function(value) {
      if (!Number.isInteger(value)) {
        throw new TypeError("Errore, il valore non è un numero intero.");
      }
      _pesoPanza = value;
    }
  });
}

const gatto = new Gatto();
gatto.pesoPanza = 50;
assert(gatto.pesoPanza === 50, "Il gatto è grasso"); // true
```

Quindi, ogni volta che viene assegnato un nuovo valore alla proprietà *pesoPanza*, controlliamo se il valore passato è un numero intero. In caso contrario, viene generata un'eccezione e la variabile privata *_pesoPanza* non verrà modificata. Se tutto è andato a buon fine e viene ricevuto un valore intero, si ottiene un nuovo valore della variabile privata *_pesoPanza*. Potremmo anche testare la situazione in cui assegniamo erroneamente un valore di un altro tipo. In tal caso, ci troveremo con un'eccezione:

```
gatto.pesoPanza = "obeso";
assert(gatto.pesoPanza === "obeso", "Il gatto è grasso"); // ???
```

In questo caso l'asserzione non sarebbe verificata, perché il valore "obeso" non può essere impostato come peso della pancia del gatto a causa della validazione del setter che accetta solo numeri interi:

```
    if (!Number.isInteger(value)) {
      throw new TypeError("Errore, il valore non è un numero intero.");
    }
  }
```

Il tentativo di assegnare un valore non intero (qui una stringa) si traduce in un'eccezione generata dal metodo *setter*.

In questo modo eviteremo tutti quei bug che si potrebbero verificare quando un valore di tipo scorretto finisce in una certa proprietà. Certo, aggiunge un sovraccarico di lavoro, ma è un prezzo che a volte dobbiamo pagare per utilizzare in sicurezza un linguaggio altamente dinamico come JavaScript.

Oltre a poter controllare l'accesso a determinate proprietà degli oggetti, *getter* e *setter* possono essere utilizzati per definire proprietà calcolate, proprietà il cui valore viene calcolato per richiesta. Le proprietà calcolate non memorizzano un valore; forniscono un metodo *get* e / o *set* per recuperare e impostare altre proprietà indirettamente. Nell'esempio seguente, l'oggetto ha due proprietà, *nome* e *razza*, che useremo per calcolare la proprietà *nomeRazza*:

```
const gattone = {
  nome: "Briciola",
  razza: "Persiano",

  get nomeRazza() {
    return this.nome + " " + this.razza},
```

Definisce un metodo *getter* su una proprietà *nomeRazza* di un oggetto letterale che calcola il valore concatenando due proprietà dell'oggetto.

```
set nomeRazza(value) {
  const pezzo = value.split(" ");
  this.nome = pezzo[0];
  this.razza = pezzo[1];
}
};
```

Definisce un metodo *setter* su una proprietà *nomeRazza* di un oggetto letterale che divide il valore passato e aggiorna due proprietà standard.

Get calcola il valore della proprietà *nomeRazza*, su richiesta, concatenando il nome e le proprietà della *razza*. *Set*, invece, utilizza il metodo incorporato *split* per dividere la stringa assegnata in segmenti. Il primo rappresenta il nome ed è assegnato alla proprietà *nome*, mentre il secondo segmento rappresenta la razza ed è assegnato alla proprietà della *razza*.

```
assert(gattone.nome === "Briciola"); // true
assert(gattone.razza === "Persiano"); // true
assert(gattone.nomeRazza === "Briciola Persiano"); // true
```

```
gattone.nomeRazza = "Kitty Siamese";
```

```
assert(gattone.nomeRazza === "Kitty Siamese"); // true
```

L'assegnazione di un valore alla proprietà *nomeRazza* richiama il metodo *set*, che calcola e assegna nuovi valori al nome e alla razza del *gattone*.

Siamo così giunti alla fine del nostro studio dei cosiddetti *getter* e *setter*. Abbiamo visto che sono un'utile aggiunta al linguaggio che può aiutarci a gestire la registrazione, la convalida dei dati e il rilevamento delle modifiche nei valori delle proprietà. Purtroppo a volte questo non basta. In alcuni casi, dobbiamo controllare tutti i tipi di interazioni con i nostri oggetti e per questo, ne possiamo usare uno completamente nuovo: il *proxy*.

Utilizzo dei proxy e i suoi vantaggi

La definizione di *proxy* in Javascript (e negli altri linguaggi di programmazione) si rifà molto alle telecomunicazioni, precisamente alle connessioni internet, dove il proxy server essenzialmente funge da intermediario tra il browser e gli altri server. Infatti nel nostro linguaggio il proxy è un surrogato attraverso il quale controlliamo l'accesso a un altro oggetto. Ci abilita per definire azioni personalizzate che verranno eseguite quando si interagisce con un oggetto, ad esempio quando viene letto o impostato un valore di proprietà o quando viene chiamato un metodo. Proxy non possiede proprietà, senza alcune funzionalità avanzate (le trappole) che vedremo, rimane solo un semplice contenitore per l'oggetto obiettivo. Potremmo pensare al proxy come ad un clone di *get* e *set*; ma essi controllano l'accesso solo a una singola proprietà dell'oggetto, mentre i proxy ti consentono di gestire genericamente tutte le interazioni con un oggetto, comprese le chiamate al metodo, quindi siamo su un livello superiore.

Infatti, sostituire *get* e *set* con i proxy, per il log, la convalida dei dati e le proprietà calcolate, ci permetterà di avere a disposizione maggiori possibilità. Potremo aggiungere facilmente la profilazione e le misurazioni delle prestazioni al nostro codice, compilare automaticamente le proprietà degli oggetti per evitare fastidiose eccezioni nulle e avvolgere gli oggetti host come il DOM per ridurre le incompatibilità tra browser. La sua sintassi:

```
let proxy = new Proxy(target, handler)
```

- *target* è l'oggetto obiettivo.
- *handler* sono i metodi del proxy, cioè le trappole che lo attiveranno.

Possiamo creare un proxy utilizzando il costruttore integrato *new Proxy()*. Cominciamo in modo semplice, cercando di intercettare tutti i tentativi di lettura e scrittura nelle proprietà di un file oggetto:

```
const auto = { nome: "Ferrari" };
const sostituto = new Proxy(auto, {
  get: (target, prop) => {
    console.log("Lettura di " + prop + " tramite un proxy");
    return prop in target ? target[prop] : undefined;
  },
  set: (target, prop, value) => {
    console.log("Scrittura di " + prop + " tramite proxy");
    target[prop] = value;
  }
});

sostituto.prezzo = "Centomila euro";
console.log(auto.prezzo); // "Centomila euro"
console.log(sostituto.prezzo); // "Centomila euro"

console.log(auto.nome); // "Ferrari"
console.log(sostituto.nome); // "Ferrari"
```

Aggiungiamo una proprietà tramite il proxy. La proprietà è accessibile sia tramite l'oggetto di destinazione che tramite del proxy.

Quindi prima creiamo il nostro oggetto *auto* di base che ha solo una proprietà *nome*. Successivamente, utilizzando il costruttore incorporato *new Proxy()*, avvolgiamo il nostro oggetto (o oggetto di destinazione) che chiameremo *sostituto*. Durante la costruzione del proxy, come secondo argomento, inviamo anche un oggetto che specifica delle *trappole*, che verranno chiamate quando determinate azioni vengono eseguite su un oggetto:

```
const sostituto = new Proxy(auto, {
  get: (target, prop) => {
    console.log("Lettura di " + prop + " tramite un proxy");
    return prop in target ? target[prop] : undefined;
  },
  set: (target, prop, value) => {
    console.log("Scrittura di " + prop + " tramite proxy");
    target[prop] = value;
  }
});
```

- **Trappola get**: viene chiamata ogni volta che tentiamo di leggere il valore di una proprietà dell'oggetto tramite il proxy. Nel codice sopra, stiamo

intercettando i tentativi di lettura delle proprietà dell'oggetto *auto* tramite il proxy sostituto. Se la proprietà esiste nell'oggetto di destinazione (*auto*), la trappola restituirà il valore di quella proprietà. In caso contrario, restituirà *undefined*. Inoltre, viene registrato un messaggio di log che indica che la proprietà è stata letta tramite il proxy.

- Trappola set: viene chiamata ogni volta che tentiamo di impostare il valore di una proprietà dell'oggetto tramite il proxy. Nel codice sopra, stiamo intercettando i tentativi di scrittura delle proprietà dell'oggetto *auto* tramite il proxy sostituto. La trappola imposta il valore della proprietà nell'oggetto di destinazione (*auto*) con il valore specificato e registra un messaggio di log che indica che la proprietà è stata scritta tramite il proxy.

In questo modo, possiamo monitorare e controllare l'accesso e la modifica delle proprietà dell'oggetto attraverso il proxy. Le trappole *get* e *set* sono solo due esempi delle molte operazioni che è possibile intercettare e personalizzare utilizzando i proxy in JavaScript. Vediamone subito un altro, sempre relativo all'esempio precedente, dove modifichiamo la riga *sostituto.prezzo*:

```
sostituto.prezzo = "Duecentomila euro"; // Assegniamo il valore
tramite il proxy
assert(auto.prezzo === "Duecentomila euro");
assert(sostituto.prezzo === "Duecentomila euro"); // true
```

Nel caso di *auto.prezzo*, stai accedendo direttamente alla proprietà e non stai attivando il getter del proxy.

Nel caso di *sostituto.prezzo*, stai accedendo tramite il proxy, e il getter del proxy viene chiamato implicitamente, restituendo *undefined* a meno che non assegni un valore tramite il proxy prima.

In poche parole, se accediamo alla proprietà *prezzo* direttamente tramite l'oggetto *auto*, viene restituito il valore *Duecentomila euro*. Ma se accediamo alla proprietà *prezzo* tramite l'oggetto proxy, il *get trap* viene chiamato implicitamente. Poiché la proprietà *prezzo* si trova nell'oggetto *auto* di destinazione, viene restituito sempre il valore *Duecentomila euro*.

È importante sottolineare che le trappole proxy vengono attivate nello stesso modo di *getter* e *setter*. Non appena eseguiamo un'azione (ad esempio, accedendo a un valore di proprietà su un proxy), la *trap* corrispondente viene

chiamata implicitamente e il motore JavaScript esegue un processo simile come se avessimo invocato esplicitamente una funzione.

D'altra parte, se accediamo a una proprietà *colore* inesistente direttamente sull'oggetto *auto* di destinazione, otterremo, un valore indefinito. Ma se proviamo ad accedervi tramite il nostro oggetto proxy, verrà attivato il gestore *get*.

Continuamo assegnando una nuova proprietà tramite il nostro oggetto proxy: *sostituto.soprannome* = "Belva". Poiché l'assegnazione viene eseguita tramite un proxy e non direttamente, il *set trap*, che registra un messaggio e assegna una proprietà al nostro oggetto bersaglio, viene attivato:

```
set: (target, prop, value) => {  
  console.log("Scrittura di " + prop + " tramite proxy");  
  target[prop] = value;  
}
```

Naturalmente, è possibile accedere alla proprietà appena creata sia tramite l'oggetto proxy e l'oggetto di destinazione:

```
sostituto.soprannome = "Belva";  
  
assert(auto.soprannome === "Belva");  
assert(sostituto.soprannome === "Belva", "Il soprannome è  
accessibile anche con il proxy");
```

Attraverso il costruttore Proxy, creiamo un oggetto che controlla l'accesso all'oggetto di destinazione attivando determinate trappole, ogni volta che un'operazione viene eseguita direttamente su un proxy.

In questo esempio, abbiamo utilizzato le trappole *get* e *set*, ma, come scritto all'inizio del paragrafo ve ne sono molte altre:

- La trappola *apply* verrà attivata quando si chiama una funzione.
- *has* intercetta le invocazioni dall'operatore *in*.
- *deleteProperty* ovviamente attivato da una rimozione delle proprietà dell'oggetto con *delete*.
- *construct* dall'operatore *new*.
- *defineProperty* attivato dalla definizione o modifica di una nuova proprietà direttamente su un oggetto.
- *getPrototypeOf* e *setPrototypeOf* verranno attivati per ottenere e impostare il valore del prototipo.

- *isExtensible* e *preventExtensions* attivati per bloccare o estendere i valori delle proprietà.

Vediamo il nostro esempio con tutte le trappole appena elencate:

```
const auto = { nome: "Ferrari" };

const sostituto = new Proxy(auto, {
  get: (target, prop) => {
    console.log("Lettura di " + prop + " tramite un proxy");
    return prop in target ? target[prop] : undefined;
  },
  set: (target, prop, value) => {
    console.log("Scrittura di " + prop + " tramite proxy");
    target[prop] = value;
  },
  apply: (target, thisArg, argumentsList) => {
    console.log("Chiamata a una funzione tramite un proxy");
    return Reflect.apply(target, thisArg, argumentsList);
  },
  has: (target, prop) => {
    console.log(`Verifica dell'esistenza della proprietà ${prop}
tramite un proxy`);
    return prop in target;
  },
  deleteProperty: (target, prop) => {
    console.log(`Cancellazione della proprietà ${prop} tramite un
proxy`);
    return Reflect.deleteProperty(target, prop);
  },
  construct: (target, argumentsList, newTarget) => {
    console.log("Chiamata di un costruttore tramite un proxy");
    return Reflect.construct(target, argumentsList, newTarget);
  },
  defineProperty: (target, prop, descriptor) => {
    console.log(`Definizione della proprietà ${prop} tramite un
proxy`);
    return Reflect.defineProperty(target, prop, descriptor);
  },
  getPrototypeOf: (target) => {
    console.log("Ottenimento del prototipo tramite un proxy");
    return Reflect.getPrototypeOf(target);
  },
  setPrototypeOf: (target, prototype) => {
    console.log("Impostazione del prototipo tramite un proxy");
    return Reflect.setPrototypeOf(target, prototype);
  },
  isExtensible: (target) => {
    console.log("Verifica dell'estensibilità tramite un proxy");
```

```

    return Reflect.isExtensible(target);
  },
  preventExtensions: (target) => {
    console.log("Prevenzione delle estensioni tramite un proxy");
    return Reflect.preventExtensions(target);
  },
});
});

```

Oltre a queste potremo anche servirci di *enumerate* attivato da un loop e molte altre, ma, attenzione, non potremo usare operazioni come: *uguaglianza* (*=*), *instanceof* e l'operatore *typeof* per problematiche legate alla coerenza dei possibili risultati.

Ora che sappiamo come funzionano i proxy e come crearli, esploriamo alcuni dei loro aspetti pratici che potranno aiutarci a semplificarci la vita da programmatore.

Utilizzo di proxy per semplificarci la vita

Uno degli strumenti più potenti quando si cerca di capire come funziona il codice o quando si prova ad arrivare alla radice di un brutto bug è il logging, l'atto di fornire informazioni che troviamo utili in un particolare momento. Potremmo, ad esempio, voler sapere quali funzioni vengono chiamate, da quanto tempo sono state eseguite, su quali proprietà vengono lette o scritte e così via.

Sfortunatamente, quando si implementa la registrazione, di solito distribuiamo le istruzioni di registrazione nel codice:

```

function Gigante() {
  let _altezza = 0;
  Object.defineProperty(this, 'altezza', {
    get: () => {
      return _altezza;
    },
    set: value => {
      _altezza = value;
    }
  });
}

```

```

const gigante = new Gigante();

```

```

console.log(gigante.altezza); // Stampa l'altezza iniziale, che è 0
gigante.altezza = 100; // Imposta una nuova altezza
console.log(gigante.altezza); // Stampa la nuova altezza, che è 100

```


Qui definiamo una funzione costruttore *Gigante* che aggiunge un *getter* e un *setter* alla proprietà *altezza*, che registra tutti i tentativi di lettura e scrittura in quella proprietà.

Notare che questa non è una soluzione ideale. Abbiamo ingombrato il nostro codice di dominio che si occupa della lettura e scrittura di una proprietà oggetto con codice di registrazione. Inoltre, se in futuro avremo bisogno di più proprietà sull'oggetto *gigante*, dobbiamo stare attenti a non aggiungere ulteriori istruzioni di registrazione a ogni nuova proprietà.

Fortunatamente, uno degli usi diretti dei proxy è abilitare la registrazione ogni volta che leggiamo o scriviamo su una proprietà, ma in un modo molto più pulito:

```
function abReg(obiettivo) {
  return new Proxy(obiettivo, {
    get: (obiettivo, proprietà) => {
      return obiettivo[proprietà];
    },
    set: (obiettivo, proprietà, numero) => {
      report("Numero " + numero + " to " + proprietà);
      obiettivo[proprietà] = numero;
    }
  });
}

let gigante = { nome: "Mario" };
gigante = abReg(gigante);

assert(gigante.nome === "Mario", "Il gigante si chiama Mario");

gigante.velocità = "lento";
assert(gigante.velocità === "lento", "Mario è lento");
```

Come vediamo dalle asserzioni possiamo leggere e scrivere nel nostro oggetto proxy. Queste azioni vengono registrate dalle trappole proxy, ma andiamo nel dettaglio: iniziamo definendo una funzione chiamata *abReg*, che accetta un oggetto di destinazione come argomento e restituisce un nuovo Proxy. Questo Proxy è dotato di un gestore che include trappole per le operazioni di lettura e scrittura (*get* e *set*). Queste trappole non solo leggono e scrivono nelle proprietà dell'oggetto, ma registrano anche informazioni su quale proprietà viene letta o scritta.

Successivamente, creiamo un oggetto chiamato *gigante* con una proprietà chiamata *nome*. Quindi applichiamo la funzione *abReg* a questo oggetto

gigante, ottenendo un nuovo oggetto proxy. Assegniamo nuovamente questo proxy all'identificatore *gigante*, sovrascrivendo il riferimento precedente, mentre l'oggetto originale viene mantenuto come oggetto di destinazione del nostro proxy.

Ogni volta che tentiamo di leggere una proprietà (come *gigante.nome*), la trappola get viene attivata e registra le informazioni sulla proprietà letta. Allo stesso modo, quando scriviamo una proprietà (ad esempio, *gigante.velocità* = "*lento*"), la trappola set viene richiamata e registra le informazioni sulla proprietà scritta.

In breve, il proxy agisce come un intermediario che monitora e registra le operazioni di lettura e scrittura sulle proprietà dell'oggetto di destinazione.

Si noti quanto questo sia molto più semplice e trasparente rispetto al modo standard di usare *getter* e *setter*. Non dobbiamo combinare il nostro codice di dominio con il nostro codice di registrazione e non è necessario aggiungere registrazioni separate per ciascuna proprietà dell'oggetto. Invece, tutte le letture e le scritture di proprietà passano attraverso i nostri metodi di trap degli oggetti proxy. La registrazione è stata specificata in un solo punto e viene riutilizzata tutte le volte che è necessario, su tutti gli oggetti necessari.

Oltre a semplificare la registrazione, i proxy possono essere utilizzati per le proprietà di compilazione automatica. Ad esempio, immagina di dover modellare la struttura delle cartelle del tuo computer, in cui un oggetto cartella può avere proprietà che possono anche essere cartelle. Ora immagina quello devi modellare un file alla fine di un lungo percorso, come questo:

