ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea Triennale in Informatica

Evoluzione storica di ECMAScript

Un cammino tra le sue feature

Relatore: Chiar.mo
Prof. FABIO VITALI

Presentata da:
ELEONORA MINGARELLI

Sessione straordinaria

Anno Accademico 2019-2020

Ai miei genitori,

alla vita.

Indice

Capitolo 1

Introduzione

Con la presente dissertazione è mio scopo illustrare come sia cambiato negli anni ECMAScript (la versione ufficiale di JavaScript, standardizzata dall'associazione ECMA[1]) attraverso un'analisi di alcune delle sue funzionalità principali. Vuole essere un'indagine non solo teorica, ma anche pratica per scoprire se l'evoluzione abbia portato effettivamente a un progresso dal punto di vista del codice e dell'efficienza del risultato.

Prima di tutto però è da chiarire cosa sia ECMAScript: termine spesso utilizzato come sinonimo di JavaScript, nei casi migliori, altre volte addirittura ignoto. Il mio lavoro ha avuto inizio quindi da una ricerca relativa a questo e alla scoperta di che storia avesse alle sue spalle, almeno in generale; tale argomento viene trattato nel capitolo due.

Successivamente ho analizzato più nello specifico le release, andando a radunare gli arricchimenti apportati da ognuna di esse e poi selezionando, tra tutte, le "feature" che sarebbero state al centro della mia attenzione. Questa parte di lavoro non è stata riportata nel testo, ma chi fosse interessato può trovare un elenco dei cambiamenti principali per ogni edizione fino al 2020 presso [LH20].

Una volta compiuta la scelta delle innovazioni più interessanti, mi sono immersa nella loro disamina, documentandomi sui singoli costrutti e sperimentandoli, per comprenderli meglio e per ideare delle situazioni appropriate di utilizzo. Ho raccolto dunque i punti caratteristici per ognuno di essi nel capitolo tre. Tale analisi mi ha spesso portato a voler capire perchè avessero luogo certi fenomeni durante l'esecuzione del codice, così ho pensato

[1]https://www.ecma-international.org/

di condividere alcune delle spiegazioni che sono riuscita a elaborare riunendole in un'appendice (*Appendice A*) .

Per la realizzazione pratica ho deciso di creare un unico progetto che includesse una possibile implementazione di tutte le funzionalità presentate e che, inoltre, riportasse per ciascuna un esempio di come lo stesso obiettivo fosse raggiungibile prima della sua introduzione nel linguaggio. Ho optato per la produzione di un'applicazione divisa in più componenti, ognuna distinta per i costrutti lì impiegati. Fortunatamente ho potuto mettere a confronto tra loro alcune delle feature in esame scegliendo un caso d'uso loro comune, in modo da ridurre il numero di versioni necessarie. La presentazione del progetto e delle parti di codice salienti è riportata nel capitolo quattro.

Infine mi sono dedicata alle valutazioni, di supporto per concludere se l'evoluzione storica del linguaggio sia stata di progresso o regresso, in base a quanto esplorato. In particolar modo ho annotato, per ogni caso ottenuto, il tempo d'esecuzione. Quest'ultimo è un elemento fondamentale oggi giorno perchè gli utenti sono sempre più abituati ad una fruizione e interazione immediate delle pagine web e sempre più infastiditi da eventuali tempi di attesa; la scelta del mezzo più efficiente in questo senso, per l'inserimento di una determinata funzionalità, va incontro a tale necessità. Per quanto riguarda la stesura del codice in sè, ho sottolineato nel corso di tutta la dissertazione quali ho trovato essere i punti a favore (alcune volte a sfavore) dell'impiego delle diverse feature, terminando con una tabella di compatibilità tra browser che fortunatamente si è rivelata essere pressoché neutrale. Le valutazioni quantitative sono state raccolte nel capitolo cinque.

In conclusione, a livello di programmazione mi sono trovata progressivamente meglio nell'impiego dei costrutti via via più recenti: sono risultati più comodi, visivamente puliti e logici. Anche gli esiti delle valutazioni mi hanno confermato come il bilancio evolutivo di ECMAScript, secondo quanto analizzato, sia positivo.

Capitolo 2

Javascript ed ECMAScript

2.1 Ecma International e ECMAScript

> «Ecma International is an industry association dedicated to the standardization of information and communication systems.» [EIa]

Ecma International nasce il 17 Maggio del 1961 come organizzazione "not-for-profit" sotto il nome di *ECMA* (European Computer Manufacturers Association), successivamente cambiato, nel 1994, per renderne intuitiva l'inclinazione all'universalità.
La creazione di standard, avendo come idealità la comunità globale, è da considerarsi infatti un bene per la collettività: è alla base dell'instaurazione di una conoscenza condivisa e di una guida verso un progresso nell'interoperabilità.
Dal punto di vista di un'organizzazione, avere uno standard significa avere un solido punto di partenza per il proprio lavoro, economizzando le risorse.
Così, *Isabelle Valet-Harper*[2] riporta nell'introduzione del Memento[3] 2020:

> «Standardization provides the means for economical solutions to complex technologies, and is required for data interchange and interoperability. Moreover, it is most effective when performed in a

[2] Presidente di Ecma International nelle annate 2013-2014 e 2020/2021.
[3] Sono così chiamati i report annuali di Ecma International.

> pre-competitive mode during product development and with all interested parties involved.» [EI2.2]

L'istituzione è costituita da più comitati tecnici per la creazione e la redazione dei diversi standard di cui si occupa; interessante da citare in questa sede è il TC39 [EIb], deputato allo sviluppo e alla manutenzione di tutto ciò che concerne ECMAScript.
In particolare ECMA262 è il documento che definisce le specifiche aggiornate ad ogni nuova edizione di ECMAScript: un linguaggio di programmazione "general-purpose".

> «ECMAScript was originally designed to be a Web scripting language, providing a mechanism to enliven Web pages in browsers and to perform server computation as part of a Web-based client-server architecture. ECMAScript is now used to provide core scripting capabilities for a variety of host environments. [...] ECMAScript usage has moved beyond simple scripting and it is now used for the full spectrum of programming tasks in many different environments and scales. As the usage of ECMAScript has expanded, so have the features and facilities it provides. ECMAScript is now a fully featured general-purpose programming language.» [TC3921]

ES[4] è definito da regole grammaticali, semantiche e pragmatiche, continuamente arricchite e mantenute, che guidano alla sua implementazione in linguaggi di programmazione ad esso conformi, gergalmente chiamati "dialetti":

> «A dialect derives most of its lexicon and syntax from its parent language, but deviates enough to deserve distinction.» [MA17]

[4] Acronimo di ECMAScript, usato in complementarietà con il numero di edizione o anno (dettagli nel prossimo paragrafo).

2.2 La storia in breve

"In origine fu JavaScript": nonostante ora JavaScript sia la più famosa implementazione di ECMAScript conosciuta, in realtà ne è stata la matrice.
La volontà principale, che portò alla nascita di questo nuovo linguaggio, fu quella di dotare i browser di dinamicità e logica[5]: fino a quel tempo erano stati pressoché solo dei visualizzatori di documenti ipertestuali.
JavaScript fu creato da *Brendan Eich*, nel 1995, sotto Netscape[6], con il nome di Mocha. Questo nome, assegnato al tempo da *Marc Andreessen*[7], fu presto cambiato in LiveScript per poi divenire finalmente JavaScript, in seguito al pagamento della licenza sul marchio allora detenuto da Sun Microsystem[8] , rendendo poi quest'ultimo un collaboratore. Le due grandi compagnie rilasciarono JavaScript 1.0 su Navigator 2, il browser di Netscape [W21a].
Il mondo informatico non rimase a guardare: Microsoft, in particolare, architettò il suo browser, Internet Explorer, e in breve tempo rilasciò le Microsoft Windows Script Technologies inclusive di VBScript [W21b], ispirato a Visual Basic, e della sua versione di JavaScript: JScript.
Da qui l'evidente necessità di una standardizzazione: Netscape e Sun si rivolsero a ECMA e, nel Giugno del 1997, venne pubblicato il primo documento ECMA-262, redatto con la collaborazione fra i più importanti produttori di browser, Microsoft incluso.
Il nome ECMAScript fu un compromesso neutrale fra le parti, commentato così da Brendan Eich:

> «ECMAScript was always an unwanted trade name that sounds like a skin disease.» [DB20]

Nel Giugno del 1998 e nel Dicembre del 1999 furono rispettivamente rilasciate le edizioni due e tre di ECMAScript: la prima consisteva, principalmente, in degli aggiustamenti alla stesura precedente, mentre la seconda apportò i primi veri cambiamenti al linguaggio con l'introduzione di nuovi costrutti [BI15].

[5] Anche solo la validazione degli input avveniva server-side, con ovvie ripercussioni sull'usabilità.
[6] Compagnia indipendente americana di computer services.
[7] Fondatore di Mosaic, uno dei primi browser, poi di Netscape in collaborazione di James H. Clark, fondatore della Silicon Graphics, Inc.
[8] Compagnia americana di computer, componenti e software, acquistata nel 2009 da Oracle Corporation.

Per quanto riguarda ES4, i confronti iniziarono nel 2000 e divennero mano a mano degli scontri fra chi (Microsoft e Yahoo) riteneva che vi dovesse essere prima una release minore, ES3.1, e chi (Adobe, Mozilla, Opera & Google) puntava a una major-release, ES4. Così il banco del TC39 si ritrovò diviso in due [BE08].

Nel Luglio del 2008 fu tenuta una riunione ad Oslo che riportò i diversi colossi sulla strada della collaborazione. ECMAScript 4 fu definitivamente abbandonato e in risposta fu deciso che il comitato avrebbe lavorato insieme su due differenti edizioni: prima ES5 rilasciata a Dicembre del 2009, minoritaria, poi ES6 pubblicata nel Giugno del 2015, più importante e ricca.

L'organizzazione pattuì infine, nel 2014, che le release sarebbero divenute annuali[9] e lo stesso ES6 venne ribattezzato ECMAScript 2015 [AR18].

Di fatto la sesta edizione fu la chiave di svolta, sia per la quantità di supplementi al linguaggio che l'accompagnarono, ma soprattutto per la ritrovata intesa tra i membri del comitato e venne, per questo, etichettata "Harmony" dallo stesso Eich:

> «The Ecma TC39 meeting in Oslo at the end of July was very productive, and if we keep working together, it will be seen as seminal when we look back in a couple of years. Before this meeting, I worked with John Neumann, TC39 chair, and ES3.1 and ES4 principals, especially Lars Hansen (Adobe), Mark Miller (Google), and Allen Wirfs-Brock (Microsoft), to unify the committee around shared values and a common roadmap. This message is my attempt to announce the main result of the meeting, which I've labeled "Harmony". [...] A split committee is good for no one and nothing, least of all any language specs that might come out of it.» [BE08]

[9] Ogni anno, solitamente in Giugno, viene redatto un nuovo documento ECMA-262.

Capitolo 3

Analisi di alcune feature di ECMAScript

Una volta inquadrata la storia di ECMAScript, l'attenzione è stata rivolta più nello specifico alla scoperta delle release: per ognuna di esse sono state individuate le aggiunte apportate al linguaggio [LH20]. A questo è seguita una fase che ha portato alla selezione di una serie di feature[10], tra quelle più rivoluzionarie e più usate, su cui concentrare lo studio.
La finalità di questo capitolo è di presentarle, una ad una, in ordine temporale di pubblicazione.