```
rootCartella.gattiDir.primoGattoDir.gattoFile = "kitty.doc";
```

Per crearlo, potresti scrivere qualcosa lungo le seguenti linee:

```
const rootCartella = nuova cartella();

rootCartella.gattiDir = nuova cartella();
rootCartella.gattiDir.primoGattoDir = nuova cartella();
rootCartella.gattiDir.primoGattoDir.gattoFile = "kitty.doc";
```

Con tanti file e cartelle il listato diventerebbe piuttosto lungo e noioso. Qui è dove le proprietà di “*auto popolazione*” entrano in gioco:

```
function Cartella() {
  return new Proxy({}, {
```

```

    get: (target, proprietà) => {
        console.log("Leggo " + proprietà);

        if (!(proprietà in target)) {
            target[proprietà] = new Cartella();
        }
        return target[proprietà];
    },
    });
}

const rootFolder = new Cartella();

try {
    rootFolder.gattiDir.primoGattoDir.gattoFile = "kitty.doc";
    console.log("Tutto liscio, file trovato");
} catch (e) {
    console.error("Problema, file non trovato");
}

```

Eseguendo il codice non avremo eccezioni, questo accade perché stiamo usando un proxy. Ogni volta che accediamo a una proprietà, la trappola *get* viene attivata. Se il nostro oggetto cartella contiene già la proprietà richiesta, viene restituito il suo valore e, in caso contrario, viene creata una nuova cartella e assegnata alla proprietà. È così che vengono create due delle nostre proprietà, *gattiDir* e *primoGattoDir*, la richiesta di un valore di una proprietà non inizializzata ne attiva la creazione.

Belle le trappole, ma...

Come già sappiamo, un proxy è un oggetto surrogato attraverso il quale controlliamo l'accesso a un altro oggetto. Attraverso le cosiddette “trappole” che in parte abbiamo già elencato poche pagine fa potremmo intercettare essenzialmente qualunque interazione con un oggetto obiettivo. Un proxy può definire funzioni “trappola” che verranno chiamate ogni volta che una determinata operazione viene eseguita su un proxy, ma.. c’è un prezzo da pagare, vediamo un listato e arriviamo al dunque:

```

const prova = {
    saluto: "ciao",
    nome: "da Mario"
};

const trappola = {
    get(target, prop, receiver) {

```

```

    return `ciao da Nino ${target[prop]}`;
  }
};

const proxy2 = new Proxy(prova, trappola);

console.log(proxy2.saluto); // ciao da Nino ciao
console.log(proxy2.nome); // ciao da Nino da Mario

```

Qui sopra tramite la costante *trappola* abbiamo modificato le proprietà iniziali.

Potremo anche utilizzare il costruttore *Proxy()* e il metodo *Proxy.revocable()* per revocare le proprietà di un oggetto, possiamo usare queste *trappole* per implementare diverse funzionalità utili, ma c'è il rovescio della medaglia: tali attività influiscono sulle prestazioni generali, ad esempio, in un browser enormemente utilizzato come Chrome i proxy sono abbastanza lenti, qualcosa meno in Firefox e negli altri, anche se molti aggiornamenti recenti hanno migliorato la situazione. In ogni caso questo vuol dire che dovremo fare molta attenzione al loro utilizzo e effettuare diversi test prima di decidere se includerli o meno.

Quiz & Esercizi

- 1) Quale istruzione bisognerà utilizzare per aggiungere un metodo al prototipo dopo che l'oggetto è stato creato?
- 2) Il prototipo come oggetto di una funzione potrà creare istanze di esso attraverso il costruttore?
- 3) Correggere il listato in modo che *provo* sia anche un costruttore:

```

function Prova() {
  const prova = new Prova();
  const provo = new Prova();
}

```

- 4) Per copiare due prototipi potremo utilizzare il seguente codice anche se dopo cambiassero i valori solo del primo (il secondo non dovrebbe variare i suoi valori)?

```

Primo.prototype = Secondo.prototype

```

- 5) Qual è il modo migliore di creare oggetti ed implementarne l'ereditarietà?
- 6) Per creare un proxy avremo bisogno di un costruttore o basterà semplicemente evocarlo?

- 7) Scrivere un listato utilizzando un proxy per recuperare dati, cioè un file *mario.txt* da una cartella *mario* su un server.
- 8) Crea un costruttore *Persona* che accetta un nome e un'età come argomenti e ha un metodo *saluta* nel suo prototipo che restituisce un messaggio di saluto personalizzato.
- 9) Crea una funzione costruttrice *Studente* che erediti da *Persona* e abbia una proprietà aggiuntiva *corso* e un metodo *descrizione* che combini la descrizione del genitore con il corso dello studente.
- 10) Crea un oggetto *veicolo* e aggiungi un metodo *avvia* al suo prototipo che restituisce "Il veicolo è in marcia". Crea un oggetto *auto* che erediti da *veicolo* e abbia un metodo aggiuntivo *frena* che restituisce "L'auto si è fermata".
- 11) Aggiungi un metodo *contaVocali* al prototipo della classe *String* che restituisce il numero di vocali presenti nella stringa.
- 12) Aggiungi un metodo *somma* al prototipo di *Array* che calcoli la somma di tutti gli elementi dell'array.
- 13) Crea una classe *Cerchio* che abbia un attributo *raggio*. Implementa getter e setter per l'attributo *raggio* in modo che il raggio non possa essere negativo. Inoltre, crea un getter per calcolare e restituire l'area del cerchio.
- 14) Crea una classe *Persona* che abbia un attributo privato *_eta* e implementa un getter e un setter per l'età. Il setter dovrebbe consentire di impostare l'età solo se è compresa tra 0 e 120.
- 15) Crea una classe *Triangolo* che accetti i lati *a*, *b* e *c* come attributi. Implementa un getter per calcolare e restituire il perimetro del triangolo. Utilizza la formula del perimetro: $\text{perimetro} = a + b + c$.
- 16) Crea un proxy per un oggetto che controlla l'accesso alle sue proprietà in modo che possa essere ottenuto solo se l'utente ha i permessi corretti.
- 17) Crea un proxy per un oggetto che tiene traccia delle modifiche apportate alle sue proprietà.
- 18) Crea un proxy per un oggetto che controlla se i dati assegnati alle sue proprietà soddisfano determinate condizioni di validazione.

Riassunto

Capire bene il concetto di chiusura e i contesti di esecuzione o anche di validità delle variabili ci permetterà di pianificare al meglio la stesura del codice JS, focalizzando la nostra attenzione sull'evitare problemi di sovrascrittura o duplicazione di variabili, ma anche sfruttando al meglio le

aree in cui un valore è valido o meno. Scopriremo anche che callback, chiusure e contesto di esecuzione sono strettamente legati tra di loro.

Approfondire tutte le possibilità offerte da variabili fisse o mutabili e dalla loro posizione farà salire di livello la nostra comprensione di JS.

10 - Modularizzazione del codice

I moduli prima di ES6

I moduli con ES6

Import e Export

Anche dopo aver esplorato solo una parte delle possibilità offerte da questo linguaggio probabilmente arriveremo a scrivere dei listati abbastanza consistenti. Sappiamo benissimo che questo comporta una serie di problematiche, sia di gestione sia di chiarezza, pulizia e comprensione. Per questo motivo esistono i moduli. Essenzialmente si tratta di parti di programma separate contenenti classi o librerie di funzioni. Sono dei file che al bisogno verranno chiamati ed eseguiti dal codice principale.

Ad esempio:

```
// moduloSaluto.js
export function moduloSaluto(user) {
  alert(`Ciao, ${user}!`);
}
```

Il modulo potrà essere importato successivamente:

```
// listato.js
import {moduloSaluto} from './moduloSaluto.js';

alert(moduloSaluto); moduloSaluto('Mario'); // Ciao, Mario!
```

In questo semplice esempio si percepisce poco, tuttavia i moduli sono unità di organizzazione del nostro codice rispetto agli oggetti e alle funzioni, ci permettono di dividere i programmi in diverse sezioni comunque unite da un codice principale. Durante la creazione dei moduli, dovremmo sforzarci di formare astrazioni coerenti e incapsulare i dettagli di implementazione. Ciò rende più facile ragionare sulla nostra applicazione, perché non ci preoccupiamo di dettagli frivoli quando si utilizza la funzionalità del nostro

modulo. Inoltre, disporre di moduli significa che possiamo riutilizzare facilmente le sue funzionalità in diverse parti delle nostre applicazioni e anche in altre, accelerando notevolmente il nostro processo di sviluppo.

Come abbiamo visto in precedenza, JavaScript ci aiuta parecchio con le variabili: ogni volta che ne definiamo una nella riga principale del codice, essa viene automaticamente considerata globale ed è possibile accedervi da qualsiasi altra parte del nostro listato. Questo potrebbe non essere un problema per i piccoli programmi, ma quando le nostre applicazioni iniziano a crescere e includiamo codice di terze parti, la possibilità che si verifichino conflitti di denominazione inizia a crescere in modo significativo. Nella maggior parte degli altri linguaggi di programmazione, questo problema viene risolto con spazi dei nomi (C ++ e C #) o pacchetti (Java), che racchiudono tutti i nomi racchiusi in un altro nome, riducendo quindi potenziali duplicazioni.

JavaScript non offriva una funzionalità integrata di livello superiore che ci consentisse di raggruppare le variabili correlate in un modulo, tramite lo spazio dei nomi o con pacchetti finché ES6 introdusse finalmente i moduli nativi.

Modularizzazione del codice in JavaScript pre-ES6

JavaScript precedente a ES6 ha solo due tipi di ambiti: ambito globale e ambito della funzione. Come detto, non ha qualcosa in mezzo, uno spazio dei nomi o un modulo che ci consentirebbe di raggruppare alcune funzionalità insieme. Per scrivere codice modulare, gli sviluppatori JavaScript erano costretti a essere creativi con le funzionalità del linguaggio esistenti.

Al momento di decidere quali funzionalità utilizzare, bisognava tenere presente che, in un minuto, ogni sistema di moduli doveva essere in grado di eseguire le seguenti operazioni:

- Definire un'interfaccia attraverso il quale si possa accedere alle funzionalità offerte dal modulo.
- Nascondere le parti interne del modulo in modo tale che gli utenti non siano gravati da tutta una serie di dettagli di implementazione non importanti. Così facendo le avremo anche protette dal mondo esterno, prevenendo modifiche indesiderate che possano portare a qualunque effetto collaterale o bug.

Si utilizzava Asynchro-Module Definition (AMD) e CommonJS, i due standard di specifica dei moduli più popolari.

AMD è stata progettata esplicitamente pensando al browser, mentre CommonJS per un ambiente JavaScript generico (come i server, con Node.js), senza essere vincolato alle limitazioni del browser.

Attualmente queste tipologie di moduli sono considerate desuete e si possono trovare solo su vecchi programmi. Dal 2015 questo sistema è stato standardizzato (con ECMAScript 6) per poi evolversi negli anni a seguire, basandosi sempre su Node.js.

I moduli con ECMAScript 6 (ES6)

I moduli ES6 sono progettati per unire i vantaggi di CommonJS e AMD, hanno una sintassi relativamente semplice e sono basati su file (un modulo per file), fornendo anche il supporto per il loro caricamento asincrono.

L'idea principale alla base è che solo gli identificatori esportati esplicitamente da un modulo sono accessibili dall'esterno di quel modulo. Tutti gli altri identificatori, anche quelli definiti nell'ambito di primo livello (l'ambito globale in JavaScript standard), sono accessibili solo dall'interno del modulo.

Per fornire questa funzionalità, come abbiamo visto nell'esempio iniziale vengono introdotte due nuove parole chiave:

export (per rendere disponibili determinati identificatori dall'esterno del modulo).

import (per importare gli identificatori dei moduli esportati).

Potrebbe sembrare tutto molto chiaro e semplice, ma vi sono diverse sfumature da comprendere che vedremo più avanti, intanto è bene sapere che lo *strict mode* è automaticamente abilitato all'interno dei moduli, quindi, se per esempio tentiamo di assegnare un valore ad una variabile non dichiarata riceveremo un messaggio di errore.

Cominciamo con un semplice esempio che mostra come esportare funzionalità da un modulo e importarlo in un altro.

```
const gatto = "Coda:";

export const messaggio = " miao miao!";
export function salutoFelino() {
  return gatto + " " + salutoFelino;
}
```

Prima definiamo una variabile *gatto*, che sarà accessibile solo all'interno del modulo, anche se è collocata nel codice di primo livello (il che la renderebbe

una variabile globale nel codice pre-ES6).

Successivamente, definiamo un'altra variabile di primo livello, *messaggio*, che rendiamo accessibile dall'esterno del modulo utilizzando la nuova parola chiave *export*. Infine, creiamo anche e esportiamo la funzione *salutoFelino*.

E questo è tutto! Questa è la base che dobbiamo conoscere per definire i nostri moduli. Non bisogna usare funzioni immediate o ricordare alcuna sintassi esoterica per esportare funzionalità da un modulo. Scriviamo il nostro codice come scriveremmo il codice JavaScript standard, con l'unica differenza che anteponiamo alcuni degli identificativi, ad esempio variabili, funzioni o classi, con una parola chiave di esportazione.

Prima di imparare a importare questa funzionalità esportata, daremo un'occhiata a un modo alternativo per esportare gli identificatori: Elenchiamo tutto ciò che vogliamo esportare alla fine di un modulo, come mostrato nell'esempio seguente:

```
const cane = "Fido:";
const messaggio = " bau bau!";

function salutoCanino() {
  return cane + " " + messaggio;
}
export {messaggio, salutoCanino};
```

Indipendentemente da come abbiamo esportato gli identificatori di un certo modulo, se abbiamo bisogno di importarli in un altro modulo, dobbiamo usare la parola chiave *import*, come nell'esempio seguente:

```
import {messaggio, salutoCanino} from "Cane.js";

assert(messaggio === " bau bau!", "Possiamo accedere alla variabile
importata");
assert(salutoCanino () === "Fido: bau bau!", "Fido ci saluta
dall'esterno del modulo");
assert(typeof cane === "undefined", "Fido non è raggiungibile da
fuori");
```

Per importare la variabile *messaggio* e la funzione *salutoCanino* dal modulo, utilizzeremo, come già detto, *import*. Per accedervi dovremo creare un file *Cane.js* con all'interno il codice visto precedentemente:

```
import {messaggio, salutoCanino} from "Cane.js";
```

In questo modo, abbiamo ottenuto l'accesso a questi due identificatori definiti nel modulo *Cane*. Infine, possiamo verificare che possiamo accedere alla variabile del messaggio e chiamare la funzione *salutoCanino*:

```
assert(messaggio === " bau bau!", "Possiamo accedere alla variabile
importata");
assert(salutoCanino() === "Fido:  bau  bau!", "Fido ci saluta
dall'esterno del modulo");
```

Quello che non possiamo fare è accedere alle variabili non esportate e non importate. Per esempio, non possiamo accedere alla variabile *Cane* perché non è contrassegnata con *export*:

```
assert(typeof cane === "undefined", "Fido non è raggiungibile da
fuori");
```

Con i moduli, siamo finalmente un pò più al sicuro dall'uso improprio delle variabili globali. Qualunque cosa che non abbiamo contrassegnato esplicitamente per l'esportazione rimane ben isolato all'interno di un modulo. Nell'esempio precedente, abbiamo esportato alcuni identificatori dal modulo *Cane.js*, cioè *messaggio* e *salutoCanino*. Potrebbe esserci il caso in cui dovremmo elencarne una lunga lista, magari più di cento, e questo potrebbe risultare alquanto noioso. Per fortuna esiste una scorciatoia per sopperire a tutto questo: il simbolo *** posto dopo il comando *import* farà sì che tutto gli identificatori del modulo siano importati:

```
import * as caneModule from "Cane.js";
```

```
assert(caneModule.messaggio === " Bau bau!", "Possiamo accedere alla
variabile importata");
assert(caneModule.salutoCanino() === "Fido:  Bau  bau!", "Fido ci
saluta dall'esterno del modulo");
assert(typeof caneModule.cane === "undefined", "Fido resta
inaccessibile dall'esterno");
```

Come appena visto per importare tutti gli identificatori esportati da un modulo, usiamo *import ** in combinazione con un identificatore che useremo per fare riferimento all'intero modulo (in questo caso, l'identificatore *caneModule*). Dopo averlo fatto, possiamo accedere agli identificatori esportati tramite la notazione delle proprietà; ad esempio, *caneModule.messaggio*, *caneModule.salutoCanino*. Si noti che non

possiamo ancora accedere alle variabili di primo livello che non sono state esportate, come nel caso della variabile *cane*.

Export con default

Spesso non vogliamo esportare un insieme di identificatori correlati da un modulo, ma vogliamo invece rappresentare l'intero modulo attraverso una singola esportazione. Una situazione abbastanza comune in cui ciò si verifica è quando i nostri moduli contengono una singola classe, come nell'elenco di seguito:

```
export default class Cane {constructor(nome) {this.name = nome;}}  
  
export function comparePets(pet1, pet2) {return pet1.name ===  
pet2.name;  
}
```

Qui abbiamo aggiunto la parola chiave *default* dopo *export*, che specifica l'associazione “predefinita” per questo modulo. Nel caso specifico, per questo modulo è la classe denominata *Cane*, e anche se abbiamo specificato un'associazione predefinita, potremo ancora utilizzare le esportazioni per gli identificatori aggiuntivi, come abbiamo fatto con la funzione di confronto dei Pets (*comparePets*).

Adesso, possiamo usare la sintassi semplificata per importare funzionalità da *Cane.js*, come mostrato nel seguente elenco:

```
import ImportedPet from "Cane.js";  
import {comparePets} from "Cane.js";  
  
const pet1 = new ImportedPet("Fido");  
const pet2 = new ImportedPet("Kitty");  
  
assert(pet1! == undefined && pet2! == undefined, "Possiamo creare un  
paio di Pets");  
  
assert(!comparePets (pet1, pet2), "Possiamo confrontare i pet");
```

All'inizio del paragrafo siamo partiti con un'esportazione predefinita. Mentre adesso abbiamo usato un più chiaro *import* senza le parentesi graffe. Inoltre, notiamo che potremo scegliere un nome arbitrario per fare riferimento a quell'esportazione, non c'è l'obbligo di mantenere il precedente. In questo esempio, *ImportedPet* si riferisce alla classe *Cane* definita nel file *Cane.js*.

L'esempio prosegue eseguendo un importando un'esportazione con nome, come negli esempi precedenti, solo per illustrare che possiamo avere sia un'esportazione predefinita che un numero di esportazioni con nome all'interno di un singolo modulo. Infine, istanziamo un paio di oggetti pet (*pet1* e *2*) e chiamiamo la funzione *comparePets*, per confermare che tutte le importazioni funzionano come dovrebbero.

In questo caso, entrambe le importazioni vengono effettuate dallo stesso file. Dopo tanta fatica non arrabbiatevi se vi sveliamo che esiste una sintassi abbreviata:

```
import ImportedPets, {comparePets} from "Cane.js";
```

Attraverso l'operatore virgola potremo importare sia le esportazioni predefinite che quelle denominate dal file *Cane.js*, in una singola istruzione.

Rinominare importazioni ed esportazioni

Se necessario, possiamo anche rinominare sia le esportazioni che le importazioni.

Vediamo in che modo:

```
// Pranzo.js
function primoPiatto() {return "Lasagne";
}

assert(typeof primoPiatto === "function" && typeof coseBuone ===
"undefined", "All'interno del modulo possiamo accedere solo a
primoPiatto");

export {primoPiatto as coseBuone}

// Pranzo.js
import {coseBuone} from "Pranzo.js";

assert(typeof primoPiatto === "undefined" && typeof coseBuone ===
"function", "Durante l'importazione, possiamo accedere solo
all'alias");
```

Nell'esempio precedente, definiamo una funzione chiamata *primoPiatto* e testiamo che possiamo accedere alla funzione solo tramite l'identificatore *primoPiatto* e non tramite l'alias *coseBuone* che forniamo alla fine del modulo tramite la parola chiave *as*:

```
export {primoPiatto as coseBuone}
```

Possiamo eseguire l'esportazione solo all'interno del modulo e non antepoendo la dichiarazione di variabile o funzione alla parola chiave *export*.

Quindi, quando eseguiamo un'importazione dell'esportazione rinominata, dobbiamo riferirci al file indicando l'alias fornito:

```
import {coseBuone} from "Pranzo.js";
```

Infine, testiamo di avere accesso all'identificatore con l'alias, ma non a quello originale:

```
assert(typeof primoPiatto === "undefined" && typeof coseBuone ===  
"function", "Durante l'importazione, possiamo accedere solo  
all'alias");
```

La situazione è simile quando si rinomina le importazioni, come mostrato nel seguente segmento di codice:

```
// Aperitivo.js  
export function fame() {return "Patatine";  
}
```

```
// Cena.js  
export function bigfame() {return "Pasta";  
}
```

```
// pasti.js  
import {fame as mangiaPatatine} from "Aperitivo.js";  
import {bigfame as mangiaPasta} from "Cena.js";
```

```
assert(typeof fame === "undefined", "Non possiamo accedere a fame");  
assert(mangiaPatatine() === "Patatine" && mangiaPasta() === "Pasta",  
"Possiamo accedere a identificatori con alias!");
```

Analogamente all'esportazione di identificatori, possiamo anche utilizzare la parola chiave *as* per creare alias durante l'importazione di identificatori da altri moduli. Questo è utile quando abbiamo bisogno di fornire un nome migliore che sia più adatto al contesto corrente.

Utilizzare “*import.meta*”

Tramite l'oggetto *import.meta* saremo in grado di visualizzare i metadati, ad esempio un URL, all'interno di un modulo, queste informazioni

dipenderanno dal contesto di esecuzione:

```
// modulo.js
function doveSono() {
  console.log(import.meta);           //           {           url:
  "file:///home/utente/modulo.js" }
}
```

doveSono();

La funzione *doveSono* utilizza *import.meta* per accedere all'URL del modulo corrente e stamparlo nella console. L'URL visualizzato dipenderà dal percorso effettivo del modulo nel sistema del tuo computer.

Osserviamo ora un esempio più complesso che dimostra come utilizzare *import.meta* in un contesto di moduli e come i metadati possono essere utili:

Supponiamo di avere due moduli: *main.js* e *utility.js*. Il modulo *utility.js* esporta una funzione *calcola* che effettua un calcolo basato su alcune informazioni sui metadati del modulo. Il modulo *main.js* importa questa funzione da *utility.js* e la utilizza:

utility.js:

```
export function calcola(a, b) {
  // Utilizziamo import.meta.url per ottenere l'URL del modulo
  // corrente
  const url = new URL(import.meta.url);
  console.log(`Eseguito da: ${url.pathname}`);
  return a + b;
}
```

main.js:

```
import { calcola } from './utility.js';
```

```
const risultato = calcola(10, 5);
console.log(`Risultato: ${risultato}`);
```

Come detto, stiamo importando la funzione *calcola* dal modulo *utility.js* nel modulo *main.js*. All'interno della funzione *calcola* nel modulo *utility.js*, utilizziamo *import.meta.url* per ottenere l'URL del modulo corrente. Poi, stampiamo l'URL nella console per mostrare da quale modulo è stato eseguito il calcolo.

Quando eseguiamo *main.js*, otterremo un output simile al seguente:

```
Eseguito da: /percorso-assoluto-del-progetto/utility.js
Risultato: 15
```

L'URL nel metadato `import.meta.url` indica il percorso del modulo da cui è stato eseguito il codice, il che può essere utile per scopi di debug o per determinare l'origine di una funzione o variabile in un'applicazione modulare più complessa.

Caricamento dei moduli

I moduli solitamente vengono caricati in sequenza, quindi il primo nel listato sarà anche quello che verrà eseguito subito, di seguito gli altri. In realtà la sequenza prevede che venga innanzitutto attuato il codice HTML, ad esempio:

```
<script type="module">
  .... // questo script attenderà prima di essere eseguito
</script>
<script>
  .... // questo invece verrà eseguito subito
</script>
```

Tutto ciò potrebbe essere un problema, dato che la pagina HTML viene visualizzata mentre viene eseguita, forse prima che la nostra applicazione sia pronta. In questo caso avremo dalla nostra parte il comando *async*, il quale ci permette di far partire subito la nostra istruzione, anche prima del codice HTML:

```
<script async type="module">
  import {contatore} from './analisiidatisitoweb.js';
  contatore.count();
</script>
```

Questo ci servirà sicuramente per avviare pubblicità o analisi di statistiche varie.

Analisi di performance

Oltre alla scrittura modulare del codice, l'analisi delle prestazioni è un aspetto fondamentale nello sviluppo di applicazioni JavaScript di alta qualità. Nonostante i notevoli miglioramenti delle prestazioni nei motori JavaScript, è importante scrivere codice efficiente per garantire che le tue applicazioni siano veloci e reattive. Per prima cosa è buona pratica misurare il tempo di esecuzione.

Per misurare il tempo di esecuzione di una determinata porzione di codice, JavaScript fornisce due metodi molto utili: *console.time* e *console.timeEnd*.

Questi metodi ci consentono di calcolare con precisione quanto tempo impiega il nostro codice per eseguire una specifica operazione.

Ecco un esempio pratico:

```
console.time('Tempo di Risposta');  
// Simuliamo un'operazione che richiede tempo  
for (let i = 0; i < 1000000; i++) {  
  // Operazioni complesse...  
}  
console.timeEnd('Tempo di Risposta');
```

In questo esempio, utilizziamo `console.time` per avviare un timer denominato "Tempo di Risposta" prima di eseguire un ciclo `for` che simula un'operazione computazionalmente intensiva. Al termine dell'operazione, utilizziamo `console.timeEnd` con lo stesso nome per fermare il timer e visualizzare il tempo trascorso nel terminale del browser.

L'analisi delle prestazioni ti consente di identificare i punti critici del tuo codice e di migliorarli per renderli più efficienti. Puoi utilizzare queste informazioni per ottimizzare al meglio le tue applicazioni e migliorare l'esperienza dell'utente.

Ecco alcune tecniche comuni per ottimizzare il codice:

Profiling del codice: utilizza strumenti di sviluppo del browser, come il "Profiler," per identificare le parti del codice che richiedono più tempo di esecuzione.

Cache dei risultati: se una parte del tuo codice esegue calcoli costosi, considera di memorizzare nella cache i risultati per evitare calcoli ridondanti.

Algoritmi ottimizzati: scegli algoritmi efficienti per risolvere problemi complessi.

Minimizzazione e compressione: riduci il carico del tuo sito web utilizzando strumenti di minimizzazione e compressione per i file JavaScript.

Eliminazione del codice morto: rimuovi il codice inutilizzato o non raggiungibile per semplificare il carico della pagina.

Risorse ottimizzate: carica solo le risorse necessarie per la pagina e sfrutta la cache del browser.

Async e Await: utilizza `async` e `await` per gestire le chiamate asincrone in modo più efficiente.

In poche parole, l'analisi delle prestazioni è un componente essenziale dello sviluppo JavaScript. Utilizzando i metodi di misurazione del tempo e seguendo le migliori pratiche di ottimizzazione, puoi assicurarti che le tue applicazioni siano veloci, reattive e pronte per l'uso da parte degli utenti.

Quiz & esercizi

1) Completare il listato per ottenere come risultato: la somma è 22

```
import {modulo} from './modulo.js';
```

2) Utilizzare il metodo *unshift* per modificare la posizione del valore contenuto in `modulo.js` da seconda a prima nel listato precedente.

3) Utilizzare il metodo `reduce` per ottenere zero come risultato.

4) Creare un Set e utilizzare l'intersezione per accomunare i vettori.

5) Creare un modulo chiamato "math" che esporta due funzioni: "add" e "subtract". La prima dovrebbe prendere due parametri e restituire la loro somma, mentre la seconda dovrebbe prendere due parametri e restituire la loro differenza. Testare il modulo importandolo in un file separato e utilizzando le funzioni esportate per eseguire alcuni calcoli.

6) Creare un modulo chiamato "person" che esporta un oggetto con le proprietà "name" e "age". Importare il modulo in un file separato e utilizzarlo per creare un'istanza di una persona. Aggiungere un metodo chiamato "greet" all'oggetto esportato che registra un saluto sulla console utilizzando il nome della persona.

7) Creare un modulo chiamato "shoppingCart" che esporta un oggetto con una proprietà di array chiamata "items". L'oggetto dovrebbe avere i metodi "add" e "remove" che aggiungono o rimuovono un elemento dall'array degli elementi e un metodo "total" che restituisce il totale di tutti gli elementi. Importare il modulo in un file separato e utilizzarlo per gestire un carrello della spesa.

8) Creare un modulo chiamato "logger" che esporta una funzione che prende in un messaggio e un livello di gravità, e registra il messaggio sulla console con il livello di gravità. Importare il modulo in un file separato e utilizzarlo per registrare messaggi in tutto il codice.

9) Creare un modulo chiamato "utility" che esporta una raccolta di funzioni di utilità come "isEmail", "isPhoneNumber", "isValidPassword". Importare il modulo in un file separato e utilizzarlo per convalidare alcuni input.

Riassunto

In questo paragrafo abbiamo trattato le operazioni fondamentali che eseguiremo con i vettori, array o insiemi, quindi con tutte quelle raccolte di stringhe o numeri con cui avremo a che fare sicuramente nel corso della creazione di un'applicazione.

11 – JSON, JQUERY, AJAX e THREE.JS

JSON e XML

Proviamo ad usare JQUERY

Accenni su AJAX

Parliamo di Three.js

In questo capitolo affronteremo quattro argomenti differenti i quali sembrerebbero accomunati da acronimi abbastanza simili, non certo per la loro somiglianza, più per il fatto di essere abbastanza criptici. Infatti, come vedremo, si tratta di quattro “possibilità” totalmente diverse, anche se, approfondendo, noteremo che ci saranno dei casi dove potremo interfacciarle tutte tra di loro, soprattutto le prime tre. In ogni caso, purtroppo non avremo abbastanza spazio per affrontare ogni argomento in maniera esaustiva, quindi sarà un ottimo esercizio quello di proseguire sulla traccia di questi paragrafi, e sicuramente in molti lo faranno appena si renderanno conto delle potenzialità offerte.

JSON e XML

JSON (*JavaScript Object Notation*) è un formato per scambiare dati in maniera che siano leggibili e chiari per chi li scrive e per chi dovrà interpretarli, cioè il computer. Possiamo anche semplificare dicendo che è il modo con cui si possono riprodurre degli oggetti javascript con una sintassi ridotta, in modo che successivamente sia riproducibile in una stringa. JSON è indipendente da Javascript e si basa su di una struttura convenzionale, riconoscibile da chi ha familiarità con qualsiasi altro linguaggio di programmazione, proprio perché JSON ha come scopo l’interscambio di dati. Dal 2009 è parte integrante di Javascript, dove essenzialmente convertirà in una stringa il nostro oggetto che, ad esempio, intendiamo inviare ad un altro computer in rete, o memorizzarlo in un file. Questo processo di conversione è definito *serializzazione*, mentre il processo inverso è chiamato *deserializzazione*.

Per convertire un oggetto in una stringa, è necessario un formato dati che specifichi come l'oggetto dovrebbe essere mappato su una stringa di caratteri, ad esempio come si fa a denotare la proprietà e i valori di un oggetto e come si codificano i vari tipi di dati come numeri e matrici? Storicamente la maggior parte dei formati di dati sono binari, ciò significa che non è possibile per un essere umano leggere i dati formattati e ottenere una comprensione della sua struttura o del significato, dato che in genere, come molti sapranno, questi dati si presentano come lunghe serie di 0 e 1.

Negli ultimi anni, c'è stato un passaggio verso i formati di dati in testo normale. L'esempio principale è XML, che utilizza una struttura e una sintassi simili all'HTML per differenziare le proprietà dai loro valori.

Non c'è nulla di male nell'usare XML per serializzare oggetti JavaScript e oramai molte applicazioni web la utilizzano. Ma perché mai dovremmo convertire i nostri dati in XML? proprio perché questo linguaggio è utilizzato per trasportare dati, anche tra programmi applicativi o tra sistemi operativi differenti. Quindi, riassumendo, se avessimo la necessità di esportare dei dati, ad esempio, dal listato Javascript che gestisce un sito di e-commerce, ad un applicativo installato sul nostro computer per analizzare le statistiche di vendita, succederà questo: tramite JSON i dati verranno convertiti in stringhe le quali saranno così comprensibili ed utilizzabili da una libreria di conversione in XML. A quel punto l'applicativo importerà il file XML e sarà in grado di visualizzare le statistiche. Senza questo procedimento e questi ponti di collegamento tutto sarebbe molto più criptico e chiuso.

Vediamo un esempio di come è strutturato un file XML:

```
<?xml version="1.0" codifica="UTF-8"?>
<prodotti>
<nomeProdotto>Miele</nomeProdotto>
<prezzo>10 euro</prezzo>
<peso>500 grammi</peso>
<gusto>Arancia</gusto>
<quantità>25</quantità>
<inserito>01-01-2023</inserito>
<azienda produttrice>
<nome>L'Alveare</nome>
```

Una struttura di questo tipo è molto simile al codice Javascript, anche se diversa. Per la conversione avremo bisogno di una libreria (in rete ve ne

sono diverse, ad esempio: <https://goessner.net/download/prj/jsonxml/>) dato che, come accennato, Javascript non supporta XML.

Il file visto precedentemente in Javascript apparirebbe così:

```
prova = {
  nomeProdotto: "Miele",
  prezzo:"10 euro",
  peso:"500 grammi",
  gusto:"Arancia",
  quantità:25,
  inserito: dataOggi
  azienda produttrice:{
    nome: "L'Alveare"
  },
}
```

Per procedere con la conversione abbiamo bisogno della funzione *JSON.stringify()*:

```
prodotti = JSON.stringify(prova)
"{\"nomeProdotto\":\"Miele\", \"prezzo\":\"10 euro\", \"peso\":\"500 grammi\", \"gusto\":\"Arancia\", \"quantità\":25, \"inserito\":\"01-01-2022\", \"azienda produttrice\":{\"nome\":\"L'Alveare\"},\"}"
```

Si noti che le proprietà e i valori vengono tutti mantenuti, e i nomi delle proprietà sono automaticamente racchiusi tra virgolette. Quindi, quando useremo *stringify*:

- Le stringhe appariranno sempre tra virgolette doppie.
- I numeri non hanno bisogno delle virgolette; ad esempio, la proprietà *quantità* ha un valore di 25.
- I valori booleani sono rappresentati senza virgolette con le parole chiave *true* o *false*, lo stesso vale anche per *undefined* o *null*.
- Gli oggetti possono incapsulare oggetti figlio.
- Gli Array possono essere utilizzati come valore per qualsiasi proprietà con le parentesi quadre, mentre gli elementi sono separati da virgole.

JSON consente di rappresentare tutti i tipi di dati JavaScript nella versione serializzata, e, quando la stringa viene riconvertita in un oggetto, le varie proprietà mantengono tutti i dati originali.

Per riconvertire la stringa in un oggetto useremo la funzione *JSON.parse()*:

```
recupero = JSON.parse(prodotti);  
typeof recupero  
recupero.nomeProdotto  
recupero.quantità
```

Sebbene il formato JSON sia ottimo per la serializzazione delle proprietà, non può essere utilizzato con i metodi, verranno semplicemente ignorati quando si utilizza *JSON.stringify()*.

JQUERY

Nonostante la possibile similitudine letterale JSON non ha nulla a che vedere con JQUERY. Difatti quest'ultima è una libreria open source, scritta sempre in Javascript, perlopiù orientata verso le pagine Web, le sue animazioni, la gestione del codice HTML, gli eventi, e tanto altro.

JQuery (jquery.com) ci interessa dato che è molto diffuso su internet per diversi buoni motivi:

- Fornisce un'API intuitiva e di facile apprendimento per eseguire le attività più comuni dello sviluppo del sito web.
- È abbastanza semplice scrivere plugin per migliorare le capacità di jQuery, e ci sono tantissime librerie di plugin disponibili gratuitamente su Internet.
- Esiste un'ampia comunità di sviluppatori pronta ad aiutarci, purtroppo soprattutto in inglese. Comunque esiste anche qualcosa in italiano.
- Probabilmente è la libreria più utilizzata per Javascript.

Quindi ci sono tanti buoni motivi per dare un'occhiata alle sue funzionalità, nonostante si tratti di una libreria un po' datata. Per utilizzare jQuery bisogna scaricare l'ultima versione dal sito jquery.com, controllando se vi sono eventuali incompatibilità con il browser che intendiamo utilizzare. Vi sono due versioni, una compressa, l'altra no. Inizialmente useremo la seconda, sarà tutto più semplice. La versione compressa ha dei vantaggi in termini di velocità che potranno essere sfruttati in tanti altri casi che vedremo in un secondo momento senza il supporto di questo testo.

Ora, importiamo in maniera molto semplice la libreria nella pagina web usando un tag sorgente nell'intestazione del documento:

```
<script src="jquery-3.6.js"></script> // ovviamente controlliamo la  
versione scaricata e l'esatta stringa da inserire.
```

Se scegliessimo di caricare jQuery tramite una CDN (*Content Delivery Network*, un'infrastruttura di rete estesa a livello globale, utile per velocizzare alcuni processi), non sarà necessario scaricare jQuery; dovremo semplicemente aggiungere il seguente tag sorgente nella sezione head della pagina web:

```
<script src=http://ajax.googleapis.com/ajax/libs/jquery/3.6/jquery.min.js></script>
```

Se carichiamo la pagina, potremo verificare se jQuery è installato. Per confermarlo, digitiamo jQuery nella console.

Proseguendo scopriremo come metodi e proprietà siano molto simili se non uguali nella sintassi o nella grammatica, e questo in parte è vero, ma non del tutto esatto, quindi, anche se siamo sempre utilizzando Javascript, dovremmo approcciarci a questo ambiente consci del fatto che le differenze, anche se piccole, ci sono.

Selettori

Le differenze di base con Javascript iniziano con i selettori. La sintassi per indicare un id che usavamo, cioè: id = "Mario" diventerà: `$("#Mario")`. Il simbolo del dollaro è un alias della funzione jQuery e ha lo scopo di risparmiare sulla digitazione, infatti: `jQuery("#Mario")` equivale alla sintassi precedente. Per le classi avremo: `$('.tipo')`; infine per gli attributi: `$('[tempo]')`; o anche: `$('temperatura[meteo]')`;

Si noti che in tutti questi casi, la sintassi di selezione è identica alla sintassi utilizzata con CSS. Pertanto, per selezionare tutti gli elementi *prova* dal documento, digiteremo: `$('prova')`;

Pseudo Selettori

Proprio come con le pseudo classi CSS, possiamo anche utilizzare i pseudo selettori per selezionare gli elementi. Essi sono sempre preceduti da due punti. Di seguito sono riportati alcuni degli pseudo selettori più utili:

- `:not()` trova gli elementi che non corrispondono alla selezione.
- `:gt()` trova gli elementi con indice maggiore del numero fornito.
- `:even` trova elementi pari in una selezione.
- `:odd` trova elementi con numero dispari in una selezione.
- `:checked` trova i pulsanti di opzione o caselle di controllo selezionate.
- `:selected` trova le opzioni nelle caselle di selezione scelte.

- `:contains()` trova elementi che contengono una data porzione di testo.
- `:first` trova il primo elemento di un set.
- `:last` trova l'ultimo elemento di un set.
- `:focus` trova l'elemento che ha il focus (ad es. selezionato dal mouse).

Ad esempio con: `$('body prova:even');` selezioneremo tutti gli elementi *prova* di numero pari nell'elemento *body*, qui invece: `$('parte:first');` troveremo il primo elemento denominato *parte* nella pagina web. Ultimo esempio, dove cerchiamo tutti gli elementi *prova*, tranne il primo: `$("#prova:gt(0)");`

Un'altra caratteristica notevole degli pseudo selettori jQuery è che è anche possibile scriverne di propri. Per approfondire è meglio fare riferimento al sito ufficiale <https://api.jquery.com/> che è in inglese, ma possiamo sempre utilizzare la traduzione di Google.

Il metodo .find()

Abbiamo visto in precedenza come è possibile selezionare elementi figli di altri elementi. Per esempio, quanto segue trova tutti gli elementi *prova* che sono figli dell'elemento pagina: `$("#pagina prova:even");`

La selezione di elementi nel contesto di un sottoalbero specifico del DOM è molto comune. Per ad esempio, nel nostro sito potremmo voler selezionare sempre gli elementi nel contesto del tag principale per la pagina. Ciò garantirà che anche se la pagina dei tuoi prodotti sia incorporata in una pagina web più grande, selezionerà solo gli elementi rilevanti per essa.

Poiché la selezione di elementi nel contesto di altri elementi è così comune, jQuery ne fornisce due meccanismi aggiuntivi per raggiungerlo. Il primo meccanismo utilizza il metodo *find*:

```
$('#prodotti').find('prova');
```

Questo prima troverà l'elemento con l'ID *prodotti* e poi cercherà all'interno del risultato qualsiasi elemento *prova*. L'altro modo per ottenere esattamente lo stesso risultato è utilizzare il secondo parametro opzionale per jQuery dopo la stringa di selezione:

```
$('.prova', '#prodotti')
```

Il secondo parametro fornisce il contesto all'interno del quale dovrebbe avvenire la selezione.

Ritorniamo un attimo su: `$('#prodotti').find('prova');` vediamo che jQuery non sta semplicemente restituendo un oggetto DOM perché essi non supportano un metodo di ricerca, in questo caso particolare, jQuery ha restituito il proprio tipo di oggetto avvolto sugli oggetti DOM, ed è l'oggetto jQuery su cui viene eseguito `find`.

In effetti, ogni selezione jQuery restituisce un oggetto specifico di jQuery, non un oggetto DOM. Tuttavia, l'oggetto restituito è in grado di mascherarsi come un array e di accedere a specifici indici restituendo gli oggetti DOM nativi. vediamo un esempio dove assegneremo un oggetto DOM nativo alla variabile *domOgg*:

```
let domOgg = $('prova')[0];
```

È sempre possibile convertire un oggetto DOM nativo in un oggetto jQuery incorporandolo nel file struttura di selezione:

```
$(domOgg);
```

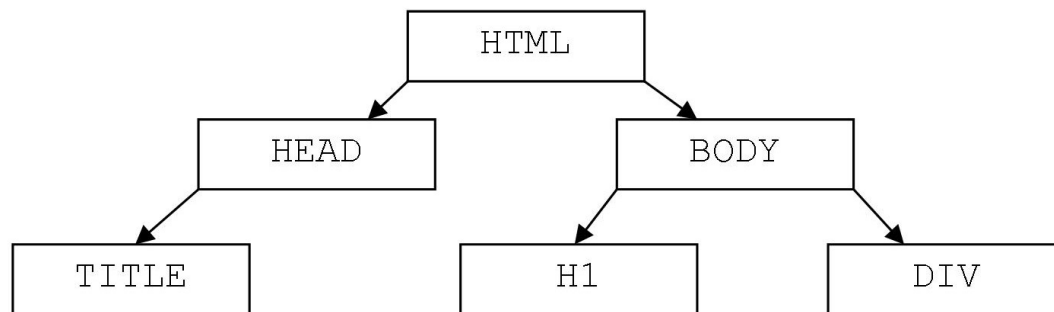
Questo ti dà quindi accesso a tutte le funzionalità aggiuntive fornite da jQuery. Ad esempio, questa riga restituisce il testo dell'elemento: `$(domOgg).text()`, mentre questa chiamata chiede se il testo dell'elemento contiene la stringa *"telefono"*:

```
$(domOgg).is(':contains("telefono)");
```

In genere lavoreremo con oggetti jQuery piuttosto che con oggetti DOM nativi, ma possiamo sempre usare questa tecnica se avremo bisogno di convertire oggetti DOM nativi in oggetti jQuery.

Attraversamento e manipolazione

Con il metodo *find* e con *domObject* abbiamo imparato come selezionare elementi dal DOM con jQuery. Adesso vedremo come passare da quegli elementi a un altro insieme di elementi correlati, manipolare i nodi nel DOM: ciò include l'aggiunta di nuovi nodi, la modifica nodi esistenti e rimozione di nodi dal DOM.



La struttura ad albero di esempio raffigura un ipotetico DOM che abbiamo visto all’inizio del libro, ci torna utile adesso per comprendere “l’attraversamento” o la “traslazione”, cioè trovare o selezionare parti del codice html in base alla relazione con altri elementi. Nello specifico la figura appena vista mostra l’elemento madre *html*, *head* e *body* saranno i figli, ma anche fratelli, *title*, *contenuto* e *div* sono sia figli sia discendenti di *html*, e così via.

```
<html>
  <head>
    <title>Titolo del Documento</title>
  </head>
  <body>
    <div id="contenuto">
      <p>Questo è un paragrafo.</p>
    </div>
  </body>
</html>
```

Quando si esegue una selezione jQuery, il risultato è un oggetto jQuery che incapsula un insieme di elementi. Le operazioni di attraversamento consentono di passare dagli elementi inizialmente selezionati a un nuovo insieme di elementi. Il risultato di un attraversamento è quindi anche un oggetto jQuery che incapsula un insieme di elementi.

Abbiamo già visto un'istanza di un'operazione di attraversamento: il metodo *find* era un'operazione trasversale perché iniziava trovando un elemento (o un insieme di essi), per poi trovare altri elementi che sono figli di questi. È anche possibile passare dagli elementi ai loro genitori. Ad esempio, quanto

segue seleziona tutti gli elementi *text* e quindi trova i loro genitori, che sono elementi *title*:

```
$('.text').parent();
```

La funzione *parent* restituisce i genitori immediati; se vogliamo trovare elementi che sono indiretti, potremo usare la funzione *parents*. Questa restituisce qualsiasi elemento che è un antenato di elementi selezionati, ma è possibile fornire una selezione a questa funzione come parametro. Per esempio, potremmo voler restituire il modulo che è il padre di tutti i campi di *input*. Vediamo come fare:

```
$('.input').parents('form');
```

Questo restituisce solo risultati univoci; pertanto da questa selezione viene restituito un singolo elemento del modulo. È anche possibile creare un set di risultati che contenga gli elementi originali insieme ai nuovi elementi selezionati. Ad esempio, quanto segue seleziona tutti i campi di input e l'elemento del modulo in una lista di selezione singola utilizzando la funzione *andSelf*:

```
$('.input').parents('form').andSelf();
```

Un'altra traversata comune è selezionare elementi che sono fratelli di altri elementi (elementi sono fratelli se hanno lo stesso genitore). Ad esempio, potresti voler selezionare tutte le etichette che sono fratelli dei campi di input, ma solo se contengono l'attributo richiesto:

```
$('.input[attributo]').siblings('label');
```

Con la maggior parte delle funzioni di attraversamento, puoi scegliere di aggiungere un filtro di selezione ("etichetta" in questo caso) o omettere il parametro per ricevere tutti i fratelli. Esistono numerose altre funzioni di attraversamento che trovano fratelli specifici:

- *next*: trova il fratello successivo che soddisfa criteri specifici.
- *prev*: trova il fratello precedente che soddisfa criteri specifici.
- *last*: trova l'ultimo fratello che soddisfa criteri specifici.
- *first*: trova il primo fratello che soddisfa criteri specifici.

Ci sono un paio di altre importanti funzioni di attraversamento che dovremo conoscere prima di andare avanti:

- *add*: fornisce un meccanismo per unire due diverse selezioni. Ad esempio, potremo utilizzarlo per creare un elenco di selezione di tutti i campi di input e di tutte le etichette:

```
$('.input').add('label');
```

- *closest*: la funzione più vicina trova l'antenato più vicino di un elemento, che soddisfa criteri specifici, ma, a differenza del selettore dei genitori, considera l'elemento originario. Immagina di voler selezionare l'elemento *mario* più vicino a qualsiasi elemento che contenga il testo “*Nino*”:

```
$('.contains("Nino").closest('mario');
```

Se l'elemento contenente il testo è un elemento *mario*, viene aggiunto al set di risultati. Altrimenti, jQuery salirà attraverso i genitori dell'elemento, cercando il primo elemento *mario*.

- *eq*: questo operatore può essere utilizzato per restituire l'elemento a un indice specifico, ad esempio quanto segue restituisce la seconda sezione del documento (si parte da zero):

```
$('.sezione').eq(1).
```

Catene

Le funzioni di attraversamento sono uno dei grandi punti di forza di jQuery, vengono eseguite su una selezione di elementi jQuery restituendo una selezione di elementi jQuery, è così possibile concatenare un intero insieme di funzioni di attraversamento in una singola istruzione, vediamo:

```
$('.time').siblings('.overlay').parents('tr').last();
```

Questo codice esegue quanto segue:

- Seleziona tutti gli elementi temporali.
- Seleziona qualsiasi fratello di questi elementi che ha la classe *overlay*.
- Seleziona l'elemento *tr* che è padre di questi elementi.
- Seleziona l'ultimo elemento restituito dall'elenco.

In effetti, questo codice che abbiamo appena visto seleziona l'ultima riga che ha un elemento *time* con una sovrapposizione. Questo concatenamento potrebbe continuare quasi indefinitamente.

Gestire il DOM

Manipolare il DOM ci aiuterà a fornire alla nostra pagina web un comportamento dinamico all'utente. Consideriamo l'esempio precedente che trovava tutte le etichette dei campi di input:

```
$('.input[richiesto]').siblings('label');
```

Potremo decidere di voler modificare il testo di etichette come questa per visualizzarlo in blu:

```
$('.input[richiesto]').siblings('label').css('color', 'blue');
```

L'esecuzione di questa singola riga di codice è sufficiente per colorare di blu tutte le etichette. Aggiungeremo anche un asterisco blu accanto a ciascuna etichetta. L'obiettivo è creare la seguente struttura:

```
<label for="Nome Contatto" style="color: blue;">
Nome del contatto<span class="Prova">*</span>
</label>
```

Inizieremo aggiungendo una classe a al foglio di stile CSS (ad esempio *Contacts.css*) in modo che corrisponda a quella richiesta:

```
.Prova {color: blue; padding-left:7px;}
```

Il primo compito è creare un nuovo elemento da aggiungere a ciascuna etichetta:

```
$('<span>').text('*').addClass('Prova');
```

Questo esegue quanto segue:

- Crea un nuovo nodo *span* che può essere aggiunto al DOM.
- Aggiunge un asterisco come testo.
- Aggiunge la classe *Prova* a *span*.
- Restituisce il nuovo elemento *span* come risultato.

Successivamente, selezioneremo tutte le etichette a cui desidereremo aggiungere l'intervallo e utilizzeremo la funzione di aggiunta per aggiungere

questa singola istruzione in tutte le etichette:

```
$('#:input[richiesto]').siblings('label').append($('>').text('*')  
)<div data-bbox="121 182 881 224" data-label="Text">

Si noti come questa complessa operazione possa essere espressa in una singola affermazione.


```

Una volta che questa riga di codice è stata testata nella Console, potremo aggiungerla al metodo *init* in un file esterno **.js* per assicurarci che venga sempre eseguito quando la pagina viene caricata.

Naturalmente, avremmo potuto aggiungere manualmente gli asterischi alle etichette, a parte la perdita di tempo ci sono dei vantaggi per fare come indicato, ad esempio quando indicheremo un nuovo campo come obbligatorio, non sarà necessario ricordarsi di aggiungere un asterisco all'etichetta.

Quando si aggiungono nuovi elementi in relazione a un elemento esistente, ci sono quattro posizioni per le quali si potrebbe voler inserire nuovi nodi:

- Potremmo voler inserire un nuovo elemento come fratello dell'elemento *prova*, ma prima di esso nel file DOM. Questo può essere ottenuto con la funzione *before*:

```
$('#prova').before('<span>eccolo</span>')
```

- Con la funzione *after* faremo lo stesso con l'elemento *prova*, ma dopo nel DOM.

- La funzione *anteponi* ci servirà per ottenere un primo figlio dell'elemento *prova*.

- Mentre la funzione *add* ci servirà per ottenere un ultimo figlio dell'elemento *prova*.

A ciascuna di queste funzioni può essere passata una stringa di markup HTML o un oggetto DOM.

Oltre all'aggiunta di nuovi nodi al DOM, per rimuovere nodi dal DOM ci serviremo della funzione *remove*:

```
$('#.prova').remove();
```

Questa funzione restituisce tutti gli elementi che sono stati rimossi.

Modificare elementi del DOM

Le tecniche di manipolazione che abbiamo esaminato finora sono progettate per aggiungere o rimuovere nodi da cui il DOM. jQuery fornisce anche la possibilità di modificare elementi esistenti.

Ad esempio, potremo manipolare direttamente il testo di un elemento in questo modo:

```
$('#dettagliContatto p3').text('DETTAGLI CONTATTO');
```

oppure potremo modificare il suo HTML come segue:

```
$('#dettContatto p3').html('<span>Dettagli contatto </span>');
```

Questa riga di codice posiziona il testo all'interno di un elemento *span*. jQuery è una libreria molto flessibile, quindi generalmente ci sono modi diversi per svolgere lo stesso compito. Ad esempio, anche il codice seguente aggiunge un elemento *span* attorno al testo dell'elemento *p3*:

```
$('#dettagliContatto p3').wrapInner('<span>');
```

È anche possibile impostare il valore degli input del modulo utilizzando la funzione *val*:

```
$('[nome="nomeContatto"]').val('proviamo abc');
```

Ognuna di queste funzioni può essere utilizzata senza un argomento per accedere al valore corrente. La prossima visualizza il valore corrente del campo *nomeContatto*:

```
$('[nome="nomeContatto"]').val();
```

Se utilizzato in questa modalità, verrà restituito un solo valore, quindi se si richiamano queste funzioni su un set di elementi, verrà restituito solo il valore del primo. Inoltre, dato che queste funzioni non restituiscono oggetti jQuery, non è possibile concatenare altre funzioni jQuery sui loro risultati.

In precedenza in questa lezione, hai visto come è possibile impostare singole proprietà CSS utilizzando la funzione *CSS*, e come le classi possono essere aggiunte a un elemento con *addClass*. Di conseguenza è anche possibile rimuoverle con *removeClass*. Ad esempio, questo rimuoverà la classe che hai aggiunto a tutti gli elementi *span* che contengono *mario*:


```
$('#label span').removeClass('mario');
```

Un'ulteriore funzione utile è *toggleClass*. Questo aggiunge una classe a un elemento se non lo fa ce l'ha già e se lo fa lo rimuove.

Infine, è possibile accedere e modificare gli attributi di un elemento. Ad esempio, il codice seguente restituisce la lunghezza massima dell'area di testo:

```
$('#textarea').attr('maxlength');
```

mentre quanto segue modifica il valore dell'attributo (o aggiunge l'attributo nel caso esista già):

```
$('#textarea').attr('maxlength', 185);
```

l'attributo ovviamente potrà anche essere rimosso utilizzando *.removeAttr*.

Iterazioni

In Javascript, ma non solo, i loop sono uno strumento imprescindibile, e solitamente una volta selezionato un array, scorreremo ogni elemento per eseguire un'operazione su di esso. Poiché il risultato di una selezione jQuery imita un array, è possibile utilizzarlo in un ciclo *for* per iterarlo.

Tuttavia potremo avvalerci di un approccio più semplice del loop, il quale consiste nell'utilizzare una funzione di supporto jQuery chiamata *each*. Essa esegue una iterazione esplicita sugli elementi, vediamo un esempio:

```
$("#prova").each(function() {  
  let $item = $(this);  
  if($item.is("#prodotti")) {  
    $item.addClass("prodotti-menu-sottomenu-categoria");  
    return false;  
  }  
});
```

Abbiamo effettuato un controllo, dopodiché se passiamo oltre, riusciremo ad aggiungere una classe alla voce selezionata. Il loop viene interrotto dopo che la condizione si è verificata, in tal caso la funzione riporterà *false* e vi sarà l'uscita dal ciclo *each*.

Eventi jQuery

jQuery viene in genere utilizzato per manipolare il DOM dopo che la pagina è stata caricata, ma per avviare questa procedura è necessario un evento: del mouse, ad esempio un clic su un pulsante, della tastiera, ad esempio l'utente che digita in un campo di input, del modulo, come il valore di un elemento selezionato che cambia, o, un evento sullo schermo, ad esempio la finestra che viene ridimensionata.

Proprio come è possibile ascoltare eventi come questi utilizzando l'API DOM nativa, è possibile ascoltare questi eventi con jQuery.

Il metodo più semplice di jQuery è *.ready()* e si attiva appena il documento è caricato e pronto all'uso:

```
$(document).ready(function(){
```

Registrazione ascoltatori di eventi

La registrazione degli ascoltatori di eventi inizia con la selezione dell'elemento che genererà l'evento. Una volta selezionato, il metodo appropriato viene invocato per registrare un ascoltatore di eventi e viene passato a una funzione di callback che dovrebbe essere invocata quando si verifica l'evento. Qualsiasi evento Javascript standard o personalizzato può essere associato all'elemento jQuery attualmente selezionato. Per comodità potremmo utilizzare *.click()*, ma *.on()* offre più opzioni.

```
$("#prova").on("click", function() {  
    alert($(this).text());  
});
```

Quando cliccheremo su prova verrà visualizzato un testo.

```
$("#prova").on("submit", function(event) {  
    event.preventDefault();  
});
```

In questo esempio con *event.preventDefault()* annulleremo solamente un'azione invece che tutto l'invio del modulo *prova*.

Alla fine del capitolo 6 abbiamo visto il metodo *bind* e forse gli esempi ce lo ricordano un po', vi rinfreschiamo la memoria:

```
$('#prova').bind('click', function(event) {  
    event.preventDefault();  
    console.log('cliccato!');  
});
```

```
$('#prova').on('click', function(event) {
    event.preventDefault();
    console.log('cliccato!');
});
```

Una volta avviato on forse avremo bisogno anche di disattivarlo, infatti in questi casi utilizzeremo *off()*: `$('#prova').off('click');`

Potremmo aver bisogno di eseguire una volta sola un evento, in tal caso abbiamo a disposizione *one()*:

```
$("#prova").one("click", function() {
    alert( "Questo sarà visualizzato una volta solamente" );
});
```

Ascoltatori di eventi delegati

Con un evento delegato, selezioni un elemento che sai essere nel DOM quando la pagina viene caricata (come l'elemento body) e ci associ un listener di eventi a uno qualsiasi dei suoi discendenti. Il bello è che i discendenti non devono esistere quando l'ascoltatore di eventi è registrato; eventuali discendenti appena aggiunti verranno automaticamente associati al listener di eventi pertinente.

```
<html>
<body>
<div id="prova">
  <y id="list">
    <qui><x href="http://sito.com">Item #1</x></qui>
    <qui><x href="http://altrosito.com">Item #2</x></qui>
    <qui><x href="/local/dir/abc">Item #3</x></qui>
  </y>
</div>
</body>
</html>
```

Capire e visualizzare come gli eventi si propagano è essenziale per poter sfruttare la delega di eventi. Ogni volta che si fa clic su uno dei nostri tag di ancoraggio, viene attivato un evento clic per quell'ancora che si espande nell'albero DOM, attivando ciascuno dei gestori dell'evento clic padre:

```
<x>
<qui>
<y #list>
<div #prova>
<body>
<html>
radice del documento
```

Conoscendo la propagazione degli eventi adesso possiamo eseguire un evento delegato:

```
$("#list").on("click", "x", function(event) {  
    event.preventDefault();  
    console.log($(this).text());  
});
```

In questo caso, se il tag di ancoraggio (<x>) corrisponde, verrà eseguita la funzione anonima. È stato collegato un ascoltatore di eventi per un singolo clic che gestirà gli eventi generati dai discendenti, invece di collegare un numero sconosciuto di eventi direttamente ai tag di ancoraggio esistenti. Concludendo, Il metodo *on* accetta i seguenti parametri:

- Un elenco separato da spazi di eventi da ascoltare.
- Un selettore per trovare i discendenti che genereranno gli eventi.
- La funzione da eseguire quando si verifica l'evento.

Gli altri eventi più comuni basati sul mouse che possono essere ascoltati sono:

- *dblclick*: si attiva se si fa clic su un elemento due volte rapidamente.
- *mousedown*: si attiva quando l'utente preme il pulsante del mouse.
- *mouseup*: si attiva quando l'utente rilascia il pulsante del mouse.
- *mousemove*: si attiva ogni volta che si muove il mouse.

Eventi tastiera

Le sezioni precedenti si sono concentrate sugli eventi del mouse, ma i nostri computer sono dotati di tastiere, e la pressione dei tasti genera eventi gestiti dai metodi *keypress* o *keydown*:

```
$("#target").keypress(function() {  
    console.log("Abbiamo chiamato .keypress()");  
});
```

```
$("#target").keydown(function() {  
    alert("Abbiamo chiamato .keydown()");  
});
```

Entrambi i metodi sono simili, ma *keydown* viene attivato anche dalla pressione di shift, ctrl, alt, mentre *keypress* no.

Vediamo gli altri eventi legati all'uso della tastiera:

- *change*: questo evento viene chiamato ogni volta che cambia il valore in un campo modulo. Questo può essere applicato a qualsiasi campo di input del modulo, ma nel caso di campi di input basati su testo, l'evento si attiva solo una volta che il l'utente esce dal campo.
- *focus*: questo evento viene richiamato quando un campo di input riceve il l'attenzione dell'utente, ad esempio cliccandoci con il mouse.
- *blur*: esattamente il contrario di focus.

Eventi sullo schermo

L'ultima grande categoria di eventi è costituita dagli eventi sullo schermo. L'evento schermo più utile è *ready*. Gli esempi JavaScript finora hanno posizionato JavaScript alla fine della pagina web per assicurarsi che il DOM è stato caricato prima che inizi la selezione dell'elemento. L'evento *ready* fornisce un modo più sicuro per garantire che il DOM sia stato caricato completamente prima di tentare per manipolarlo. È possibile registrare un ascoltatore di eventi pronto allegando il documento del browser oggetto in un selettore jQuery e invocando il metodo *ready* su di esso. Ad esempio:

```
$(document).ready(function(evt) {  
  let elementoProva = document.getElementById('prova');  
  let schermo = paginaProva(elementoProva);  
  screen.init();  
});
```

L'altro evento principale basato sul browser è l'evento di ridimensionamento. Questo si attiva ogni volta che l'utente ridimensiona la finestra del browser:

```
$(window).resize(function(evt) {
```

Osserviamo quanto può essere semplice cambiare l'ascoltatore di eventi per nascondere e mostrare le note popup come segue:

```
if (evt.type === "mouseenter") {  
  $(evt.target).siblings('.overlay').slideDown();  
} else {  
  $(evt.target).siblings('.overlay').slideUp();  
}
```

Se ricaricheremo la pagina web, passando il mouse su un elemento temporale, noteremo che il popup sarà visualizzato come se venisse trascinato verso il basso, come lo schermo di un proiettore. Allo stesso modo, quando è nascosto, sarà come se lo schermo del proiettore fosse stato rilasciato di nuovo. È possibile controllare quanto impiega l'intero effetto fornendo un tempo in millisecondi come il primo parametro a queste funzioni: Il valore predefinito è 400 millisecondi (0,4 di secondo), è anche possibile controllare molti altri aspetti del processo di animazione. Ovviamente per problemi di spazio non riusciremo a parlarne ma potremo facilmente saperne di più dal sito Web di jQuery, difatti questa libreria supporta molti altri effetti, ad esempio potremo effettuare una chiamata asincrona personalizzata con il metodo: `$.ajax()`. Quindi jQuery e AJAX sono perfettamente interfacciabili tra di loro. Una volta padroneggiate queste librerie saremo in grado di fare grandi cose. Diamo una veloce occhiata ad AJAX, giusto per stuzzicare l'appetito.

AJAX

Anche con AJAX possiamo lavorare e migliorare le nostre pagine Web. È l'acronimo di *Asynchronous Javascript And Xml*. Essenzialmente con AJAX possiamo apportare dei cambiamenti alla pagina senza che vi sia il refresh della stessa. Questo tipo di tecnologia è utilizzata, tra gli altri, da uno dei siti più famosi e frequentati in assoluto, cioè *Google Maps*.

Semplificando al massimo, AJAX non è un linguaggio a sé stante, bensì è una sorta di Frankenstein dove sono mixati insieme XML, XHTML, CSS, il DOM aggiornato dinamicamente e *XMLHttpRequest* che ci consente di dialogare con il server in maniera asincrona, quindi senza attendere la risposta del server. Nel nostro caso alla base c'è sempre Javascript che gestisce il tutto.

```
let prova = new XMLHttpRequest();
prova.onreadystatechange = function() {
    if (prova.readyState == 4) {
        if (prova.status == 200) {
            alert('Sta funzionando');
        } else {
            alert('Errore')
        }
    }
}
prova.open("GET", "prova.html", true);
prova.send();
```

In questo esempio ci sono diverse cose strane, come i codici numerici sottoposti alla condizione `if`. Brevemente segnaliamo che i *readyState* possono essere cinque: 0, richiesta non inizializzata, 1, connessione con il server stabilita, 2, richiesta ricevuta, 3, richiesta processata, 4, richiesta finita e risposta pronta. Gli *status* invece sono molti di più, possiamo vedere la lista su: https://it.wikipedia.org/wiki/Codici_di_stato_HTTP . Nello specifico, 200 è il codice standard per una richiesta HTML andata a buon fine. A parte questo il cuore di AJAX nel listato che abbiamo visto è *XMLHttpRequest*, il quale, quando evocato invia la richiesta al server Web, la risposta viene elaborata e rimandata indietro al programma il quale la eseguirà. Questa operazione si chiama “*remote scripting*” e ci permetterà di effettuare anche decine di operazioni asincrone nello stesso istante, senza che vi siano particolari problemi.

I metodi AJAX

Questi metodi ci serviranno per delle operazioni molto utili, potremo usarli con jQuery, ma non necessariamente.

Dovremo innanzitutto servirci del metodo *open()* con questa sintassi, come abbiamo visto nel primo esempio:

```
ajax.open(metodo, url, true);
```

```
ajax.open(get, 'desktop/esempio.html', true);
```

Potremo anche usare AJAX con jQuery, ad esempio con il metodo *load()* riusciremo a caricare dati dal server per inserirli all'interno del nostro elemento selezionato, come `id=prova`:

```
[javascript]$('#prova').load('esempio.html');[/javascript]
```

get() e *post()* invece ci serviranno per scambiare informazioni con il server. Con il primo otterremo delle informazioni, anche se dovremo fare attenzione perché non saranno criptate, con il secondo invieremo delle informazioni in maniera più privata, dato che i dati saranno celati all'utente.

I parametri AJAX

I parametri sono fondamentali per scambiare dati correttamente, il principale è *readyState*, già descritta, poi abbiamo *onreadystatechange*, il quale eseguirà una funzione automaticamente ogni volta che *readyState* cambia.

Lo *status* esegue un ulteriore controllo, il codice 200 ci darà l'ok. *responseText* e *responseXML* acquisiranno la risposta del server rispettivamente come stringa o come oggetto DOM XML, infine, con *statusText* otterremo la descrizione e il significato degli stati restituiti dal server.

Ora vedremo un esempio molto interessante dove dovremo compilare un piccolo form e il codice nel frattempo in maniera asincrona aggiornerà continuamente il campo dei suggerimenti recuperati da un file php:

```
<html>
<body>
<h2>Il nostro esempio</h2>
<h3>Inserisci un nome nel form e leggi il suggerimento:</h3>

<p>Suggerimento: <span id="aiuto"></span></p>
<p>Nome:          <input                type="text"                id="txt1"
onkeyup="mostrAiuto(this.value)"></p>

<script>
function mostrAiuto(str) {
    if (str.length == 0) {
        document.getElementById("aiuto").innerHTML = "";
        return;
    }
    const xhttp = new XMLHttpRequest();
    xhttp.onload = function() {
        document.getElementById("aiuto").innerHTML =
        this.responseText;
    }
    xhttp.open("GET", "suggerimenti.php?q="+str);
    xhttp.send();
}
</script>
</body>
</html>
```

I suggerimenti saranno all'interno di un documento php *suggerimenti.php*:

```
<?php
// Un vettore o array con dei nomi
$a[] = "Anna";
$a[] = "Bice";
$a[] = "Camelia";
$a[] = "Diana";
$a[] = "Eva";
$a[] = "Fiona";
```



```

$a[] = "Guenda";
$a[] = "Heide";
$a[] = "Ines";
$a[] = "Johanna";
$a[] = "Kitty";
$a[] = "Linda";
$a[] = "Nina";
$a[] = "Olga";
$a[] = "Petunia";
$a[] = "Amanda";
$a[] = "Rachele";
$a[] = "Camilla";
$a[] = "Dora";
$a[] = "Eleonora";
$a[] = "Evita";
$a[] = "Sara";
$a[] = "Tina";
$a[] = "Ursula";
$a[] = "Violetta";
$a[] = "Lisa";
$a[] = "Elizabeth";
$a[] = "Ellen";
$a[] = "Wenche";
$a[] = "Vicky";

```

```

// Acquisiamo il carattere digitato
$q = $_REQUEST["q"];
$hint = "";

```

```

// Confrontiamo i nomi nell'array e se $q è diverso da ""
if ($q !== "") {
    $q = strtolower($q);
    $len=strlen($q);
    foreach($a as $name) {
        if (stristr($q, substr($name, 0, $len))) {
            if ($hint === "") {
                $hint = $name;
            } else {
                $hint .= ", $name";
            }
        }
    }
}

```

```

// Se le condizioni non sono soddisfatte non ci saranno suggerimenti
echo $hint === "" ? "nessun suggerimento" : $hint;
?>

```

Nell'esempio sopra, quando un utente digita un carattere nel campo di input, viene eseguita la funzione *mostrAiuto()* la quale è attivata dall'evento *onkeyup*. I suggerimenti saranno nel file php il quale sarà costantemente utilizzato per cercare il suggerimento da mostrare. Molto semplice ma d'effetto.

Anche in questo caso lo spazio è tiranno ma quel poco utilizzato ci è sicuramente servito a capire che AJAX è fondamentale in alcuni casi, come quello relativo allo scambio di dati con i server. Abbiamo anche compreso che Javascript (involontariamente ma con grande piacere) si avvale di strumenti eccezionali utili per tantissimi ambiti, dobbiamo solo conoscerli e sfruttarli.

Three.js

Three.js è una libreria estremamente potente perché ci permetterà, utilizzando Javascript, di realizzare qualsiasi progetto attraverso la grafica tridimensionale fotorealistica, che girerà semplicemente su qualunque computer e smartphone, nel nostro browser. Il suo utilizzo è relativamente facile, richiede ovviamente una conoscenza di base di Javascript e, altrettanto ovviamente ha una sua sintassi che dovremo imparare. Nello specifico avremo bisogno del solito browser, di un editor di testo come quelli elencati all'inizio del libro, un Web server per permettere il funzionamento delle nostre app, e di un'API Javascript, come WebGL che ci permetterà di disegnare grafica 3D sulle pagine Web (anche 2D).

Per includere questa libreria nei nostri progetti dovremo scaricare il modulo del core di three.js sul nostro pc. Sul sito ufficiale threejs.org esiste un'ampia documentazione gratuita e open source con esempi pratici, oltre ad una vasta comunità che ci aiuterà nei nostri lavori.

Le applicazioni tridimensionali create con questa libreria sono strutturate all'incirca come i normali listati Javascript. Hanno bisogno di una *Scena*, dove il nostro mondo tridimensionale prenderà vita, di una parte dedicata al codice chiamata *Renderer* e, infine di una area dedicata alle *Camere*, che dovremo posizionare al meglio per visualizzare correttamente la scena. Il listato lavorerà in maniera asincrona tra questi tre oggetti.

Ora vedremo un esempio semplificato di come creare un cubo tridimensionale in modo da comprendere la facilità e la potenzialità di questa libreria.

Iniziamo creando la scena dove il mondo 3D prenderà vita utilizzando il nostro amico costruttore *new*:

```
const scena = new Scene();
scena.background = new Color('black'); // il colore di sfondo
```

Ora imposteremo dei parametri che serviranno per regolare la camera che visualizzerà la scena:

```
const fov = 36; // La profondità di campo
const aspect = container.clientWidth / container.clientHeight;
const near = 0.2;
const far = 110;
const camera = new PerspectiveCamera(fov, aspect, near, far);
camera.position.set(0, 0, 11);
```

Adesso con poche righe di codice andremo a creare un cubo 3D:

```
const geometry = new BoxBufferGeometry(2, 2, 2);
const material = new MeshBasicMaterial();
const cube = new Mesh(geometry, material);
scena.add(cube);
```

Queste istruzioni lo posizioneranno sulla nostra scena:

```
const renderer = new WebGLRenderer();
renderer.setSize(container.clientWidth, container.clientHeight);
renderer.setPixelRatio(window.devicePixelRatio);
container.append(renderer.domElement);
renderer.render(scena, camera);
```

Lo script si avvalerà anche dell'HTML e del CSS in cui dovrà interfacciarsi per poter funzionare.

Proviamo un altro esempio più complesso che mostra come creare una scena 3D con un cubo rotante e una sfera rimbalzante:

```
// Inizializzazione di Three.js
const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera(75, window.innerWidth /
window.innerHeight, 0.1, 1000);
const renderer = new THREE.WebGLRenderer({ antialias: true });

// Impostazione delle dimensioni del renderer e aggiunta al DOM
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

// Creazione di un cubo colorato
const cubeGeometry = new THREE.BoxGeometry();
```

```

const cubeMaterial = new THREE.MeshBasicMaterial({ color: 0x00ff00
});
const cube = new THREE.Mesh(cubeGeometry, cubeMaterial);
scene.add(cube);

// Creazione di una sfera rimbalzante
const sphereGeometry = new THREE.SphereGeometry(1, 32, 32);
const sphereMaterial = new THREE.MeshBasicMaterial({ color: 0xff0000
});
const sphere = new THREE.Mesh(sphereGeometry, sphereMaterial);
sphere.position.set(0, 0, -5);
scene.add(sphere);

// Impostazione della posizione della telecamera
camera.position.z = 5;

// Animazione della scena
const animate = () => {
  requestAnimationFrame(animate);

  // Rotazione del cubo
  cube.rotation.x += 0.01;
  cube.rotation.y += 0.01;

  // Rimbalzo della sfera
  sphere.position.y = Math.sin(Date.now() * 0.002) * 2;

  // Rendering della scena
  renderer.render(scene, camera);
};

// Avvio dell'animazione
animate();

```

Questo codice crea una scena 3D con un cubo rotante di colore verde e una sfera rimbalzante di colore rosso. La telecamera è posizionata in modo da guardare la scena e l'animazione viene gestita dalla funzione *animate*. L'effetto rimbalzante della sfera è ottenuto modificando la sua posizione in base al tempo.

Ecco un altro esempio che utilizza Three.js per creare un effetto di neve cadente in uno scenario 3D:

```

// Importa la libreria Three.js
import * as THREE from 'three';

// Inizializza la scena, la camera e il renderer
const scene = new THREE.Scene();

```

```

const camera = new THREE.PerspectiveCamera(75, window.innerWidth /
window.innerHeight, 0.1, 1000);
const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

// Crea una geometria per i fiocchi di neve
const snowflakeGeometry = new THREE.BoxGeometry(0.05, 0.05, 0.05);

// Crea un materiale bianco per i fiocchi di neve
const snowflakeMaterial = new THREE.MeshBasicMaterial({ color:
0xffffffff });

// Genera una serie di fiocchi di neve
const snowflakes = [];
for (let i = 0; i < 100; i++) {
    const snowflake = new THREE.Mesh(snowflakeGeometry,
snowflakeMaterial);
    snowflake.position.set(
        Math.random() * 10 - 5,
        Math.random() * 10 - 5,
        Math.random() * 10 - 5
    );
    snowflakes.push(snowflake);
    scene.add(snowflake);
}

// Posiziona la camera
camera.position.z = 5;

// Crea una funzione di animazione
const animate = () => {
    requestAnimationFrame(animate);

    // Ruota i fiocchi di neve
    snowflakes.forEach((snowflake) => {
        snowflake.rotation.x += 0.005;
        snowflake.rotation.y += 0.005;
    });

    // Rendi visibile la scena con la camera
    renderer.render(scene, camera);
};

// Avvia l'animazione
animate();

```

Questo esempio utilizza Three.js per creare una scena 3D in cui dei fiocchi di neve cadono casualmente. La scena contiene una serie di cubi bianchi che

rappresentano i fiocchi di neve. La funzione `animate` viene chiamata in modo ricorsivo per animare i fiocchi di neve ruotandoli continuamente.

Per chi ha una discreta base con Javascript è evidente come possa aprirsi un mondo di possibilità che dovremmo tenere in considerazione, ma anche ignorare del tutto. In ogni caso era mio dovere illustrare questa evenienza, e, se vi dovesse attrarre, a questo punto, come già detto, su threejs.org avrete gli strumenti utili per proseguire, tutti assolutamente gratuiti e, per chi ha problemi con l'inglese, è inutile sottolineare che oramai Google translate funziona piuttosto bene. In alternativa dovrete attendere che scriva un nuovo libro sull'argomento.

Quiz & esercizi

- 1) Creare un listato che utilizzando jQuery possa gestire i seguenti eventi:
 - al click del mouse su di un pulsante cambia il colore di sfondo
 - il doppio click sposta lo stesso pulsante
 - il passaggio del mouse sopra il pulsante lo fa ingrandire
- 2) Creare un listato che utilizzando jQuery possa realizzare le seguenti funzioni su tre bottoni:
 - al click sul primo bottone cambia il testo sui pulsanti
 - il click sul secondo cambia colore del titolo della pagina
 - il passaggio del mouse sopra terzo pulsante il primo cambia colore
- 3) Creare un listato che mostri o nasconda un testo su una pagina web in base al passaggio del mouse.
- 4) Attraverso i pseudo selettori potremo manipolare il DOM o selezionare elementi di un vettore?
- 5) Il selettore è obbligatorio o esistono delle scorciatoie per utilizzare jQuery?
- 6) Per rimuovere una classe è meglio utilizzare `.removeClass()` oppure `toggleClass`?
- 7) Scrivi un codice JavaScript che effettua una richiesta GET a una URL e gestisce la risposta.
- 8) Scrivi un codice JavaScript che effettua una richiesta POST a una URL con dati JSON e gestisce la risposta.
- 9) Modifica un tuo listato in modo da gestire gli errori durante le richieste AJAX.
- 10) Riscrivi un tuo listato per eseguire una richiesta AJAX utilizzando *Promises*.

- 11) Inizializza una scena Three.js con una telecamera prospettica e un renderer.
- 12) Aggiungi un cubo alla scena Three.js con un materiale e una posizione.
- 13) Crea una funzione di animazione per ruotare il cubo.
- 14) Carica e visualizza un modello 3D OBJ utilizzando l'*OBJLoader* di Three.js.

Riassunto

Siamo giunti quasi alla fine del libro e qui abbiamo esplorato alcuni argomenti avanzati, come Ajax e Three.js, i quali ci fanno capire l'enorme potenzialità inespresse di Javascript.

Per le cose concrete c'è sempre jQuery e i suoi vantaggi relativi al suo utilizzo in determinate situazioni, come le animazioni grafiche.

12 – *Le espressioni regolari*

Introduzione alle espressioni regolari

Esempi di espressioni regolari

Al lavoro con le espressioni regolari

Eccoci qui, siamo giunti (purtroppo per voi) all'argomento probabilmente più odiato e ostico dei linguaggi di programmazione. D'altronde, le espressioni regolari, abbreviate in *RegExp*, sono un potente strumento e una necessità dello sviluppo di qualunque software. Di contro la grammatica e la sintassi appaiono molto complesse, praticamente una lingua a parte che richiede parecchio tempo per essere padroneggiata. Tuttavia, con l'uso appropriato delle espressioni regolari, qualcosa che potrebbe richiedere una schermata di codice può essere riassunto in una singola affermazione. In realtà non è tutto qui, questa condensazione di una porzione del programma è solo la punta dell'iceberg. Ovviamente non ci metteremo ad elencare tutto ciò che saremo in grado di fare, ma dobbiamo anche renderci conto della potenza di questo strumento. Quindi, essenzialmente, le espressioni regolari ci serviranno anche per, ad esempio: trovare e modificare parti di una più ampia area di testo, velocizzare la creazione di codice votato all'analisi di stringhe, determinare se un elemento ha un nome di classe specifico, validare un nome, un'email o una password in un form, manipolare il DOM e l'HTML, e molto altro ancora.

Sintetizzando in maniera estrema, storicamente tutto nasce intorno agli anni 50, quando un matematico Statunitense di nome Stephen Kleene con i suoi studi ci regala i "linguaggi formali", che oggi ci permettono di comunicare con le macchine, nello specifico con i computer. Di fatto senza le espressioni regolari probabilmente la stessa internet avrebbe avuto enormi difficoltà ad esistere.

Ma cosa sono le *RegExp*? cercando sempre la sintesi potremmo affermare che si tratta di uno strumento atto al controllo sintattico di una data stringa per verificare che essa coincida ad un *pattern* (o modello) definito. Il pattern

sarebbe il motivo della ricerca generale, ed è composto da caratteri semplici, speciali o entrambi.

La parte facile è che possiamo utilizzare direttamente il costruttore *new RegExp()* o la sintassi letterale che prevede due delimitatori espressi con due barre laterali: `/ /` dove all'interno ci sarà il pattern.

Per la parte complicata procederemo con i prossimi paragrafi.

Esempi di espressioni regolari

In questo passaggio insisteremo sui vantaggi di utilizzo delle *RegExp*. Immaginiamo di dover scrivere un programma per controllare ciò che viene inserito in un campo di un form. Proviamo a vedere che differenza potrebbe esserci tra un semplice listato diciamo standard, e tra uno che più o meno fa la stessa cosa però utilizzando una *RegExp*.

Creiamo un codice che convalidi il formato dell'orario inserito in un campo d'input html, tralasciando l'ambito che dovrebbe essere all'interno di un form html, per ora concentriamoci sulle differenze tra i listati:

```
let time = document.form.time.value;
let timeParts = time.split(':');

if (timeParts.length !== 2) {
    alert('Orario non valido, inserisci nel formato HH:MM');
} else {
    let hours = parseInt(timeParts[0]);
    let minutes = parseInt(timeParts[1]);

    if (isNaN(hours) || isNaN(minutes) || hours < 0 || hours > 23 ||
minutes < 0 || minutes > 59) {
        alert('Orario non valido, inserisci nel formato HH:MM');
    } else {
        // Codice...
    }
}
```

Vediamo cosa succede qui: si ottiene il valore dell'orario dal campo di input utilizzando *document.form.time.value*, l'orario viene suddiviso in due parti separate, ore e minuti, utilizzando il metodo *split(':')*. Ad esempio, se l'utente inserisce "12:30", questo metodo creerà un array con due elementi: ["12", "30"].

Si verifica se l'array ottenuto ha esattamente due elementi. Se ciò non accade, mostra un messaggio di errore che indica che l'orario non è nel formato corretto "HH:MM". Se l'array ha due elementi, converte le ore e i minuti in numeri interi utilizzando *parseInt()*. Poi esegue una serie di

controlli per verificare che le ore siano comprese tra 0 e 23 e che i minuti siano compresi tra 0 e 59.

Se tutti i controlli hanno esito positivo, il codice successivo viene eseguito. Altrimenti, viene visualizzato un messaggio di errore se l'orario inserito è nel formato corretto ma non rispetta le regole delle ore e dei minuti.

Bisogna ammettere che non è proprio il miglior esempio di come si può utilizzare Javascript. Proviamo a confrontarlo con un'altra porzione di listato che fa lo stesso lavoro:

```
let time = document.form.time.value;
let timePattern = /^(0[0-9]|1[0-9]|2[0-3]):[0-5][0-9]$/;

if (!timePattern.test(time)) {
    alert('Orario non valido, inserisci nel formato HH:MM');
} else {
    // Codice...
}
```

Questo sembra decisamente meglio! Il listato utilizza un'espressione regolare per convalidare il formato dell'orario in modo più conciso. Ecco come funziona:

Ottiene il valore dell'orario dal campo di input utilizzando *document.form.time.value*.

Utilizza l'espressione regolare `^(0[0-9]|1[0-9]|2[0-3]):[0-5][0-9]$` per convalidare l'orario. Questa espressione regolare corrisponde a stringhe che iniziano con "0" seguito da un numero compreso tra 00 e 09, oppure iniziano con "1" seguito da un numero compreso tra 00 e 19, oppure iniziano con "2" seguito da un numero compreso tra 00 e 23. Segue poi il carattere ":" e i minuti compresi tra 00 e 59.

Utilizza il metodo *.test(timePattern)* per verificare se l'orario inserito corrisponde all'espressione regolare. Se l'orario non corrisponde, mostra un messaggio di errore che indica che l'orario non è nel formato corretto "HH:MM".

Se l'orario corrisponde all'espressione regolare, il codice successivo viene eseguito.

In sintesi, il secondo listato utilizza un'espressione regolare per semplificare la convalida del formato dell'orario, rendendo il codice più breve e leggibile.

Se escludiamo la seconda riga, la quale ha una sintassi un po' strana e, ad una prima occhiata incomprensibile, si può comunque affermare che il codice è più conciso e forse anche più elegante. Si deduce quindi che le espressioni regolari hanno un enorme potenziale. Tuttavia avremo bisogno di

molta pratica per comprendere bene il loro funzionamento, anche se come vediamo anche nel prossimo breve esempio i vantaggi sono notevoli rispetto all'uso del codice tradizionale:

```
let password = 'Tony Chan';
let controlla = /^[a-z0-9]+$/i;
if (!controlla.test(password)) {
    alert('Per la password ci vogliono lettere e numeri!');
}
```

Come si può vedere nell'esempio anche in questo caso l'espressione che controlla la nostra password è molto semplice e breve. Ancora una volta abbiamo evitato giri di inutile e arzigogolato codice, sicuramente vale la pena approfondire l'argomento, anche se sembrerebbe quasi che si debba approcciare ad un nuovo linguaggio.

I fondamenti delle espressioni regolari

Ma come è strutturato questo linguaggio? Perché è considerato complicato? Il linguaggio in questione, riassumendo e semplificando, equivale ad un modello formale dove troveremo dei *termini*, ossia il suo alfabeto, e degli *operatori*, i quali rappresentano delle operazioni specifiche, diverse da quelle classiche conosciute da tutti (come + o -). Vedremo a breve in cosa consistono questi termini e operatori.

In JavaScript, come con la maggior parte degli altri tipi di oggetti, abbiamo due modi per creare un file espressione: tramite un'espressione regolare letterale oppure costruendo un'istanza di un file RegExp.

Ad esempio, se vogliamo creare un'espressione regolare banale di solo testo potremmo farlo con un letterale regex:

```
let espressione = /prova/;
```

In alternativa, potremmo utilizzare il costruttore *new RegExp()* :

```
let espressione = new RegExp("prova");
```

Entrambi i formati danno come risultato la stessa regex creata nella variabile *espressione*.

La sintassi letterale è preferita quando la regex è nota al momento dello sviluppo mentre il costruttore viene utilizzato quando la regex è costruita durante l'esecuzione del programma.

Nella sintassi, dopo l'espressione potremmo avvalerci di cinque flag con diverse specifiche:

g: La verifica del pattern andrà effettuata per tutte le corrispondenze, non solo per la prima.

i: Nel pattern non viene fatta distinzione tra maiuscole e minuscole.

m: In questo caso la corrispondenza sarà su più righe, non solo dall'inizio alla fine della stringa.

u: Attiva il supporto *unicode* per la corrispondenza.

y: La ricerca partirà dalla posizione assegnata nella stringa bersaglio tramite *lastIndex*.

Verranno aggiunti alla fine dell'espressione o passati in una stringa come secondo parametro del costruttore *new RegExp*. Vediamo un altro esempio:

```
let stringa = "Amate le regex, le Regex ci facilitano la vita";  
let espressione = /regex/ig;
```

In queste righe verrà controllata la corrispondenza alla parola “*regex*”, per tutta la stringa (*g*) senza distinzioni di maiuscole/minuscole (*i*).

Ovviamente finora abbiamo visto solo dimostrazioni piuttosto banali, andiamo ad approfondire la conoscenza degli operatori avanzati che ci permetteranno di avere accesso ad un potere più grande.

La sintassi avanzata delle RegExp

Oltre alle banali stringhe, la sintassi delle espressioni regolari prevede alcune regole precise e degli operatori specifici che andremo ad elencare.

Intanto iniziamo sapendo che qualsiasi carattere che non sia un carattere o un operatore speciale, che vedremo più avanti, deve apparire letteralmente nell'espressione. Ad esempio, nella espressione regolare */prova/*, cinque termini rappresentano caratteri che devono apparire letteralmente in una stringa affinché corrisponda il modello espresso.

Qualche volta potrà servirci un carattere da un insieme di essi non necessariamente con un senso. In quel caso lo specificheremo con l'operatore *set* inserendo l'insieme di caratteri che vogliamo far corrispondere tra parentesi quadre: *[xyzwz]*.

Di contro, altre volte, ci servirà di trovare qualsiasi cosa che sia all'inizio di una stringa. Allora utilizzeremo il carattere speciale *^*: *^[xyzwz]*, al contrario useremo *\$* per la fine, *^&* per l'intera stringa.

Se invece volessimo trovare una corrispondenza con uno qualsiasi dei caratteri minuscoli tra x e z, scriveremo `[x-z]`. Il trattino indica che tutti i caratteri dalla x alla z compresa sono inclusi.

Per specificare che un carattere è opzionale utilizzeremo il carattere `?`. Per esempio, `/p?rova/` corrisponde sia a *prova* che a *rova*.

Il carattere `+`, come in `/pro+va/`, corrisponde a: *prova*, *proprova* e *propoprova*, ma non *va*.

Per specificare che il carattere appare una o molte volte, useremo `*`, come in `/*prova/`, che corrisponde a: *prova*, *pprova*, *pppprova* e *rova*.

Metacaratteri

Nelle regex la maggior parte dei caratteri non corrisponde alla lettera a quello che vorremmo cercare come abbiamo visto finora, invece aggiungono opzioni alla composizione della regex stessa, parliamo quindi dei *metacaratteri*, vediamoli di seguito.

Iniziamo dalle parentesi graffe, utili per alcune ripetizioni: ad esempio, `/x{2}/` equivale a una stringa di due x, altrimenti, `/x{5,8}/` corrisponde da cinque a otto caratteri consecutivi, infine, lasciando lo spazio dopo la virgola vuoto, come: `/x{7,}/` la corrispondenza sarà da almeno sette x consecutive in su.

Se volessimo applicare le condizioni a un gruppo di termini avremo a disposizione le parentesi tonde: `/(vor)+/` dove cercheremo una o più occorrenze consecutive della sottostringa *vor*.

Con il carattere barra verticale `|` potremo avere più opzioni in alternativa: `/(in)+|(il)+/` corrisponde a una o più occorrenze di *in* oppure di *il*.

Adesso vediamo un metodo leggermente più complesso per confrontare le stringhe tramite il simbolo `\` seguito da un numero che corrisponde a un riferimento che ancora non si conosce. Ad esempio `/^([abc])i\1/`, che corrisponde a una stringa che inizia con uno qualsiasi dei caratteri *a*, *b* o *c*, poi da una *i*, seguita da qualsiasi carattere che corrisponda alla prima cattura. Quindi, al momento della stesura del codice non sapremo se la cattura sarà *abito* oppure *cima* (due parole casuali corrispondenti ai parametri di ricerca della regex), dipenderà dalle variabili legate al contesto di esecuzione. In ogni caso se la parola fosse *cima* vorrà dire che la stringa da trovare sarà una che dopo la *i* dovrà avere la *m* di cima.

Nella lista seguente elenchiamo le funzioni dei metacaratteri corrispondenti ancora non esaminate che useremo nelle nostre espressioni:

\A - Inizio del testo.	+ - L'elemento prima una o più volte
\a - Codice Bell (segnale ? acustico).	- Anche senza l'elemento prima
\b - Bordo di una parola.	* - L'elemento prima zero o più volte
\B - Tutto tranne \b.	. - Tutti i caratteri singoli
\c - Codice di controllo (Ctrl).	{n} - Quantità di elementi
\d - Una cifra da 0 a 9	{n,} - Quantità di elementi o più
\D - Tutto tranne \d.	{n, m} - Quantità di elementi da un minimo ad un massimo.
\e - Codice di "escape".	\t - Codice di tabulazione.
\f - Codice di fine pagina.	\v - Codice di tabulazione verticale.
\n - Codice di fine riga.	\w - Tutti i caratteri di una parola.
\r - Codice di ritorno carrello.	\W - Tutto tranne \w.
\s - Spazio.	\xHH - Codice di un carattere ASCII.
\S - Tutto tranne \s.	\Z - Fine del testo.
\uHHHH - Codice di un carattere Unicode.	

Le espressioni regolari al lavoro!

Come abbiamo appreso nella nostra panoramica sulle espressioni regolari nella sezione precedente, ci sono due modi per creare un'espressione regolare compilata in JavaScript: tramite un valore letterale e tramite un costruttore. Vediamo nell'esempio seguente la differenza:

```
const espress1 = /\b/;
const espress2 = new RegExp('\b');

console.log(espress1); // /\b/
console.log(espress2); // /\b/
console.log(espress1 === espress2); //le due espressioni sono simili, ma non sono uguali, risultato = false
```

Come possiamo vedere, entrambe le espressioni regolari sono state eseguite correttamente dopo la creazione. Ogni regex ha una rappresentazione di oggetto univoca: ogni volta che viene compilata viene creato un nuovo oggetto di espressione regolare, diversamente da stringhe, numeri e così via, perché il risultato sarà sempre unico.

Decisamente importante è l'uso del costruttore (*new RegExp (...)*), questa tecnica ci consente di costruire e compilare un'espressione da una stringa che possiamo creare dinamicamente in fase di esecuzione. Ci sarà utilissima

anche per riutilizzare la regex più volte. Ma andiamo sul campo, proponiamo un esempio in cui cercheremo di determinare quali elementi all'interno di un documento hanno un particolare nome di classe, di cui non conosceremo il valore fino all'esecuzione. Poiché gli elementi sono in grado di avere più nomi di classi associati ad essi questo può servire come un interessante esempio di studio:

```
<html>
  <body>
    <div class="catalogo vini">
      <vini>rossi</vini>
    </div>
    <div class="catalogo birre">
      <birre>chiare</birre>
    </div>
    <span class="catalogo generale"></span>
    <script>
      function trovaElem(classe, tipo) {
        const elem = document.getElementsByTagName(tipo || "*");
        const esp = new RegExp("(^|\\s)" + classe + "(\\s|$)");
        const risultato = [];
```

In queste righe appena viste creiamo soggetti di prova di elementi con vari nomi di classi, dopodiché raccoglieremo gli elementi per tipo ed infine viene eseguita una regex utilizzando il nome della classe passato.

```
      for (let i = 0, length = elem.length; i < length; i++) {
        if (esp.test(elem[i].className)) {
          risultato.push(elem[i]);
        }
      }
      return risultato;
    }

    // Definiamo una funzione assert per i test
    function assert(condition) {
      if (!condition) {
        throw new Error("Il test ha fallito!");
      }
    }

    assert(trovaElem("catalogo", "div").length === 2);
    assert(trovaElem("catalogo", "span").length === 1);
    assert(trovaElem("catalogo").length === 3);
  </script>
</body>
</html>
```

Da questo listato possiamo vedere un po' di cose interessanti. Per iniziare, abbiamo impostato una serie di elementi `<div>` e `` del soggetto di prova con varie combinazioni di nomi e di classi. Quindi abbiamo definito la nostra funzione di controllo del nome della classe, la quale accetta come parametri sempre il nome e il tipo di elemento da controllare all'interno. Quindi raccogliamo tutti gli elementi del tipo specificato utilizzando il metodo `getElementsByTagName` integrato e impostiamo l'espressione regolare:

```
const esp = new RegExp("(^|\\s)" + classe + "(\\s|$)");
```

Per prima cosa c'è da spiegare che nell'espressione avremo un doppio carattere `\\` prima dell'operatore `s` (utilizzato per lo spazio) perché siamo all'interno di una stringa. In secondo luogo notiamo l'uso del costruttore `new RegExp()` per compilare un'espressione regolare basata sul nome della classe passato alla funzione. Questo è un caso in cui non possiamo utilizzare un file regex letterale, poiché il nome della classe che cercheremo non è noto in anticipo, quindi costruiremo e compileremo questa espressione solo una volta per evitare inutili ripetizioni di codice.

Poiché il contenuto dell'espressione è dinamico (basato sull'argomento *classe* in entrata), gestendo l'espressione in questo modo è possibile ottenere notevoli risparmi in termini di prestazioni.

Dopo che la regex è stata compilata, usarla per raccogliere gli elementi corrispondenti è un gioco da ragazzi richiamando l'espressione al momento utile all'interno del ciclo *for let*.

La precostruzione e la precompilazione di espressioni regolari in modo che possano essere riutilizzate (eseguite) più e più volte è una tecnica consigliata che, come abbiamo visto, fornisce notevoli vantaggi.

Adesso vedremo che il massimo dell'utilità rispetto alle espressioni regolari si realizza quando catturiamo i risultati che si trovano in modo da poter fare qualcosa con essi. Determinare se una stringa corrisponde a un modello è un ovvio primo passo e spesso tutto ciò di cui abbiamo bisogno, ma determinare cosa è stato abbinato è utile anche in molte situazioni.

RegExp: la cattura

Supponiamo di voler estrarre un valore incorporato in una stringa complessa. Abbiamo visto nel capitolo 3 che normalmente avremmo usato i metodi `.substring`, `.substr` e `.slice`, tuttavia con una `RegExp` dovremo servirci del

metodo `.match(regex)`, il quale confronta una stringa con un'espressione regolare.

Se l'argomento *regex* non è un'espressione regolare, `.match()` lo convertirà in un'espressione regolare utilizzando il costruttore *RegExp()*.

Il metodo `.match()` si abbina ad un marcatore globale (*g*) restituendo un array che memorizza tutti i risultati corrispondenti.

Se non si utilizza il marcatore *g*, verrà riportata solo la prima corrispondenza, ma in aggiunta sarà evidenziato il relativo gruppo di cattura. Vediamo l'esempio di seguito per i dettagli:

```
let stringa = "Prezzo pendrive 16gb: €10, 32gb: €20"
let cattura = stringa.match(/\€\d+/g);
console.log(cattura);
```

Abbiamo visto come utilizzare il metodo `.match()` con l'indicatore globale *g*, il quale, dopo aver cercato qualunque numero che abbia il simbolo € prima di esso, restituirà un array di corrispondenze: (2) [`'€10'`, `'€20'`].

L'esempio seguente illustra come utilizzare il metodo `.match()` con un'espressione regolare che non dispone di un indicatore globale *g*. Restituisce un array della prima corrispondenza con proprietà aggiuntive:

```
let stringa = "Prezzo pendrive 16gb: €10, 32gb: €20"
let cattura = stringa.match(/\€\d+/);
console.log(cattura);
```

Risultato:

```
['€10', index: 22, input: 'Prezzo pendrive 16gb: €10, 32gb: €20', groups: undefined].
```

Vediamo cosa viene mostrato:

index: come abbiamo già visto, partendo da sinistra verso destra e da 0, riporta il segnaposto equivalente della stringa cercata, cioè il simbolo € che si trova appunto nella posizione n° 22 (P 0, r 1, e 2, z 3, z 4, o 5, spazio 6, e così via).

input: la stringa che abbiamo preso in esame.

groups: È l'oggetto dei gruppi di cattura denominati le cui chiavi e valori sono rispettivamente i nomi e i gruppi di cattura da noi indicati. In questo esempio, è *undefined* perché non è stato definito alcun gruppo.

Il codice seguente mostra come utilizzare il metodo `.match()` con il gruppo di acquisizione denominato. Cattura *prosperose* in un gruppo chiamato *donne*:

```
let s = "Quelle prosperose sono le donne che guardo di più"
let r = /(?!<donne>prosperose) sono le donne/;
let ri = s.match(r);
console.log(ri);
```

Risultato:

```
index: 7, input: 'Quelle prosperose sono le donne che guardo di più', groups: {...}
0: "prosperose sono le donne"
1: "prosperose"
groups: {donne: 'prosperose'}
index: 7
input: "Quelle prosperose sono le donne che guardo di più"
length: 2
[[Prototype]]: Array(0)
```

Come esercizio scrivete del codice con almeno tre voci nel gruppo di acquisizione (ad esempio: *donne: prosperose, alte, simpatiche*).

In alcuni casi potremmo aver bisogno di controllare più corrispondenze in una stringa di testo con gruppi di acquisizione in un loop, in questo caso invece di `.match()` potremo utilizzare il metodo `.exec()`. Anche se da poco è stato introdotto il nuovo e migliorativo metodo `.matchAll()` che vedremo poco più avanti. Procediamo in ogni caso a dare una breve occhiata:

```
const espressione = new RegExp('fritt[a-z]*','g');
const stringa = 'frittura, frittata, frittella, fringuello';

let controlla;
while ((controlla = espressione.exec(stringa)) !== null) {
  console.log(`Trovato:   ${controlla[0]}   start=${controlla.index}
end=${espressione.lastIndex}.`);
}
```

Praticamente in questo ciclo viene controllata la corrispondenza di tutte le parole che iniziano per *fritt*, elencando i risultati e la loro posizione. Vediamo ancora un ultimo esempio da confrontare con i nuovi metodi nel prossimo paragrafo:

```
let testString = "ciao ciao 123 ciao";
let regex = /ciao/gi;
let match;
```

```
// con exec troveremo le occorrenze del pattern nella stringa

while (match = regex.exec(testString)) {
  console.log(match[0]); // "ciao", "ciao"
}

// utilizziamo replace per sostituire tutte le occorrenze del
pattern nella stringa con un'altra stringa

let replacedString = testString.replace(regex, "hello");
console.log(replacedString); // "hello hello 123 hello"
```

Non ci soffermiamo troppo e andiamo a vedere come si è evoluto Javascript.

I metodi matchAll e replaceAll

Adesso vedremo ed utilizzeremo due nuovi metodi: *.matchAll()* e *.replaceAll()* standardizzati rispettivamente nel 2020 e nel 2021 dalle ultime edizioni dell'ECMA.

.matchAll() andrà praticamente a sostituire il vecchio e appena visto metodo *.exec()*, rendendo anche possibile un iterazione con *for*, sicuramente molto più utile. Vediamo gli esempi già proposti poco prima reinterpremati con il nuovo metodo:

```
const espressione = new RegExp('fritt[a-z]*', 'g');
const stringa = 'frittura, frittata, frittella, fringuello';
const controlla = stringa.matchAll(espressione);

for (const controlli of controlla) {
  console.log(`Trovato:   ${controlli[0]}      start=${controlli.index}
end=${controlli.index + controlli[0].length}.`);
}
```

I risultati sono gli stessi dell'espressione precedente che utilizzava *exec*. I vantaggi, come accennato sono diversi, inanzitutto la possibilità di iterare con il ciclo *for*, inoltre la possibilità di usare la sintassi *spread* (...) che ci serve per passare i risultati in un elenco di qualsiasi tipo e il metodo *Array.from()* il quale, copiando l'originale, crea un nuovo oggetto iterabile o convertibile in un vettore (array). Vediamo gli esempi:

```
const espressione = /t(r)(e(\d?))/g;
const stringa = 'tre330tre';
```

```
const risultato = [...stringa.matchAll(espressione)];

console.log(risultato[0]);
console.log(risultato[1]);
```

Qui sotto troviamo ancora il listato visto all'inizio del capitolo modificato con l'aiusilio di *Array.from()*:

```
const espressione = new RegExp('fritt[a-z]*', 'g');
const stringa = 'frittura, frittata, frittella, fringuello';
const controlla = stringa.matchAll(espressione);