3.1 Try

Il costrutto Try, introdotto con ES3 nel 1999, fornisce per la prima volta il linguaggio di un metodo specifico per la gestione delle eccezioni a runtime [AR14]. Sin dalle origini venivano usati i classici blocchi condizionali "if...else" nei quali il programmatore è tenuto a pensare a tutti i possibili casi di errore per poterli gestire; nell'eventualità ne venisse dimenticato anche solo uno, questo porterebbe lo script ad uno stato di errore bloccandolo per intero o almeno in parte. Un altro svantaggio è che, usando l'If, per rispondere in modo specifico ad ogni tipo di errore si finisce per farlo in punti distinti del codice, ovvero nel blocco corrispondente al caso. L'uso di Try previene dal ricadere in questi due scenari grazie al suo stesso funzionamento.

[10] ES3: Try; ES5: Getters/Setters; ES2015: Modules, Promises, Map, Spread operator; ES2017: Async/Await; ES2018: Flat(); ES2020: Optional chaining.

La sintassi dell'istruzione, come riportato nel relativo Ecma-262, è la seguente:

```
«  TryStatement:
         try Block Catch
         try Block Finally
         try Block Catch Finally
   Catch:
         catch ( Identifier ) Block
   Finally:
         finally Block »
```
[EI99]

La parola chiave *try* è sempre seguita, nell'ordine, da un blocco di istruzioni che vengono eseguite in maniera sincrona, e da uno o entrambi i blocchi *catch* e *finally*. Durante l'esecuzione di tale parte potrebbero verificarsi diversi errori, previsti o meno, la cui gestione viene reindirizzata: quando sopravviene o viene scatenata un'eccezione, l'esecuzione del Try si conclude dove è giunta e il controllo viene passato direttamente al Catch, se presente, o al Finally.

```
try {
   let a =  Math.random() * 11 ;  //numero random tra 0 e 10
   if (a < 5) throw new Error( "Messaggio di errore!" );
   console.info("Questa linea verrà stampata in caso di successo.");
} catch (error) {
   console.error( error.message );
} finally {
   console.info( "Questa linea verrà stampata sempre." )
}
```

Esempio [3.1]: Il costrutto try

Il blocco Catch riceve l'errore e permette di rispondere ad esso. Può essere creato un Catch condizionale se combinato con l'uso di controlli "if...else", così da essere fornito di risposte diverse in base al tipo di errore ricevuto. Fino a ES2019 il trasferimento dell'eccezione al blocco Catch avveniva sempre e comunque. In tale edizione, invece, fu introdotto l'"optional catch binding" [SG19], ovvero la possibilità di omettere tale passaggio

(torna utile quando si vuole definire un'unica risposta indipendentemente dal tipo di eccezione).

Il Finally, da ultimo, etichetta un blocco che viene sempre eseguito dopo il Try e il Catch, a prescindere dall'avvento di un'eccezione e dalla sua gestione o meno. É importante sapere che se in questo punto del costrutto venisse generato un qualche ritorno, questo sarebbe riportato al flusso principale andando a sostituire un eventuale valore già emesso da uno dei blocchi precedenti [MDN21a].

3.2 Getters/Setters

ECMAScript è un linguaggio "Object oriented"[11] con approccio "Prototype based" : i suoi elementi fondanti sono degli oggetti, contenenti dati statici o dinamicamente calcolati, in cui l'ereditarietà è resa possibile attraverso riferimento al Prototipo [MDN21b]. Quest'ultimo è un Oggetto che può avere un Prototipo a sua volta e che fornisce una stessa parte di dati e funzionalità ai suoi discendenti. La sua peculiarità è che può essere modificato a tempo d'esecuzione: in tal caso il cambiamento si ripercuote su tutti gli elementi che lo posseggono come padre.

Un Oggetto è costituito da una lista non ordinata di proprietà: non solo quelle ereditate, ma anche altre sempre definibili sullo stesso. Queste sono divise in interne, come il riferimento al Prototipo, e in nominali (*Named Data Properties*), che sono quelle interessanti in questa sede.

Una proprietà con nome è una corrispondenza fra una parola chiave, del tipo stringa o simbolo, e un valore che può essere un qualsiasi tipo di dato appartenente al linguaggio[12]; nel caso questo sia una funzione[13] la proprietà viene detta *metodo*. L'accesso al valore può avvenire tramite la "dot-notation" o la "bracket-notation"[14], ma nel caso di un metodo bisogna sempre ricordare di seguire con la invocazione appropriata [MDN21c].

Con l'introduzione dei Getters e Setters (in ES5 del 2009), anche conosciuti come *Named Accessor Properties*, si è aggiunta la possibilità di accedere a una funzionalità interna

[11] Per la precisione è un linguaggio multi-paradigma: procedurale, funzionale e orientato agli oggetti.
[12] Che abbia come Prototipo uno fra undefined, Boolean, Number, String, BigInt, Symbol o Object.
[13] Le funzioni in ECMAScript sono entità di prima classe: sono trattate come variabili regolari.
[14] La dot-notation permette di ottenere il valore di una proprietà tramite `nomeOggetto.nomeProprietà`, `nomeOggetto.nomeProprietà()` nel caso di un metodo, mentre la bracket-notation tramite `nomeOggetto["nomeProprietà"], nomeOggetto["nomeProprietà"]()`, nel caso di un metodo.

all'Oggetto come se fosse un dato statico. Ugualmente ai metodi, essi hanno come "scope"[15] l'Oggetto stesso, quindi possono accedere a tutte le proprietà in esso definite, inclusi altri Getters/Setters. Un'altra cosa interessante è che, a differenza delle "named data", non sono modificabili e enumerabili[16] [PR19]: non vengono considerati nel caso di iterazione sull'Oggetto.

Ad un nome possono essere attribuiti uno o entrambi gli accessori in base al fatto che si voglia, tramite esso, reperire un valore, modificarlo o entrambe le cose. Per leggere la proprietà viene fatto riferimento al Getter associato: una funzione preceduta dalla parola chiave *get*, che non vuole argomenti e restituisce un valore eventualmente in seguito a computazioni. Nel caso di modifica della proprietà, invece, si rimanda al Setter: una funzione preceduta da *set*, che non ha valore di ritorno e si aspetta il passaggio del nuovo valore da impostare attraverso l'operatore di assegnazione " = ".

```
let persona = {
   _nome: "Mario",
   _cognome: "Rossi",
   get nomeCompleto() {
         return ( this._nome + " " + this._cognome );
   },
   set nomeCompleto( data ) {
         if ( data.nome && data.nome.trim().length > 0 )
               this._nome = data.nome;
         else throw  new Error ("Inserire un nome.");
         if ( data.cognome && data.cognome.trim().length > 0 )
               this._cognome = data.cognome;
         else throw new Error ("Inserire un cognome.");
   }
}

persona.nomeCompleto = { nome: "Franco", cognome: "Verdi" };
console.log (persona.nomeCompleto); //stampa "Franco Verdi"
```

Esempio [3.2]: Getters e Setters

[15] Per approfondimenti sull'argomento si veda Appendice A: sezione 1 - *Il contesto di esecuzione*.
[16] L'enumerabilità e la scrivibilità sono degli attributi di una proprietà il cui valore di default può essere eventualmente cambiato [EI09].

Gli *Accessors*, dunque, sono un connubio fra i due tipi di proprietà loro preesistenti e aggiungono un livello di astrazione all'Oggetto nascondendone la struttura interna: permettono di compiere computazioni senza renderlo evidente a livello di scrittura di codice e danno la possibilità di mantenere dei dati privati[17]. Potrebbe infatti accadere di volere nascondere un dato all'esterno rendendolo accessibile e modificabile solo previo calcolo o validazione. Fino ad oggi i programmatori hanno assunto la convenzione di aggiungere l'underscore come suffisso ai nomi di proprietà che si vogliono mantenere private [IK20]. Tuttavia è in discussione una proposta[18], che potrebbe far parte di *ESNext* (acronimo usato per indicare la prossima versione di ECMAScript [W21a]), volta a standardizzare la differenza fra "named properties" pubbliche e non, fornendo quest'ultime di un nome che inizi con un hashtag.

3.3 Modules

Questa feature arricchisce ECMAScript per la prima volta, nel 2015 con ES2015, di un sistema di Moduli "built-in". Fino ad allora per scrivere codice modulare si faceva ricorso a librerie di terze parti[19], ma volendo usare solo ciò che lo standard forniva, il programmatore si ritrovava a dover gestire un'applicazione monolitica su un unico grande file. L'introduzione dei Moduli ha permesso di ottenere un programma frammentato: il risultato è dato dalla composizione di più parti funzionali tra loro disaccoppiate. Un Modulo, infatti, è un pezzo di codice riusabile che incapsula i dettagli implementativi di una certa parte dello script [JI20]. Sono diversi i vantaggi nell'uso di questo nuovo approccio:

- La strutturazione: un Modulo è deputato all'assolvimento di una certa funzionalità, il codice va quindi a dividersi per parti logiche;
- L'astrazione: ci possono essere Moduli deputati allo sviluppo di codice di basso livello, che vengono importati in altri per essere utilizzati senza interessarsi della implementazione;

[17] Esistono anche altri modi per farlo, ad esempio attraverso l'utilizzo di IIFE; qui ho considerato l'uso di un Oggetto.
[18] "*Private methods and getter/setters for JavaScript classes"* di Daniel Ehrenberg [DE21] .
[19] I più famosi sono i "Module-loader" basati sui formati AMD (Asynchronous Module definition) e CommonJs. [AO12]

- La riusabilità: un Modulo può essere importato più volte, ove se ne abbia bisogno, ed è un singleton[20];
- La manutenibilità: è più facile mantenere e modificare una parte rispetto a un tutto;
- La leggibilità: non si ha bisogno di riempire il codice di commenti per trovare una certa sezione;
- La propensione al teamworking: avere delle parti distinte permette una divisione del lavoro più facile e immediata.

In JavaScript ogni Modulo è esattamente un file con un proprio scope privato di default, quindi le dichiarazioni di variabile presenti al suo interno non vanno ad inquinare il "namespace" globale. Nel momento del parsing del codice, ovvero di lettura, le dichiarazioni di Import subiscono il fenomeno di *hoisting*[21]: sono disponibili prima della loro dichiarazione. Perchè ciò sia possibile c'è bisogno che gli Import e gli Export siano a "top-level", cioè non annidati [JI20]. Tutto questo è necessitato dal fatto che la gestione dei Moduli è statica e le eventuali dipendenze vengono caricate in maniera sincrona prima dell'esecuzione del codice. Una volta individuati i Moduli necessari, essi vengono scaricati asincronicamente dal server, sottoposti a parsing e gli Import vengono collegati ai relativi Export.

Tutto ciò che viene esposto all'esterno da un Modulo è la sua *PublicAPI*, che può essere consumata importandone uno o più valori da essa resi disponibili. ECMAScript distingue due tipologie di esportazione, quella di default e per nome[22], ed avviene tramite l'uso della parola chiave *export* seguita da ciò che si vuole disporre. L'importazione prevede invece la voce *import*, un nome[23] e un percorso per trovare il file di riferimento. Un caso particolare di Import è quello seguito solamente da un "path": esso induce l'esecuzione del Modulo specificato senza provocarne alcuna inclusione [MDN21d], [MDN21e].
Esistono diverse sintassi utilizzabili in base a cosa si voglia fornire o acquisire[24] ed è inclusa la possibilità di rinominare ciò che viene importato o esportato : evitando conflitti di nomi, nel primo caso, mantenendo una differenziazione tra ciò che è locale e ciò che viene esposto, nel secondo.