Array.from(stringa.matchAll(espressione), m => m[0]);
```

Siamo così arrivati ad analizzare il metodo *.replaceAll()* il quale molto semplicemente sostituisce una data stringa con un'altra sempre da noi indicata. Quindi adesso proviamo il listato visto precedentemente con l'aiusilio di entrambi i nuovi metodi:

```
let testString = "ciao ciao 123 ciao";
let regex = /ciao/gi;

// matchAll restituisce un iteratore che contiene tutte le
// occorrenze del pattern nella stringa

let matches = testString.matchAll(regex);
for (let match of matches) {
  console.log(match[0]); // "ciao", "ciao"
}

// replaceAll sostituisce tutte le occorrenze del pattern nella
// stringa con un'altra stringa

let replacedString = testString.replaceAll(regex, "hello");
console.log(replacedString); // "hello hello 123 hello"
```

In questo esempio, utilizziamo una stringa chiamata *testString* che contiene la parola "ciao" ripetuta più volte. Creiamo poi un'espressione regolare chiamata *regex* che cerca la parola "ciao" in modo insensibile alle maiuscole o minuscole.

Utilizziamo quindi *matchAll* per ottenere un iteratore che contiene tutte le occorrenze del pattern nella stringa, mostrandole sulla console.

Successivamente, utilizziamo *replaceAll* per sostituire tutte le occorrenze del pattern con la stringa "hello". Nel risultato tutte le occorrenze di "ciao" sono

state sostituite con "hello".

Nel capitolo 3 abbiamo visto gli altri metodi per lavorare con le stringhe, la differenza da rimarcare è che questi ultimi appena visti si applicano alle espressioni regolari, teniamo bene a mente la distinzione.

Evitare catture doppie con le sottoespressioni passive

Come abbiamo notato, le parentesi hanno un doppio dovere: non solo raggruppano i termini per le operazioni, ma specificano anche le acquisizioni. Questo di solito non è un problema, ma nelle espressioni regolari in cui sono in corso molti raggruppamenti, potrebbe causare molte catture inutili, e quindi rendere noioso lo smistamento delle acquisizioni risultanti, vediamo un esempio:

```
const model = /((calciatore-)+) pallone/;
```

Qui, l'intento è creare una regex che consenta al prefisso *calciatore-* di apparire una o più volte prima della parola *pallone*, e vogliamo catturare l'intero prefisso. Questa regex, come visto anche in altri esempi precedenti, richiede due serie di parentesi:

- Le parentesi che definiscono l'acquisizione (tutto prima della stringa *pallone*).
- Le parentesi che raggruppano il testo *calciatore* per l'operatore +.

Funziona tutto bene, ma si traduce in più della singola acquisizione prevista a causa dell'insieme interno di parentesi di raggruppamento.

Per indicare che un insieme di parentesi non dovrebbe risultare in una cattura, la sintassi dell'espressione regolare ci permette di mettere la notazione `?:` immediatamente dopo la parentesi di apertura sorella. Questo è noto come sottoespressione passiva. Quindi, andremo a cambiare l'espressione regolare in:

```
const model = /((?: calciatore-)+) pallone/;
```

facendo così in modo che solo la serie esterna di parentesi crei un'acquisizione. Le parentesi interne sono state convertite in una sottoespressione passiva, verifichiamolo subito:

```
const model = /((?: calciatore-)+) pallone/;  
const calciatori = "calciatore-calciatore-pallone".match(model);
```

```
assert (calciatori.length === 2, "È stata restituita una sola  
cattura.");  
assert (calciatore[1] === "calciatore-calciatore-", "Corrisponde a  
entrambe le parole, senza alcuna ulteriore cattura.");
```

Eseguendo queste prove, possiamo vedere che la sottoespressione passiva `/((?: calciatore-)+) pallone/` come già detto previene acquisizioni non necessarie.

Laddove possibile nelle nostre espressioni regolari, dovremmo sforzarci di seguire questo esempio al posto della cattura quando essa non è necessaria, in modo che il motore delle espressioni avrà molto meno lavoro da fare nel ricordarla e restituirla. Se non abbiamo bisogno di risultati acquisiti, è ovvio che non è necessario chiederli. Il rovescio della medaglia sarà solamente una grammatica più complessa e forse indecifrabile, ma ne varrà la pena per far sì che il nostro codice sia più snello.

Sostituzione utilizzando le funzioni

Il metodo di sostituzione dell'oggetto stringa è un metodo potente e versatile, che abbiamo visto brevemente utilizzato nella nostra discussione sulle acquisizioni. Quando un'espressione regolare è vista come il primo parametro da sostituire, provocherà una sostituzione su una corrispondenza (o corrispondenze se è globale) al pattern piuttosto che su una stringa fissa.

Ad esempio, supponiamo di voler sostituire tutti i caratteri maiuscoli in una stringa con A. Potremmo scrivere quanto segue:

```
"RSTUVXyz".replace(/[AZ]/ x, "A")
```

Il risultato sarà: `AAAAAAyz`, ma forse la caratteristica più potente presentata dalla sostituzione è la capacità di promuovere una funzione come valore di sostituzione piuttosto che una stringa fissa.

Quando il valore di sostituzione è una funzione, esso viene invocato per ogni corrispondenza trovata con un elenco di parametri variabili:

- *Il testo completo corrispondente.*
- *Le catture corrispondenti, un parametro per ciascuna.*
- *L'indice della corrispondenza all'interno della stringa originale.*
- *La stringa di origine.*

Ciò che deriva dalla funzione funge da valore sostitutivo. Proviamo a vedere un esempio pratico:

```

<!DOCTYPE html>
<html>
<body>
<p>Se clicchi qui sotto le minuscole diverranno MAIUSCOLE!</p>
<p id="prova">Ecco qui come alcune parole sono cambiate.</p>
<button onclick="laFunzione()">Clicca qui!</button>
<script>
function laFunzione() {
let prendi = document.getElementById("prova").innerHTML;
let cambia = prendi.replace(/qui|alcune|cambiate/gi,
function(a) {
return a.toUpperCase();
});
document.getElementById("prova").innerHTML = cambia;
}
</script>
</body>
</html>

```

Poiché un'espressione regolare globale farà sì che tale funzione di sostituzione venga eseguita per ogni corrispondenza in una stringa sorgente, questa tecnica può anche essere estesa oltre le sostituzioni meccaniche. Possiamo usare la tecnica come mezzo di attraversamento delle stringhe, invece di eseguire *exec()* con un loop *while* o anche *matchAll()*, il suo sostituto, come abbiamo visto in precedenza.

Ad esempio, supponiamo di voler prendere una stringa di query e convertirla in un file formato alternativo che si adatta ai nostri scopi. Trasformeremo una stringa come quella qui sotto:

```
valore=5&lettera=x&valore=10&lettera=y&valore=15&lettera=z
```

in questo:

```
valore=5,10,15&lettera=x,y,z
```

Vediamo come con *replace* riusciamo ad arrivare al risultato, come mostrato qui sotto:

```

function riduci(dati) {
  const codici = {};
  dati.replace(/([^\&]+)=([^\&]*)/g, function(match, cod, val) {
    codici[cod] = (codici[cod] ? codici[cod] + "," : "") + val;
    return "";
  });
}

```

```

    const risultato = [];

    for (let cod in codici) {
        risultato.push(cod + "=" + codici[cod]);
    }

    return risultato.join("&");
}

const assert = (condition, message) => {
    if (!condition) {
        throw new Error(message);
    }
};

assert(
    riduci("valore=5&lettera=x&valore=10&lettera=y&valore=15&lettera=z") ===
    "valore=5,10,15&lettera=x,y,z",
    "Il test ha fallito!"
);

console.log("Il test è passato con successo!");

```

Da notare l'uso del metodo di sostituzione delle stringhe anche come mezzo per attraversarle per i valori, piuttosto che la classica ricerca e sostituzione. L'idea è quella di passare una funzione come argomento del valore sostitutivo e invece di restituire un valore, usarlo come mezzo di ricerca.

Il listato appena mostrato dichiara prima una chiave *hash* (una sorta di array che ci consente di effettuare diverse operazioni) in cui archiviamo le chiavi e i valori che troviamo nella stringa di query di origine. Quindi chiamiamo il metodo di sostituzione sulla stringa di origine, passando una regex che corrisponderà alle coppie chiave-valore e acquisiamo la chiave e il valore. Passiamo anche una funzione a cui verrà passata la corrispondenza completa, l'acquisizione della chiave e l'acquisizione del valore. Questi valori acquisiti vengono archiviati nell'hash per riferimento futuro.

Notate che restituiamo la stringa vuota perché non ci interessa quali sostituzioni avvengono al file stringa di origine: stiamo usando solo gli effetti collaterali piuttosto che il risultato.

Dopo i ritorni di sostituzione, dichiariamo un array in cui aggregheremo i risultati e itereremo le chiavi che abbiamo trovato, aggiungendole ciascuna all'array. Infine, ci uniamo a ciascuno dei risultati che abbiamo memorizzato nell'array utilizzando & come delimitatore e restituiamo il risultato.

Usando questa tecnica, possiamo cooptare il metodo di sostituzione dell'oggetto String come nostro meccanismo di ricerca delle stringhe. Il risultato non è solo veloce, ma anche semplice ed efficace. Si noterà che a dispetto della sua grandezza il listato è molto “performante”.

Tutte queste tecniche utilizzate con le espressioni regolari possono avere un enorme impatto sul modo in cui scriviamo gli script. Vediamo come applicare ciò che abbiamo imparato per risolvere alcuni problemi comuni che potremmo incontrare.

Risoluzione di problemi con le RegEx

Spesso nell'uso di espressioni regolari, vogliamo abbinare caratteri alfanumerici, come ad esempio per scrivere ed eseguire un identificatore per una parte di codice CSS. Ovviamente dovremo assicurarci che i caratteri alfabetici provengano non solo dal set ASCII Italiano o Inglese per evitare di perdere qualcosa. Di conseguenza è consigliabile espandere il set per includere caratteri Unicode, poiché supporta esplicitamente più lingue non coperte dal set di caratteri alfanumerici tradizionale, vediamo come:

```
const setVari = "\u4e3a \u7269 \u30D1 \u5927 \u800c";  
const tuttOk = /[\\w\\u0080-\\uFFFF_-] +/;  
  
assert(setVari.match(tuttOk)); // true
```

Questo elenco include l'intera gamma di caratteri *Unicode* nella corrispondenza creando una classe di caratteri che includa il termine `\w`, in modo che corrisponda a tutti i caratteri "normali" delle parole, più un intervallo che copre l'intero set di caratteri Unicode sopra U + 0080. A partire da 128 ci dà alcuni caratteri ASCII alti insieme a tutti i caratteri Unicode inil piano multilingue di base.

Gli astuti tra di voi potrebbero notare che aggiungendo l'intero intervallo di caratteri Unicode sopra `\u0080`, vengono abbinati non solo i caratteri alfabetici, ma anche tutta la punteggiatura Unicode e altri caratteri speciali (il simbolo dell' €, per esempio). Ma va bene, perché il punto dell'esempio è mostrare come abbinare i caratteri Unicode in generale. Se si dispone di un intervallo specifico di caratteri che si desidera abbinare, è possibile utilizzare la lezione di questo esempio per aggiungere qualsiasi intervallo desiderato alla classe di caratteri.

È comune per gli autori utilizzare nomi conformi agli identificatori di programma quando assegnano valori id agli elementi della pagina, ma questa

è solo una convenzione; I valori id possono contenere caratteri diversi da "parola", compresa la punteggiatura. Ad esempio, lo sviluppatore web potrebbe utilizzare il modulo del valore id: *aggiornamento* per un elemento. Uno sviluppatore di librerie, quando scrive un'implementazione per, ad esempio, un foglio di stile CSS, vorrebbe supportare i caratteri speciali. Ciò consente all'utente di specificare nomi complessi che non sono conformi alle convenzioni di denominazione tipiche. Quindi sviluppiamo una regex che consentirà la corrispondenza dei caratteri speciali. Proviamo:

```
const carspeciali = /^((\w+)|(\.\.))+$/;
const prova = ["modulo", "modulo\\.agg\\.", "form\\:aggiornamento"];

for (let x=0; x<prova.length; x++) {
  assert(pattern.test(prova[x]), prova[x]); // true
}
```

Test espressioni regolari

Per un controllo della nostra espressione regolare possiamo utilizzare il metodo *test()* dell'oggetto *RegExp* il quale restituirà *true* in caso di successo o *false* in caso contrario. Vediamo come impostare questo controllo:

```
if (espressReg.test(regex))
{
  // "regex" promossa a pieni voti
}
else
{
  // "regex" non va bene, ricontrollare
}
```

Possiamo anche utilizzare un'impostazione differente per lo stesso identico scopo. Per fare in modo che il nostro listato agisca solo se l'espressione regolare non è verificata utilizzeremo questa sintassi:

```
if (!espressReg.test(regex))
{
  // Qui ci sarà il nostro errore
}
```

Abbiamo visto che con *test()* avremo come risultato semplicemente *true* o *false*. Tuttavia con il metodo *exec()* avremo la possibilità di ottenere la

stringa corrispondente al pattern (se trovata) oppure null (in caso di match non riuscito). Vediamo cosa accade dopo aver definito una stringa:

```
let txt = 'Tony Chan programmatore Jackie Chan attore';
let verTxt = /^Tony/;
let prova = verTxt.exec(txt);

document.write(prova);
```

Il risultato sarà ovviamente Tony. Come abbiamo visto con questo metodo avremo maggiori possibilità di controllo.

Siamo giunti alla fine e indubbiamente ci saremo resi conto che anche avendo una buona esperienza di programmazione, con le espressioni regolari è tutto un altro paio di maniche, bisognerà ancora una volta studiare e sperimentare molto. Ricordiamoci che esiste sempre una grande community e alcuni siti specializzati che ci aiuteranno con qualunque problema: <http://www.regexr.com> , <https://regexpal.com/> , <https://txt2re.com> , <https://regex101.com/> e ve ne sono molti altri, basterà utilizzare un po' il nostro motore di ricerca.

Prima di provare alcuni esercizi volevo ringraziare tutti per la lettura e augurare i migliori auspici di buona fortuna con JavaScript e le sue applicazioni.

Quiz & esercizi

1) Correggere l'espressione per far sì che passi il controllo password:

```
let password = 'tonychan9';
let controlla = /^[a-z0-8]+$ /i;
if (!controlla.test(password)) {
    alert('Per la password ci vogliono lettere e numeri!');
}
```

2) Correggere l'espressione in modo che possa recuperare i prezzi:

```
let stringa = "Prezzo pendrive 16gb: 10 euro, 32gb: 20 euro"
let cattura = stringa.match(/\euro\d+/g);
console.log(cattura);
```

3) Completare l'espressione per ottenere una frase corretta:

```
const testo = 'Stamattina il lume ha illuminato tutte le ilume';
```

```
const espr = /lume/ig;  
console.log(testo.replaceAll(espr, '????'));
```

- 4) Le sottoespressioni passive hanno una sintassi che necessita delle parentesi per poter funzionare?
- 5) Scrivere una regex per verificare se una stringa è un indirizzo email valido.
- 6) Scrivere una regex per estrarre tutti i numeri da una stringa.
- 7) Scrivere una regex per verificare se una password soddisfa determinati criteri (almeno 8 caratteri, almeno una lettera maiuscola e almeno un numero).
- 8) Scrivere una regex per dividere una stringa in parole.
- 9) Scrivere una regex per rimuovere tutti gli spazi vuoti da una stringa.
- 10) Scrivere una regex per verificare se una stringa è un numero di telefono valido (formato XXX-XXX-XXXX).

Riassunto

Questo capitolo l'abbiamo volutamente messo come ultimo per evitare che qualcuno potesse rimanere impantanato prima di terminare il percorso di studio. A parte questo le espressioni regolari sono un elemento fondamentale per tutti i programmi e i linguaggi di un certo livello. Anche se, purtroppo, per questo paragrafo non abbiamo potuto usufruire di uno spazio adeguato all'importanza e alla complessità dell'argomento. In conclusione il suggerimento banale è quello di esercitarsi a lungo per fare proprie tutte le sfumature e le possibilità offerte da questo strumento.

Ti ringrazio per aver acquistato questo libro, spero sia stato utile e ti sia piaciuto.

Se vorrai lasciare una recensione su Amazon sarà molto gradita, grazie mille, ti voglio bene.

Tony Chan.

Link soluzioni esercizi:



Il link indicato è indirizzato verso un server gratuito (*mega.nz*) da dove potrai scaricare le soluzioni degli esercizi in formato pdf in modo da poter usufruire al meglio di tutti i listati presenti nel libro, senza quindi ogni volta doverli riscrivere da zero.

Bibliografia

I sottoelencati siti internet hanno fornito ispirazione per diversi argomenti trattati nel libro:

<https://www.swiftios.it/>

<https://qastack.it/>

<https://www.w3schools.com/>

<https://stackoverflow.com/>

<http://www.datrevo.com/>

<https://it.javascript.info/>

<https://www.w3bai.com/>

<https://ichi.pro/>

<https://it.wikibooks.org/wiki/JavaScript>

<https://goessner.net/>

<https://gabrieleromanato.com/>

<https://www.marchettidesign.net/>

<https://yourinspirationweb.com/>

<https://it.wikipedia.org>

<https://www.codingcreativo.it/>

<https://www.html.it/>

E anche i testi:

C# Java, PHP, Python, la guida completa alla programmazione ad oggetti

di Tony Chan

HTML5 & CSS3, la guida completa *di Tony Chan*

Python, la guida completa *di Tony Chan*

PROGRAMMARE, Impara velocemente *di Tony Chan*

Python & Ethical Hacking *di Tony Chan*

Trading con Python *di Tony Chan*

Link per un video su come creare una blockchain con JS:

per il video su youtube e i dettagli: <https://tinyurl.com/mfkzryec>