[20] Esiste un'unica istanza del Modulo in tutto il programma, importarlo non ne crea una copia, ma una referenza.
[21] Per approfondimenti sull'argomento si veda Appendice A: sezione 1 - *Il contesto di esecuzione*.
[22] L'esportazione di default è unica nel file, mentre quelle con nome possono essere multiple.
[23] Fa in modo che il contenuto importato possa essere trattato come una variabile locale, ma comunque non modificabile.
[24] Un Modulo fondamentalmente è un pezzo di codice riusabile che incapsula i dettagli implementativi di una certa parte del codice.

```
export default function() { ... }
import myFun from " //path al Modulo ";

export { default, nome1, nome2 };
import nomeDefault, * as altroNome from " //path al Modulo ";

export { nome1, nome2 as nome3, nome4 };
import {nome1 as nome5} from " //path al Modulo ";
import * from " //path al Modulo ";
```

Esempio [3.3]: Uso di diverse sintassi per Import / Export

3.4 Promises

Le Promesse, introdotte in ES2015 del 2015, sono la più grande innovazione per la gestione dell'asincronia[25]. In particolare esse permettono di risolvere il problema del *Callback-Hell* [AN20].

Una Callback è una funzione passata come parametro di coda a un'altra funzione asincrona la cui computazione si può prolungare per un tempo arbitrario: solo una volta conclusa l'esecuzione del codice chiamante allora sarà il suo turno. Questo stile di programmazione è detto *Continuation Passing Style* (o CPS) proprio perché viene passata la continuazione ad una funzione in modo esplicito, come detto, tramite parametro[26]. Il problema è che ogni volta che si esegue una Callback il controllo del flusso del programma continua al suo interno e, di conseguenza, in caso di Callback con dentro altre Callback si finisce nel sopra citato Callback-Hell[27]. Questo inferno di chiamate consecutive si traduce volta per volta in un nuovo blocco annidato di gestione dati e casi di errore. Con le Promesse si ottiene una conduzione delle catene di chiamate più lineare [ZL17].

[25] Per approfondimenti sull'argomento si veda Appendice A: sezione 3 - *L'asincronia*.
[26] Hanno un ambiente di variabili distinto da quello del chiamante, ne aspettano e ricevono il valore di ritorno come parametro.
[27] Graficamente caratterizzato da una forma piramidale, ruotata di 90 gradi verso sinistra, data dai ripetuti annidamenti di funzione in funzione.

```
//Callbacks
setTimeout(() => {
  console.log( "1 secondo");
  setTimeout(() => {
    console.log( "2 secondi" );
    setTimeout(() => {
      console.log( "3 secondi" );
     }, 1000);
  }, 1000);
}, 1000);

//Promises
const asyncFun = function( secondi ) {
  return new Promise( function( resolve ) {
    setTimeout( resolve, secondi * 1000 );
  });
};
asyncFun( 1 )
  .then(() => {
    console.log( "1 secondo" );
    return asyncFun( 1 );
  })
  .then(() => {
    console.log( "2 secondi" );
    return asyncFun( 1 );
  })
  .then(() => {
    console.log( "3 secondi" );
  });
```

Esempio [3.4]: Conversione di Callbacks in Promises

Una Promessa è un Oggetto speciale che viene usato come segnaposto per un valore che si promette esisterà in seguito a una qualche operazione asincrona. Essa cambia stato nel tempo: prima che qualsiasi valore sia disponibile è "pending" poi, quando esso viene restituito, diventa "settled" e, in base al fatto che si sia verificato un errore o meno, si considera "fulfilled" o "rejected". Si può reagire a questi cambi di stato al fine di consumare la

Promessa: la funzione che la crea la restituisce, una volta soddisfatta con successo o fallimento, nel punto del codice in cui essa è attesa dove è poi possibile utilizzarla attraverso i suoi metodi.

Per la gestione dello stato di riuscita esiste il metodo "then()" [MDN21f], proprio di ogni Promessa, in cui viene passata la Callback da eseguire. Per lo stato di errore invece c'è il metodo "catch()". Il Then può essere concatenato a un altro Then che amministra la Promessa restituita da quello sovrastante [SP16]. É questo il mezzo con cui viene risolto il Callback-Hell, infatti non c'è bisogno di annidare una Promessa nell'altra in quanto è possibile restituirla e gestirla sempre allo stesso livello della precedente, ottenendo quindi un codice visivamente verticalizzato.

Un altro importante miglioramento è che la gestione delle eccezioni può essere fatta in un unico punto finale: nel caso si verificasse un errore, questo verrebbe propagato in giù, lungo la catena, e il controllo passerebbe al Catch finale, similmente a quanto accade nel caso del costrutto Try precedentemente visto. In relazione a questo, esiste anche un altro metodo opzionale a disposizione su una Promessa: il Finally[28]. Il suo scopo è di poter rendere eseguibile del codice a prescindere dall'esito della Promessa stessa; un esempio di caso d'uso comune è quello di nascondere un messaggio di caricamento.

Per concludere, esistono altri due metodi statici sul costruttore delle Promesse: "resolve()" e "reject()". Essi permettono di risolvere o rifiutare una Promessa immediatamente e metterla a disposizione di Then o Catch.

3.5 Map

Map è una nuova struttura dati, introdotta con ES2015 nel 2015, che permette di associare delle chiavi a dei valori. La cosa più rivoluzionaria è la possibilità di avere come chiave un qualsiasi tipo di dato, quando per un Oggetto non è così [MS18].

Una Map al suo interno è un Array di Array, dove il primo indicizza e mantiene gli elementi nell'ordine di inserimento, mentre quelli contenuti rappresentano le corrispondenze chiave-valore.

Per creare una Map il modo più semplice è formarla vuota, tramite l'operatore "new", e poi aggiungere gli elementi singolarmente. Per questa operazione è messo a disposizione il metodo "set()" che prende come argomenti la nuova chiave e il valore associato e restituisce

[28] Concatenabile a entrambi Then e Catch.

la struttura dati aggiornata, di conseguenza è possibile concatenare un nuovo inserimento al precedente.
Un altro modo di generare una Map è passando come argomento al costruttore l'Array desiderato; sicuramente più macchinoso e pesante a livello di scrittura, ma che produce lo stesso risultato.
Esistono altre funzionalità utili per lavorare con una Map [MDN21g], [GG19]:

- Clear: pulisce la Map rimuovendo tutte le coppie contenute;
- Delete : permette di rimuovere un singolo elemento passandone la chiave;
- Get: restituisce una corrispondenza tramite la chiave;
- Has : restituisce un booleano in base alla presenza della chiave nella struttura.

```
let mappa =  new Map();
mappa.set( 1, "uno" ).set( 2, "due" ).set( 3, "tre");
console.log( mappa.get( 1 ) ); // stampa "uno"
mappa.delete ( 2 );
console.log( mappa.has( 2 ) ); // stampa "false"
mappa.clear();
console.log( mappa.size ); // stampa "0"
```

Esempio [3.5]: Uso di Map

Un'ulteriore importante differenza con gli Oggetti è che Map è un iterabile [MDN21h]: predispone gli strumenti per ciclare su se stessa[29] sia attraverso un *forEach*, implementato sul suo Prototipo, che usando un loop "for...of". Per iterare su un Oggetto si è sempre dovuto farlo manualmente sino all'inserimento di un nuovo metodo in ES2017: "Object.entries()" [EI17]. Questo fondamentalmente converte un Oggetto nella struttura dati di una Map per renderlo iterabile.

3.6 Spread operator

Questo inserimento nello standard, in ES15 del 2015, fu subito accolto con entusiasmo dalla comunità degli sviluppatori perché utile e compatto. Si tratta dell'operatore " … ", che permette di spacchettare un iterabile nei suoi singoli elementi.

[29] In base alle entrate, alle chiavi o ai valori; esistono i metodi `entries(), keys(), values()` appositamente.

L’utilizzo più frequente che ne viene fatto è con gli Array: rende possibile l'aggiunta di elementi prima e dopo di essi, il crearne una copia o unirli fra loro senza dover ricorrere all’impiego dei metodi relativi.

```
let arr1 = new Array(3, 4); // arr1 vale [3,4]
let arr2 = [1, 2, ...arr1, [5]];
let arr3 = [...arr1];
arr1 = [...arr2, ...arr1];
// qui arr2 vale [1, 2, 3 , 4, [5]], arr3 vale
// [3, 4] e arr1 vale [1, 2, 3, 4, [5], 3, 4]
```

Esempio [3.6]: Manipolazione di Array tramite Spread operator

Un’altro caso d’uso interessante, oltre a quelli appena citati, è l’impiego dello Spread al posto del metodo "apply()" di Funzione per il passaggio di più argomenti quando essi sono contenuti in un Array [MDN21i]; in entrambi i casi, nell’eventualità ne vengano passati in maggior numero rispetto a quelli attesi, lo scarto viene ignorato.

In ECMAScript2018 è stata estesa la possibilità di utilizzo di questa nuova feature anche agli oggetti[30]: tramite l’operatore Spread è possibile ora clonarli o unirli tra loro e nel caso di chiavi ripetute viene mantenuto l’ultimo valore incontrato per le stesse.
É bene notare che le copie create, tramite questo operatore, non risultano in elementi del tutto distinti: per gli item strutturati viene copiato solo l’indirizzo di riferimento[31].

3.7 Async/Await

Async/Await, introdotti con ES2017 nel 2017, arricchiscono il linguaggio di una modalità più intuitiva ed elegante per la gestione delle *Promises* [AN20].
In una funzione etichettata dalla parola chiave *async*, ci possono essere una o più istruzioni caratterizzate dalla voce *await* seguita da una Promessa. L’ Await serve per bloccare l’esecuzione del codice della funzione in quel punto, in attesa che venga restituito un valore di risoluzione. Questo non comporta problemi perchè tale tipo di funzione è processata asincronicamente sullo sfondo e il flusso principale non viene quindi bloccato[32].

[30] Vengono ignorate le proprietà non enumerabili.
[31] Per approfondimenti sull’argomento si veda Appendice A: sezione 2.1 - *La gestione della memoria*.
[32] Per approfondimenti sull’argomento si veda Appendice A: sezione 3 - *L’asincronia*.

```
const asyncFun = function() {
  return new Promise( function( resolve ) {
    setTimeout( resolve ( "secondo" ) );
  });
};
const myFun = async function() {
   let risultato = await asyncFun();
   console.log(1 + " " + risultato ); // Stampa "1 secondo"
}
myFun();
```

Esempio [3.7]: Uso di Async/Await

Far apparire il codice asincrono come se fosse sincrono è l'innovazione principale di Async/Await: non serve più una concatenazione di Then e Callback in quanto l'attesa è implicita e nascosta. Inoltre, data questa sincronia simulata, per la gestione delle eccezioni è possibile usare il Try/Catch [AC20b] .
Quando una Promessa attesa restituisce un valore, l'intera istruzione relativa viene sostituita con esso introducendo così, per la prima volta, la possibilità di assegnarlo a una variabile.

Infine è da rendere presente che una funzione contrassegnata asincrona ha sempre una Promessa come valore di ritorno, di conseguenza se lo si vuole gestire nel chiamante è necessario utilizzare il metodo Then o un Await a sua volta[33].

3.8 Flat()

Flat è un metodo che è stato aggiunto, nel 2018 con ES2018, al Prototipo degli Array. Esso permette di appiattire una collezione che presenta diversi livelli di annidamento al suo interno, dati da altri Array, quante volte si voglia. A tal proposito questa funzionalità si aspetta di ricevere come parametro un numero indicante quanti gradi di spacchettamento[34] si vogliano eseguire; nel caso non venga passato nulla viene considerato l'uno come default, al contrario se viene passato " + Infinity "[35] allora tutti gli annidamenti vengono sciolti e si ottiene un Array lineare [MDN21n]. Ad ogni modo si otterrà sempre almeno un Array dove gli elementi vuoti interni vengono rimossi.

[33]Caso in cui anche la funzione chiamante sia etichettata async.
[34] La rimozione dei livelli di annidamento procede da quelli più esterni a quelli più interni.
[35] `Infinity` è una variabile di sola lettura, nello scope globale, che rappresenta il concetto di infinito numerico.

```
let arr = [ [ 1, [ [] , 2, , 3], [ 4 ], 5 ] ];
arr = arr.flat( 3 );
console.log( arr ); // stampa [ [ 1, 2, 3, 4, 5 ] ]
```

Esempio [3.8]: Uso di Flat()

È una feature interessante perché permette di fare qualcosa che prima al meglio era ottenibile per mezzo di una chiamata ricorsiva su una combinazione di metodi: Reduce e Concat. In tal caso il primo mantiene i valori da restituire e l'elemento da processare nella data iterazione, se questo risulta essere un Array si procede con un'altra ricorsione altrimenti lo si aggiunge ai valori di ritorno tramite Concat [MDN21n]. Con l'utilizzo di Flat si rendono le cose evidentemente più semplici senza il bisogno di creare una funzione di supporto.

Questo metodo, data la sua capacità, può anche essere usato in alternativa allo Spread operator per unire tra loro più Array: si costituisce un unico Array contenitore e vi si applica Flat. Si noti però che lo Spread non è in grado di gestire l'annidamento, o meglio, dove si presenta lo mantiene: se si vuole ottenere un Array lineare a partire da collezioni poste su più livelli è sicuramente più agibile l'utilizzo della feature qui introdotta.

Da ultimo, si vuole almeno citare il metodo pubblicato in coppia con quello appena presentato: "flatMap()" [MDN21j]. Come suggerisce il nome, è una combinazione tra "flat()" e "map()" per cui su ogni elemento è possibile compiere una certa computazione e lo spacchettamento di un livello.

3.9 Optional chaining

Una feature nuovissima (ES2020 del 2020) e già molto usata è l'operatore " .? " di Optional chaining. Questo permette di raggiungere una proprietà di un Oggetto anche particolarmente annidata, senza bisogno di preoccuparsi di validare ogni passo della catena. Se viene incontrato un elemento che non ha valore allora viene restituito "undefined"[36] immediatamente, prima di ricadere nell'errore di lettura su qualcosa di non definito.
La cosa importante è che il primo elemento sia un Oggetto che si sappia essere definito:
l'Optional chaining controlla che l'item alla sua sinistra sia valido e solo in quel caso procede nella lettura. Viene per esso introdotto il concetto dei valori *Nullish* [MDN21k]:

[36] Undefined è un tipo primitivo il cui unico valore è `undefined`; viene usato quando una una variabile non possiede un valore assegnato.

una proprietà è inaccessibile solo se risulta "undefined" o "null". La stringa vuota e lo zero sono pertanto valori accettabili anche se fanno parte dei valori *Falsy*[37]. Se si arriva all'obiettivo e anch'esso risulta valido, allora viene restituito automaticamente.

```
let obj = { a : { b: { c: 0 }}}
let toPrint = null;
// controllo con if
if ( obj.a.b.c ) toPrint = obj.a.b.c ;
else toPrint = false;
console.log (toPrint); // stampa false
// controllo con ternary operator
obj.a.b.c
? toPrint = obj.a.b.c
: toPrint = false;
console.log (toPrint);   // stampa false
// controllo con optional chaining
toPrint = obj?.a?.b?.c ?? false;
console.log (toPrint);   // stampa 0
```

Esempio [3.9]: Controlli sul valore zero.

L'operatore che nell'esempio appena riportato viene usato in combinazione con la feature introdotta è il *Nullish coalescing*. Esso è stato inserito nello standard per poter assegnare un valore di default a una valutazione che risultasse Nullish. In ES, infatti, il classico operatore logico " Or || " considera invalidi anche i valori Falsy poiché ne controlla la veridicità, di conseguenza non può essere utilizzato in concatenamento all'Optional chaining mantenendone la semantica [AC20a].

[37] Sono dei valori che vengono valutati come false da un controllo condizionale, tra di essi ricadono: false, lo zero, le stringhe vuote, null, undefined, NaN.

Capitolo 4

Appresa la funzionalità e le caratteristiche di ogni feature in esame, l'impegno è stato rivolto alla ricerca di messe in pratica che permettessero di raggrupparle per mansione e usufruirne in parallelo nella risoluzione degli stessi problemi.
In questo capitolo viene presentato il lavoro svolto, anticipato da una breve premessa in merito alle tecnologie utilizzate.

Il progetto

4.1 Le tecnologie

Vengono qui elencate le tecnologie impiegate nella realizzazione del progetto, seguite da una specificazione se meno note:

- "HTML" come linguaggio di Markup;
- "CSS" come linguaggio per gli stili;
- "JavaScript" come linguaggio di programmazione per entrambe le parti client e server;
- "Vue.js" come framework JavaScript front-end: permette di costruire interfacce interattive attraverso la combinazione dinamica di più Componenti, ovvero di contenitori di codice, HTML[38] e CSS riutilizzabili e personalizzabili in base ai dati forniti;
- "Node.js" come ambiente backend: un framework che permette di utilizzare JavaScript anche lato server e che include un proprio gestore di librerie chiamato Node.js packet manager (o npm);
- "MongoDB" come database: un database NoSQL allestito in collezioni di documenti[39].

[38] Utilizza una sintassi basata su HTML arricchita di funzionalità.
[39] Strutture dati in BSON: un formato dati come JSON, quindi di associazioni chiave valore, ma tipato.

4.2 L’idea

Il presupposto è quello di mostrare per ogni feature un’implementazione che la utilizzi e una che in suo luogo presenti un costrutto ad essa preesistente.
I casi in oggetto sono stati suddivisi nei seguenti gruppi logico-evolutivi:

- Callbacks **- Promises - Async/Await**;
- Reduce/Concat **- Spread operator - Flat**;
- If - **Try - Optional chaining**;
- Static values - Methods **- Getters/Setters**;
- Array **- Map**;
- Monolithing application **- Modular application**.

Si è optato per la creazione di un’unica applicazione a carattere puramente dimostrativo[40] e che includesse l’impiego di tutte le funzionalità sopra elencate. É stata eseguita una riproduzione della stessa sia su un unico file che attraverso l’uso dei Moduli, di cui i vantaggi sono già stati discussi.

L’applicativo è sviluppato su due interfacce: una di sola visualizzazione e l’altra di interazione. Entrambe sono state realizzate in più versioni per poter esibire ed eseguire, come detto, i cambiamenti in parallelo.
L’idea generale consiste nell’avere una collezione di articoli di vestiario, ad esempio presenti in un magazzino, da reperire e mostrare a schermo e sui quali sono resi disponibili meccanismi di filtraggio in base a delle proprietà; è concesso inoltre l'inserimento di ordini che li includano e ne manipolino le quantità disponibili.

[40]Non ha lo scopo di essere utilizzabile nel mondo reale.

4.2.1 L’interfaccia di visualizzazione

Questa interfaccia viene riprodotta in tre versioni che arrivano allo stesso risultato facendo uso di implementazioni procedurali diverse, in base ai costrutti utilizzati:

- Prima versione: Callbacks - Reduce/Concat - If ;
- Seconda versione: Promises - Spread operator - Try;
- Terza versione: Async/Await - Flat - Optional chaining.

Essa è essenziale: presenta un bottone per poter richiedere i dati al server e delle selezioni per poter filtrare i dati in base a caratteristiche proprie di ogni Oggetto. Un singolo articolo viene visualizzato entro un contenitore che ne riporta la collezione di appartenenza, data da stagione e categoria, il codice identificativo e un’immagine.

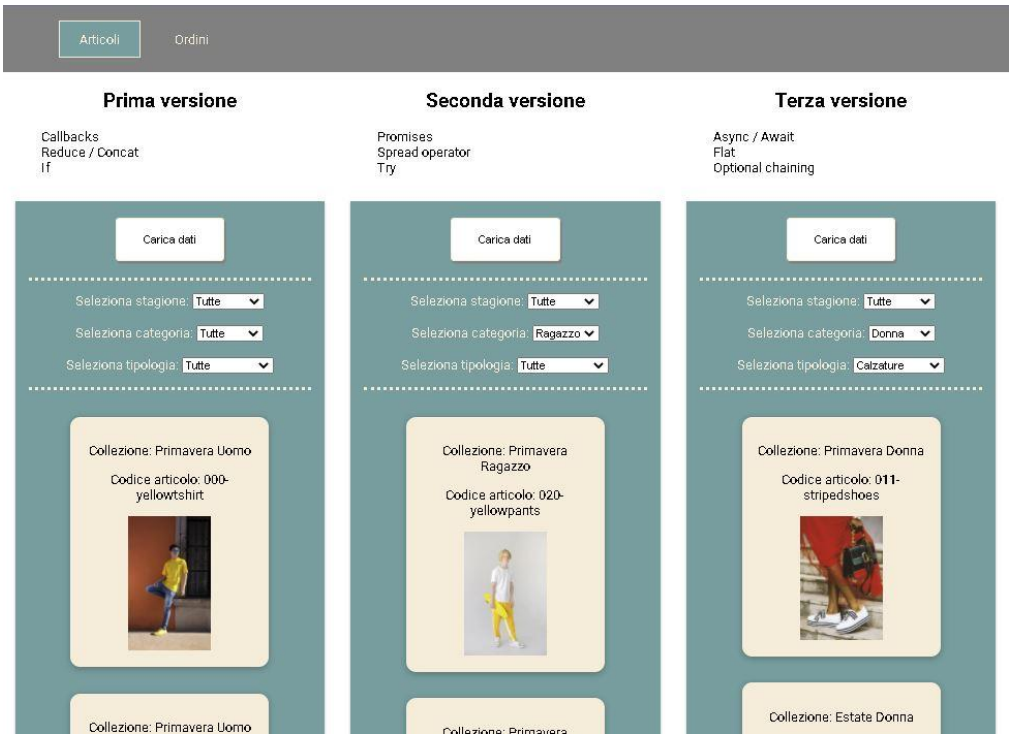

Immagine [4.2.1]: Interfaccia di visualizzazione

4.2.2 L'interfaccia di interazione

Questa interfaccia è stata riadattata in quattro versioni: le prime tre permettono il confronto tra i diversi modi di interazione con le proprietà degli oggetti, la terza è stata poi riprodotta in una quarta per convertire l'utilizzo di Array in Map.

In ogni implementazione viene caricata la stessa serie di ordini in attesa a cui possono esserne aggiunti dei nuovi tramite un form. Ogni ordine è dato da un insieme non vuoto di articoli per cui viene specificata la quantità richiesta e se essa sia da considerarsi in entrata o meno. Inseriti gli elementi desiderati è possibile aggiungere l'ordine nella lista di attesa. L'ordine meno recente può essere poi dichiarato completato: la disponibilità della merce in esso inclusa subisce effettivamente una variazione.

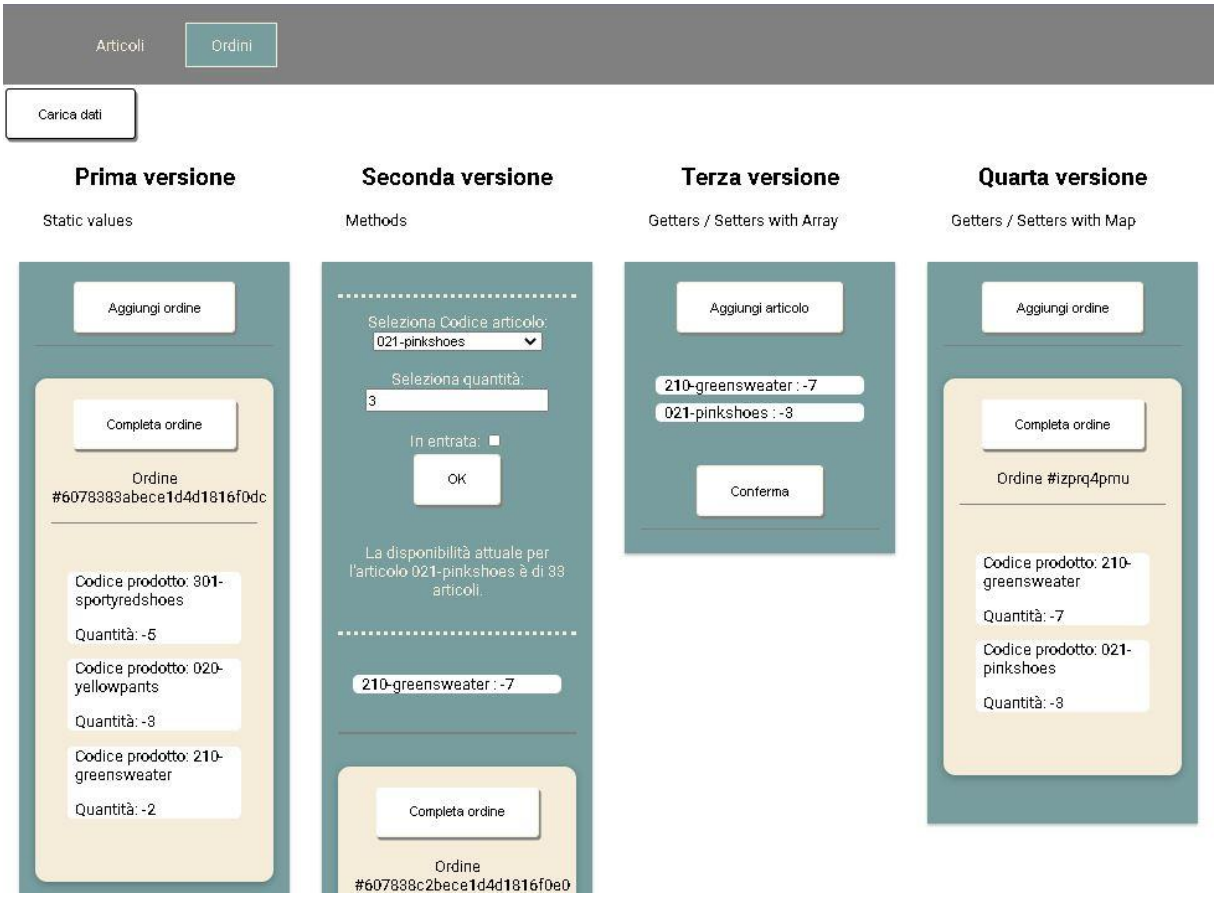

Immagine [4.2.2]: Interfaccia di interazione

4.3 Le implementazioni

In questo paragrafo viene esplicato ogni gruppo di feature a livello implementativo: vengono illustrate le specifiche assunte per permetterne la messa in opera, il loro ruolo e il codice relativo.
I primi tre sottoparagrafi si riferiscono alla parte di visualizzazione mentre gli ultimi due a quella di interazione.

4.3.1 Callbacks - Promises - Async/Await

Innanzi tutto l'obiettivo prefisso è stato quello di ottenere un esempio di Callback-Hell. È stato supposto di doversi interfacciare ad una *API*[41] che mettesse a disposizione solo un percorso[42] per il reperimento di gruppi di articoli in base a due parametri, la stagione e la categoria (una tra "men, women, youngs, children"), e che li restituisse in un Array includente a sua volta una prima parte relativa al vestiario e una seconda alle calzature[43]. Ogni stagione inoltre si è assunto imposto da specifica che dovesse consistere di un'organizzazione di dati ordinata per categoria e tale concetto è stato il punto chiave sfruttato per ricavare il caso voluto.
La struttura dati finale, decisa a fini implementativi, è quindi la seguente (i tre punti di sospensione sottendono la presenza di ulteriori dati e/o annidamenti interni).:

```
articles: {
    "spring": [
          [[...], [...]], // categoria uomini
          [[...], [...]], // categoria donne
          [[...], [...]], // categoria ragazzi
          [[...], [...]]  // categoria bambini
],
"summer" : [...], // stessa struttura di "spring"
"autumn" : [...], // stessa struttura di "spring"
"winter" : [...]  // stessa struttura di "spring"       }.
```

Esempio [4.3.1]: Struttura dell'Oggetto "articles"

[41] Application Programming Interface: l'insieme dei pezzi di codice che, in questo caso, un server dispone all'esterno in modo da definire le modadità di comunicazione e lo scambio di informazioni da lui concesse.
[42] Per l'interfaccia ordini sono stati creati path più comodi per permettere l'ottenimento di tutti i dati in una volta, siccome lì non necessaria la creazione di un Callback-Hell.
[43] Per ottenere degli Array di Array utili all'applicazione del gruppo di feature successiva.

Come sotteso nelle righe precedenti, questo gruppo di feature è stato adoperato nella comunicazione asincrona con il server per ottenere i dati da database.
Segue il codice relativo ad ognuna.

```
xhrRequest(url, callback) {
      const request = new XMLHttpRequest();
      request.addEventListener('readystatechange', () => {
        if (request.readyState === 4 && request.status === 200) {
          const data = JSON.parse(request.responseText);
          callback(undefined, data);
        } else if (request.readyState === 4) {
          callback('Could not fetch data.', undefined);
        }
      });
      request.open('GET', url, true);
      request.send(null);
    },
    getDataFromAPI() {
      this.resetData(); // metodo di supporto per inizializzare dati.
      let temporaryObj = {};
      ['spring', 'summer', 'autumn', 'winter'].forEach((season) => {
        temporaryObj[season] = [];
        this.xhrRequest(
          `${this.baseURL}/articles/${season}/men`,
          (err, res) => {
                if (err) {
                  console.error(err);
                  this.loadingMsg = 'Errore nel reperire i dati.’;
                }
            temporaryObj[season].push(res.data.articles);
            this.xhrRequest(
              `${this.baseURL}/articles/${season}/women`,
              (err, res) => {
                if (err) {
                  console.error(err);
                        this.loadingMsg = 'Errore nel reperire i dati.';
                }
```

```
            temporaryObj[season].push(res.data.articles);
            this.xhrRequest(
              `${this.baseURL}/articles/${season}/youngs`,
              (err, res) => {
                if (err) {
                 console.error(err);
                 this.loadingMsg = 'Errore nel reperire i dati.';
                }
                temporaryObj[season].push(res.data.articles);
                this.xhrRequest(
                  `${this.baseURL}/articles/${season}/children`,
                  (err, res) => {
                    if (err) {
                      console.error(err);
                      this.loadingMsg =
                        'Errore nel reperire i dati.';
                    }
                    temporaryObj[season].push(res.data.articles);
                    this.callCounter++;
                    if (this.callCounter === 4) {
                      this.articles = temporaryObj;
                      this.loadingMsg = '';
                      this.dataAreLoaded = true;
                    }
                  }
                );
              }
            );
          }
        );
      }
    );
  });
}.
```

Codice [4.3.1|A]: Implementazione con Callbacks

```
xhrRequest(url) {
      return new Promise((resolve, reject) => {
        const request = new XMLHttpRequest();
        request.addEventListener('readystatechange',() => {
          if (request.readyState === 4 && request.status === 200) {
            const data = JSON.parse(request.responseText);
            resolve(data);
          } else if (request.readyState === 4) {
            reject('Could not fetch data.');
          }
        });
        request.open('GET', url, true);
        request.send(null);
      });
    },
   getDataFromAPI() {
      this.resetData(); // metodo di supporto per inizializzare dati.
      let temporaryObj = {};
      ['spring', 'summer', 'autumn', 'winter'].forEach((season) => {
        temporaryObj[season] = [];
        this.xhrRequest(`${this.baseURL}/articles/${season}/men`)
          .then((res) => {
            temporaryObj[season].push(res.data.articles);
             return this.xhrRequest(`${this.baseURL}/articles/${season}
                  /women`);
          })
          .then((res) => {
            temporaryObj[season].push(res.data.articles);
             return this.xhrRequest(`${this.baseURL}/articles/${season}
                  /youngs`);
          })
          .then((res) => {
            temporaryObj[season].push(res.data.articles);
             return this.xhrRequest(`${this.baseURL}/articles/${season}
                  /children`);
          })
          .then((res) => {
            temporaryObj[season].push(res.data.articles);
```

```
            this.callCounter++;
            if (this.callCounter === 4) {
              this.articles = temporaryObj;
              this.loadingMsg = '';
              this.dataAreLoaded = true;
            }
          })
          .catch((err) => {
            console.error(err);
            this.loadingMsg = 'Errore nel reperire i dati.';
          });
      });
    }.
```

Codice [4.3.1|B]: Implementazione con Promises

```
xhrRequest(url) {
   return new Promise((resolve, reject) => {
     const request = new XMLHttpRequest();
     request.addEventListener('readystatechange', () => {
       if (request.readyState === 4 && request.status === 200) {
         const data = JSON.parse(request.responseText);
         resolve(data);
       } else if (request.readyState === 4) {
         reject('error getting articles');
       }
     });
     request.open('GET', url, true);
     request.send(null);
   });
 },
 getDataFromAPI() {
   this.resetData();   // metodo di supporto per inizializzare dati.
   let temporaryObj = {};
   ['spring', 'summer', 'autumn', 'winter'].forEach(async (season) => {
     try {
       temporaryObj[season] = [];
      temporaryObj[season].push((await this.xhrRequest(`${this.baseURL}/
          articles/${season}/men`)).data.articles);
```

```
        temporaryObj[season].push((await this.xhrRequest(`${this.baseURL}/
            articles/${season}/women`)).data.articles);
        temporaryObj[season].push((await this.xhrRequest(`${this.baseURL}/
            articles/${season}/youngs`)).data.articles);
        temporaryObj[season].push((await this.xhrRequest(`${this.baseURL}/
            articles/${season}/children`))data.articles);
       this.callCounter++;
       if (this.callCounter === 4) {
          this.articles = temporaryObj;
          this.loadingMsg = '';
          this.dataAreLoaded = true;
       }
     } catch (err) {
        console.error(err);
        this.loadingMsg = 'Errore nel reperire i dati.';
     }
   });
}.
```

Codice [4.3.1|C]: Implementazione con Async/Await

La funzione *xhrRequest* è quella contenente il compito asincrono: la gestione effettiva della comunicazione con il server tramite l'Oggetto *XMLHttpRequest*, come accennato in *Appendice A (sezione tre)*. Nel primo caso vengono restituiti i dati reperiti, negli altri due una Promessa che viene gestita nelle rispettive modalità già presentate.
Si può notare a vista d'occhio di come si sia ottenuto un codice via via più fluido e comprensibile: si veda nella prima versione la struttura piramidale precedentemente citata, che si verticalizza e poi semplifica ulteriormente.

4.3.2 Reduce/Concat - Spread operator - Flat

Come anticipato, è stata creata una struttura dati di Array di Array proprio per far al caso di questo gruppo di feature.
In particolare esse sono state utilizzate per fornire dinamicamente l'elenco di articoli da visualizzare in base alle selezioni eseguite dall'utente. Nella pratica è stata creata una *Computed Property*: una tipologia di metodi resa disponibile sugli oggetti di Vue.js che

restituisce un valore, dipendente da altri dati e condizioni mutabili nel tempo, che viene automaticamente ricalcolato ad ogni cambiamento che lo intacchi.

```
flattenArray(myArr) {
      return myArr.reduce((accumulator, element) => {
        if (Array.isArray(element))
          return accumulator.concat(this.flattenArray(element));
        return accumulator.concat(element);
      }, []);
    },
articlesToRender() {
      if (!this.dataAreLoaded) return [];
      let arrayToReturn = [];
      if (this.selectedSeason === 'all') {
        ['spring', 'summer', 'autumn', 'winter'].forEach((season) =>{
          this.selectedCategory === 'all'
            ? arrayToReturn.push(this.articles[season])
            :arrayToReturn.push(this.articles[season][this.
                    categoryIndexes[this.selectedCategory]]);
        });
      } else {
        this.selectedCategory === 'all'
          ? arrayToReturn.push(this.articles[this.selectedSeason])
          : arrayToReturn.push(this.articles[this.selectedSeason][
                this.categoryIndexes[this.selectedCategory]]);
      }
      arrayToReturn = this.flattenArray(arrayToReturn);
      this.selectedTypology !== 'all'
        ? (arrayToReturn = arrayToReturn.filter(
            (el) => el.typology === this.selectedTypology
          ))
        : null;
      return arrayToReturn;
    }.
```

Codice [4.3.2|A]: Implementazione con Reduce/Concat

```
articlesToRender() {
      if (!this.dataAreLoaded) return [];
      let arrayToReturn = [];
      if (this.selectedSeason === 'all') {
        ['spring', 'summer', 'autumn', 'winter'].forEach((season) => {
          this.selectedCategory === 'all'
           ? this.articles[season].forEach((category) => {
                arrayToReturn = [...arrayToReturn, ...category];
              })
           : (arrayToReturn = [...arrayToReturn, ...this.articles
                [season][this.categoryIndexes[this.selectedCategory]] ]);
        });
      } else {
        this.selectedCategory === 'all'
          ? this.articles[this.selectedSeason].forEach((category) => {
              arrayToReturn = [...arrayToReturn, ...category];
            })
          :(arrayToReturn= [...arrayToReturn,...this.articles[this.
          selectedSeason][this.categoryIndexes[this.selectedCategory]] ]);
      }
      this.selectedTypology !== 'all'
        ? (arrayToReturn = arrayToReturn.filter(
            (el) => el.typology === this.selectedTypology
          ))
        : null;
      return arrayToReturn;
    }.
```

Codice [4.3.2|B]: Implementazione con Spread operator

```
 articlesToRender() {
      if (!this.dataAreLoaded) return [];
      let arrayToReturn = [];
      if (this.selectedSeason === 'all') {
        ['spring', 'summer', 'autumn', 'winter'].forEach((season) => {
          this.selectedCategory === 'all'
            ? arrayToReturn.push(this.articles[season].flat(+Infinity))
```

```
        : arrayToReturn.push(this.articles[season][this.
            categoryIndexes[this.selectedCategory]].flat(+Infinity));
    });
  } else {
    this.selectedCategory === 'all'
      ? arrayToReturn.push( this.articles[this.selectedSeason]
            .flat(+Infinity))
      : arrayToReturn.push( this.articles[this.selectedSeason][this.
            categoryIndexes[this.selectedCategory].flat(+Infinity));
  }
  arrayToReturn = arrayToReturn.flat(+Infinity);
  this.selectedTypology !== 'all'
    ? (arrayToReturn = arrayToReturn.filter(
        (el) => el.typology === this.selectedTypology
      ))
    : null;
  return arrayToReturn;   }.
```

Codice [4.3.2|C]: Implementazione con Flat()

Nella prima versione si è dovuto creare una funzione di supporto che impiega, appunto, i metodi Reduce e Concat. Nella seconda è stato necessario specificare ogni Array singolarmente, in quanto lo Spread operator non gestisce gli annidamenti come anticipato, ottenendo così un codice meno intuitivo. L'ultima implementazione invece risulta la più chiara e di veloce stesura.

4.3.3 If - Try - Optional Chaining

Per inserire l'utilizzo di questi controlli separatamente dal resto, è stato realizzato lo scenario per cui non tutti gli articoli siano dotati interamente o in parte della proprietà *image*. Ogni immagine è un Oggetto composto dal campo *src*, l'URL dell'immagine[44], e dal campo *alt*, il testo alternativo. Alcune di loro però non posseggono o l'intera proprietà o una/entrambe le contenute. Sono stati così creati due piccoli metodi che prendono come parametro un articolo e ne restituiscono i dati relativi se presenti, dei dati di default altrimenti[45].

[44] Sono state utilizzate immagini copyright-free reperite presso il nome di dominio "*pexels.com"*.
[45] Se presente il testo alternativo ma non l'immagine, viene comunque restituita l'alt di default in quanto sarebbe visualizzata l'immagine predefinita.

```
  getSrc(article) {
    if (article.image) {
      if (article.image.src) {
        return article.image.src;
      }
    }
    return this.defaultSrc;    },

  getAlt(article) {
      if (article.image) {
        if (article.image.src) {
          if (article.image.alt) {
            return article.image.alt;    }
          return article.productCode;
        }
  }
    return this.defaultAlt; }.
```

Codice [4.3.3|A]: Implementazione con If

```
getSrc(article) {
     try {
       if (!article.image.src) throw new Error('Src is undefined');
       return article.image.src;
     } catch {
       return this.defaultSrc;
     }   },

  getAlt(article) {
    try {
      if (!article.image.src) throw new Error('Src is undefined');
      if (!article.image.alt) throw new Error('Alt is undefined');
      return article.image.alt;
    } catch (err) {
      if (err.message === 'Alt is undefined')
        return `${article.productCode}-image`;
      return this.defaultAlt;
    }   }.
```

Codice [4.3.3|B]: Implementazione con Try

```
getSrc(article) {
     return article?.image?.src ?? this.defaultSrc;
   },
   getAlt(article) {
     let toReturn;
     if (article?.image?.src)
       return article?.image?.alt ?? `${article.productCode}-image`;
     else return this.defaultAlt;
   }.
```

Codice [4.3.3|C]: Implementazione con Optional chaining

Nel primo caso viene riportato un esempio di controlli If consecutivi per l'individuazione dei diversi casi di errore, cosa che viene successivamente realizzata nel costrutto Try attraverso lo scatenamento e la gestione di eccezioni. Tuttavia l'Optional chaining risulta a colpo d'occhio la scelta più agevole e pulita.

4.3.4 Static values - Methods - Getters/Setters

Per la feature Getters/Setters si è deciso di realizzare una versione per entrambe le modalità di accesso agli oggetti che l'hanno preceduta e ispirata.
In ogni implementazione gli articoli vengono innanzitutto passati in una funzione costruttore: vengono inserite solo le proprietà importanti per questa interfaccia[46] e aggiunti i metodi o i Getters/Setters nei rispettivi casi.

Nell'ultimo caso per l'introduzione degli Accessors è stato utilizzato il metodo *Object.defineProperties* [MDN21s], inserito nello standard con gli stessi (in ES5 del 2009), che permette di aggiungere o modificare delle proprietà in un Oggetto dopo la sua creazione ed è, in particolare, quello raccomandato per la definizione di Getters/Setters tramite funzione costruttore [MDN21r], [SO15] .

[46] Le proprietà relative all'immagine, la stagione, la categoria e la tipologia di appartenenza, introdotte per la parte di visualizzazione, vengono qui ignorate quindi escluse.

```
beforeMount() {
    let tmpArray = [];
    function ArticlesFirstVersion(article) {
      this.productCode = article.productCode;
      this.counter = article.counter;
      this.ordersCounter = article.ordersCounter;
    }
    this.articles.forEach((el) => {
      this.productCodes.push(el.productCode);
      tmpArray.push(new ArticlesFirstVersion(el));
    });
    this.articlesVersioned = tmpArray;
  }.
```

Codice [4.3.4|A.1]: Implementazione con Static values - costruttore

```
beforeMount() {
    let tmpArray = [];
    function ArticlesSecondVersion(article) {
      this.productCode = article.productCode;
      this.counter = article.counter;
      this.ordersCounter = article.ordersCounter;
      this.availability = function () {
        return this.counter + this.ordersCounter;
      };
      this.setIncomingChanges = function (quantity) {
        this.ordersCounter += quantity;    };
      this.newAvailability = function (quantity) {
        this.counter += quantity;
        this.ordersCounter -= quantity;
      };    }
    this.articles.forEach((el) => {
      this.productCodes.push(el.productCode);
      tmpArray.push(new ArticlesSecondVersion(el));
    });
    this.articlesVersioned = tmpArray;    }.
```

Codice[4.3.4|A.2]: Implementazione con Methods - costruttore

```
beforeMount() {
   let tmpArray = [];
   function ArticlesThirdVersion(article) {
     this.productCode = article.productCode;
     this.counter = article.counter;
     this.ordersCounter = article.ordersCounter;
     Object.defineProperties(this, {
       availability: {
         get: function () {
           return this.counter + this.ordersCounter;
         },
         set: function (quantity) {
           this.counter += quantity;
           this.ordersCounter -= quantity;
         }
       },
       incomingChanges: {
         set: function (quantity) {
           this.ordersCounter += quantity;
         }
       }
     });
   }
   this.articles.forEach((el) => {
     this.productCodes.push(el.productCode);
     tmpArray.push(new ArticlesThirdVersion(el));
   });
   this.articlesVersioned = tmpArray;
 }.
```

Codice [4.3.4|A.3]: Implementazione con Getters/Setters - costruttore

Ogni articolo, in tutte le ricostruzioni, è caratterizzato da due campi statici, *counter* e *ordersCounter*, che ne mantengono la disponibilità reale in magazzino e una somma dipendente dagli ordini in coda che lo includono. Ciascun ordine, infatti, contiene una collezione di oggetti, uno per ogni articolo incluso, composti da una proprietà per il codice articolo e una per la quantità associata.

Di conseguenza, prima dell'inserzione di un elemento in un nuovo ordine, è possibile compiere una validazione sulla quantità richiesta per verificarne l'effettuabilità.

Nei seguenti frammenti di codice viene riportata la prima versione per intero, in cui viene evidenziata la parte invariabile, e solo i cambiamenti per le successive.

```
validateArticle() {
      this.validationError = '';
      let temporaryArticle = this.articlesVersioned.find(
        (el) => el.productCode === this.selectedArticle
      );
      let quantity = Number(
       this.checked ? this.selectedQuantity : '-' + this.selectedQuantity
      );
      if ( this.temporaryOrder.find((el) =>
          el[0] === temporaryArticle ? true : false
        )
      ) {
        this.validationError = `L'articolo ${this.selectedArticle} è già
             presente nell'ordine.`;
        this.selectedArticle = '';
        return null;
      }
      if (quantity >= 0 || temporaryArticle.counter +
             temporaryArticle.ordersCounter + quantity >= 0 ) {
        this.temporaryOrder.push([temporaryArticle, quantity]);
        if (this.orderIsEmpty && this.temporaryOrder.length > 0)
          this.orderIsEmpty = false;
      } else {
        this.validationError = `La disponibilità attuale per l'articolo
          ${this.selectedArticle} è di ${temporaryArticle.counter +
          temporaryArticle} articoli.`;
        this.selectedQuantity = '0';
        return null;    }
      this.resetValidation(); // metodo di supporto per pulizia dati    }.
```

Codice [4.3.4|B.1]: Implementazione con Static values - validazione

```
validateArticle() {
      // [...] parte comune
     if (quantity >= 0 || temporaryArticle.availability() + quantity
         >= 0) {
       this.temporaryOrder.push([temporaryArticle, quantity]);
       if (this.orderIsEmpty && this.temporaryOrder.length > 0)
         this.orderIsEmpty = false;
     } else {
       this.validationError = `La disponibilità attuale per l'articolo
         ${this.selectedArticle} è di ${temporaryArticle.availability()}
         articoli.`;
      // [...] parte comune  }.
```

Codice [4.3.4|B.2]: Implementazione con Methods - validazione

```
validateArticle() {
      // [...] parte comune
    if (quantity >= 0 || temporaryArticle.availability + quantity >= 0) {
       this.temporaryOrder.push([temporaryArticle, quantity]);
       if (this.orderIsEmpty && this.temporaryOrder.length > 0)
         this.orderIsEmpty = false;
     } else {
    this.validationError = `La disponibilità attuale per l'articolo
            ${this.selectedArticle}  è  di  ${temporaryArticle.availability}
            articoli.`;
      // [...] parte comune  }.
```

Codice[4.3.4|B.3]: Implementazione con Getters/Setters - validazione

Si può vedere nella prima implementazione la necessità di un doppio accesso all'Oggetto *temporaryArticle* per prelevare rispettivamente i valori delle proprietà *counter* e *ordersCounter*, cosa effettuata un una sola volta nei casi successivi secondo le modalità discusse nel capitolo precedente.

Sono stati infine creati due metodi rispettivamente per l'inserimento di un ordine in lista e per il suo completamento dove si ripete la situazione di doppio accesso contro quello singolo nella funzione *completeOrder*:

```
acceptOrder() {
      let temporaryId = Math.random().toString(36).substr(2, 9);
      let newOrder = { articles: [], date: Date.now(), _id: temporaryId };
      this.ordersVersioned.temporaryId = [];
      this.temporaryOrder.forEach((item) => {
        item[0].ordersCounter += item[1];
        newOrder.articles.push({
          productCode: item[0].productCode,
          quantity: item[1]
        });
      });
      this.ordersVersioned.push(newOrder);
      this.resetOrder();
    },
    completeOrder() {
      let removedOrder = this.ordersVersioned.shift();
      removedOrder.articles.forEach((article) => {
        let temporaryArticle = this.articlesVersioned.find((el) => {
          return el.productCode === article.productCode;
        });
        if (temporaryArticle) {
          temporaryArticle.counter += article.quantity;
          temporaryArticle.ordersCounter -= article.quantity;
        }
      });
    }.
```

Codice[4.3.4|C.1]: Implementazione con Static values - gestione ordini

```
acceptOrder() {
       // [...] parte comune
      this.temporaryOrder.forEach((item) => {
        item[0].setIncomingChanges(item[1]);
        newOrder.articles.push({
          productCode: item[0].productCode,
          quantity: item[1]
        });
      });
```

```
      // [...] parte comune
    },
    completeOrder() {
      // [...] parte comune
        if (temporaryArticle)
          temporaryArticle.newAvailability(article.quantity);
      });    }.
```

Codice[4.3.4|C.2]: Implementazione con Methods - gestione ordini

```
acceptOrder() {
      // [...] parte comune
      this.temporaryOrder.forEach((item) => {
        item[0].incomingChanges = item[1];
        newOrder.articles.push({
          productCode: item[0].productCode,
          quantity: item[1]
        });
      });
      // [...] parte comune
    },
    completeOrder() {
      // [...] parte comune
        if (temporaryArticle)
          temporaryArticle.availability = article.quantity;
      });
    }.
```

Codice [4.3.4|C.3]: Implementazione con Getters/Setters - gestione ordini

4.3.5 Array - Map

In ultimo luogo è stata riadattata la versione con i Getters/Setters all'uso di Map. La struttura dati[47] in questione è quella detenuta al mantenimento dei dati relativi all'ordine in costruzione. In particolare sono state apportate modifiche nelle funzioni di validazione, di inserimento ordine e di pulizia dati conseguente.

Viene riportato a seguire il codice riarrangiato per le prime due[48] e quello di entrambe le versioni per la terza.

[47]Nel codice ha nome `temporaryOrder` e nelle versioni precedenti è una collezione di Array.

[48] Già riportate nel paragrafo precedente per il caso Array.

```
validateArticle() {
      this.validationError = '';
      let temporaryArticle =  // [...] parte comune
      let quantity =  // [...] parte comune
      if (this.temporaryOrder.has(temporaryArticle)) {
        this.validationError = `L'articolo ${this.selectedArticle} è già
                  presente nell'ordine.`;
        this.selectedArticle = '';
        return null;
      }
      if(quantity >= 0 || temporaryArticle.availability + quantity >= 0
       {
        this.temporaryOrder.set(temporaryArticle, quantity);
        if (this.orderIsEmpty && this.temporaryOrder.size > 0)
          this.orderIsEmpty = false;
      } else {
        // [...] uguale all'implementazione con array    }.
```

Codice [4.3.5|D.1]: Implementazione con Map - validazione

```
acceptOrder() {
       // [...] parte comune
      this.temporaryOrder.forEach((quantity, article) => {
        article.incomingChanges = quantity;
        newOrder.articles.push({
          productCode: article.productCode,
          quantity: quantity
        });
      });
       // [...] parte comune
    },
    completeOrder() {
       // [...] uguale all'implementazione con array    }.
```

Codice [4.3.5|D.2]: Implementazione con Map - gestione ordini

```
    CASO ARRAY:
resetOrder() {
      this.showAddArticle = false;
      this.showAddOrder = true;
      this.orderIsEmpty = true;
      this.temporaryOrder.splice(0, this.temporaryOrder.length);
}.
    CASO MAP:
 resetOrder() {
       // [...] parte comune
   this.temporaryOrder.clear();
 }.
```

Codice [4.3.5|D.3]: Implementazione con Array vs. Map - pulizia dati.

Si noti l'impiego dei metodi *Has*, *Set*, *forEach* e *Clear* propri della struttura dati Map. In particolare il primo risparmia dall'utilizzo del *Find* seguito dalla costituzione di un'*Arrow-function* e l'ultimo dall'impiego di *Splice*, meno compatto e scelto per simulare lo svuotamento di un Array senza riallocarlo [AG19], come avviene in una Map.

Capitolo 5

Valutazioni

Vengono di seguito esposte le misurazioni eseguite sul codice prodotto, per fornire la possibilità di un confronto quantitativo tra i diversi gruppi di feature.
Per ogni funzione di interesse sono stati rilevati il numero di righe di codice, il tempo di esecuzione e lo sbalzo di occupazione provocato all'interno del *JS Heap*[49]. Da ultimo viene fornita una tabella di compatibilità tra browser per ogni funzionalità introdotta.
Da sottolineare che tutti i test seguenti sono stati effettuati eseguendo i task per 10 volte, e i valori riportati sono calcolati ricavando la media di queste dieci misurazioni, che si è cercato di ripetere il più possibile sotto le stesse condizioni.

5.1 Gli strumenti utilizzati

Per la rilevazione del tempo di esecuzione è stato utilizzato il pannello *Performance* all'interno del *Chrome DevTools*: esso permette di registrare e analizzare il rendimento di una pagina web rispetto a ciò che lo influenza. In particolare è stata sfruttata la sezione *Timings* attraverso l'introduzione nel codice di marchi all'inizio e alla fine di ogni funzione sotto studio. Il metodo "Performance.mark()", in cui viene passato un nome di variabile, crea nel "performance-entry-buffer" del browser un timestamp con quel nome [MDN21m]. Seguentemente è possibile utilizzare "Performance.measure()" [MDN21l] per la creazione di un ulteriore timestamp, per ogni coppia di marchi, e ottenere il segmento di durata voluto nel pannello Timings.

[49] Per approfondimenti sull'argomento si veda Appendice A: sezione 2 - *Call Stack e Heap*.

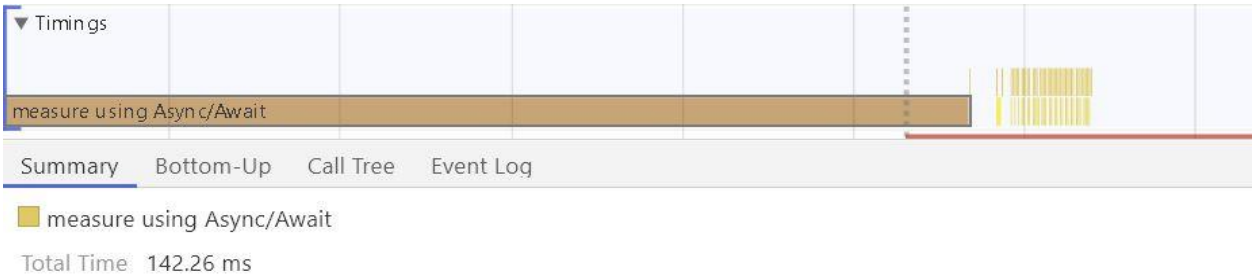

Immagine [5.1.1]: Misurazione riportata nella sezione Timings

Per quanto riguarda la memoria è stata utilizzata l'estensione *Window.performance.memory* di Google Chrome: ritorna un Oggetto che permette di rilevare, in un certo punto dell'esecuzione, alcune metriche relative al JS Heap [THG21]. Nello specifico è stato utilizzato l'attributo *usedJSHeapSize* che contiene il segmento attivo in Byte, della suddetta memoria [MDN21q]. Per ogni funzione sono stati eseguiti un rilevamento alla sua entrata e uno esattamente prima della sua uscita, dopo di che è stato stampato a console il risultato della differenza tra essi.
Quasi tutte le funzioni sotto studio non hanno però comportato sbalzi di Byte all'interno del JS Heap, se non per delle operazioni successive di aggiustamento del layout.

Il primo confronto, tra applicazione monolitica e modulare, è stato invece realizzato tramite l'uso del pannello denominato *Network*. Esso permette di analizzare tutte le richieste che vengono effettuate via rete e, in particolare, riporta la somma di risorse caricate dalla pagina e quanto l'evento *Dom Content Loaded*[50], o DCL, impieghi ad essere emesso.

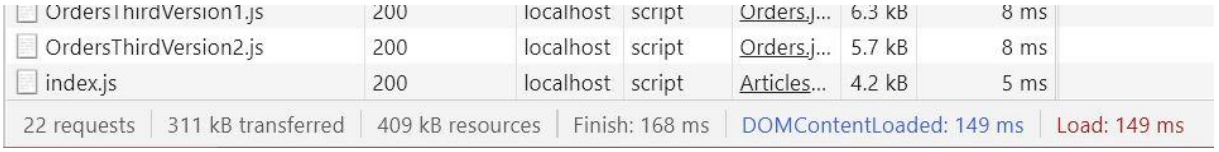

Immagine [5.1.2]: Misurazione riportata nella sezione Network

[50] Indica quando il documento HTML è stato completamente caricato (senza stili e immagini) [MDN20].

5.2 Le misurazioni effettuate

- Single file vs Modules

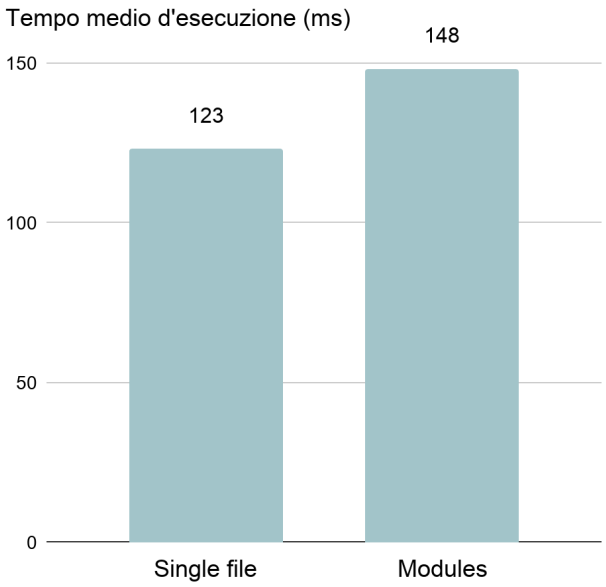

Grafico [5.2.1]: Tempo - Single file vs Modules

Grafico [5.2.2]: Risorse - Single file vs Modules

L'uso dei Moduli richiede più tempo tempo per il caricamento del *DOM*, questo è dovuto al fatto che vengono effettuate più richieste al server per ottenere la stessa quantità di codice. Anche se in questo caso si parla di differenze minime è da precisare che l'applicazione in oggetto è sia contenuta sia poco modularizzata.

- Callbacks vs Promises vs Async/Await

Linee di codice: **67 - 48 - 37**

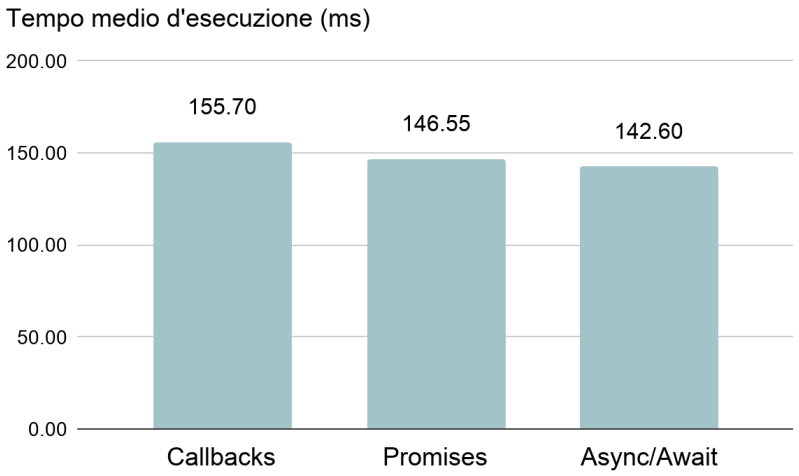

Grafico [5.2.3]: Tempo - Callbacks vs Promises vs Async/Await

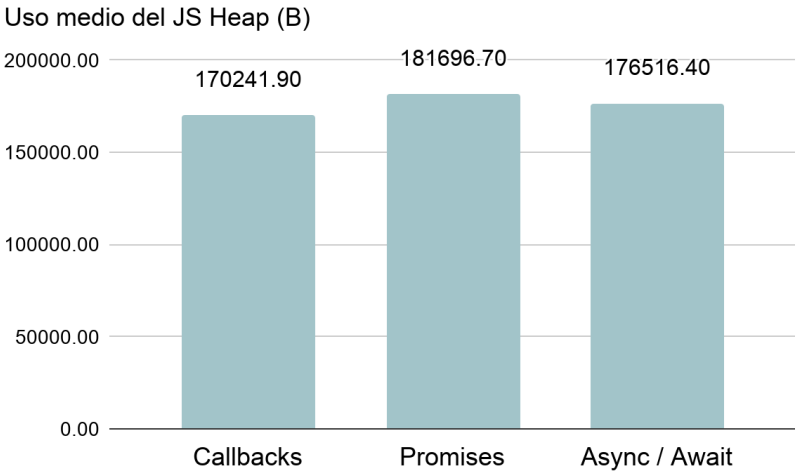

Grafico [5.2.4]: Memoria - Callbacks vs Promises vs Async/Await

Il conteggio delle righe di codice permette di individuare il progresso via via ottenuto in materia di compattezza; nulla di inaspettato considerando che entrambi i costrutti, delle Promesse e di Async/Await, hanno fra i presupposti il miglioramento della leggibilità.
Anche il confronto tra i tempi di esecuzione conferma un continuo miglioramento dal costrutto più antico al più recente.

Gli sbalzi di Byte rilevati nel JS Heap, invece, sono a sfavore dell'impiego delle Promesse; è da notare però che Async/Await riporta la situazione di crescita un passo indietro, ottenendo un risultato circa a metà tra gli altri due.

- Reduce/Concat vs Spread operator vs Flat()

Linee di codice: **29 - 25 - 22**

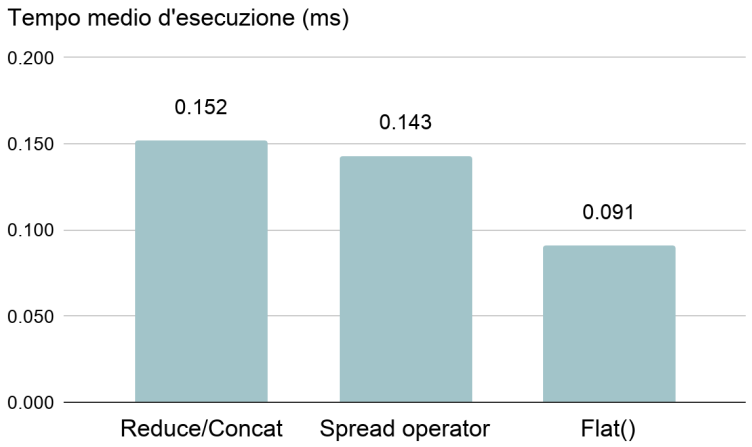

Grafico [5.2.5]: Tempo - Reduce/Concat vs Spread operator vs Flat()

Anche in questo caso vi è una riduzione progressiva sia delle righe di codice ottenute che del tempo di esecuzione. Flat risulta quello più ottimizzato per il contesto creato ed è corretto sia così in quanto è stato introdotto nel linguaggio proprio per sciogliere i livelli di annidamento negli Array. Per ottenere lo stesso risultato con lo Spread operator infatti si è dovuto adattare il codice specificando i singoli Array da unire tra loro, diminuendone di conseguenza, come si è visto, anche la leggibilità.

- If vs Try vs Optional chaining

Linee di codice: **19 - 19 - 9**

L'Optional chaining, il costrutto più recente fra quelli presentati, risulta conveniente sia per la compattezza che per la velocità. La valutazione del Try invece avviene più lentamente dei classici controlli condizionali consecutivi, anche se vi è un guadagno in leggibilità. Non è un risultato del tutto inaspettato poiché in esso sono state generate e gestite apposite eccezioni [MWJ06],[SO17] per la distinzione dei diversi casi.

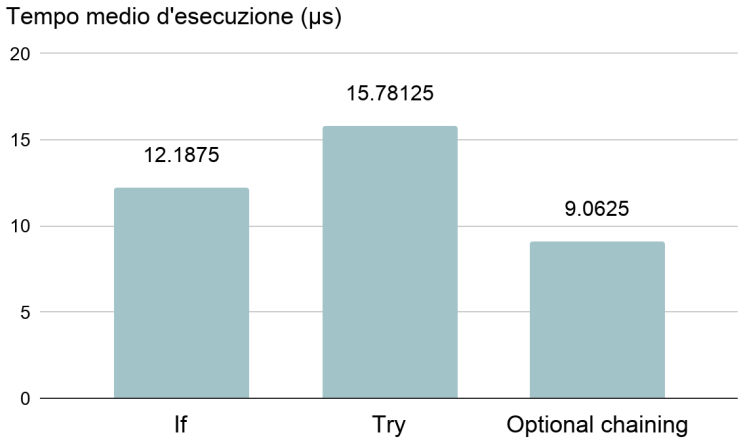

Grafico [5.2.6]: Tempo - If vs Try vs Optional chaining

- Static properties vs Methods vs Getters/Setters

Successivamente vengono riportati i dati raccolti relativi all'interfaccia di interazione. Le linee di codice risultano in crescita a causa della funzione costruttore, che è via via più complessa, ma escludendo questa non vi sono praticamente disparità.

Secondo quanto ottenuto dalle rilevazioni temporali, la feature dei Getters/Setters sembra sconveniente, di nuovo a causa del costruttore. Si noti però che, ignorando quest'ultimo, la velocità della loro applicazione risulta pressoché in linea con quella dei metodi.

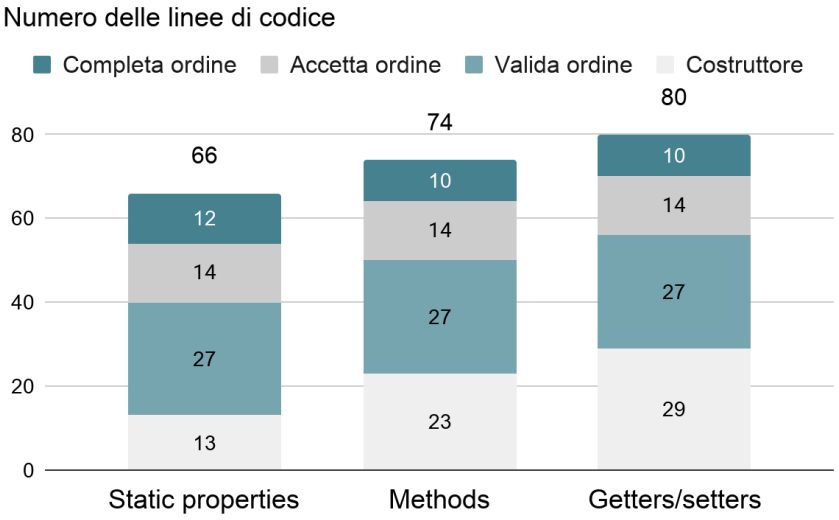

Grafico [5.2.7]: Linee di codice - Static properties vs Methods vs Getters/Setters

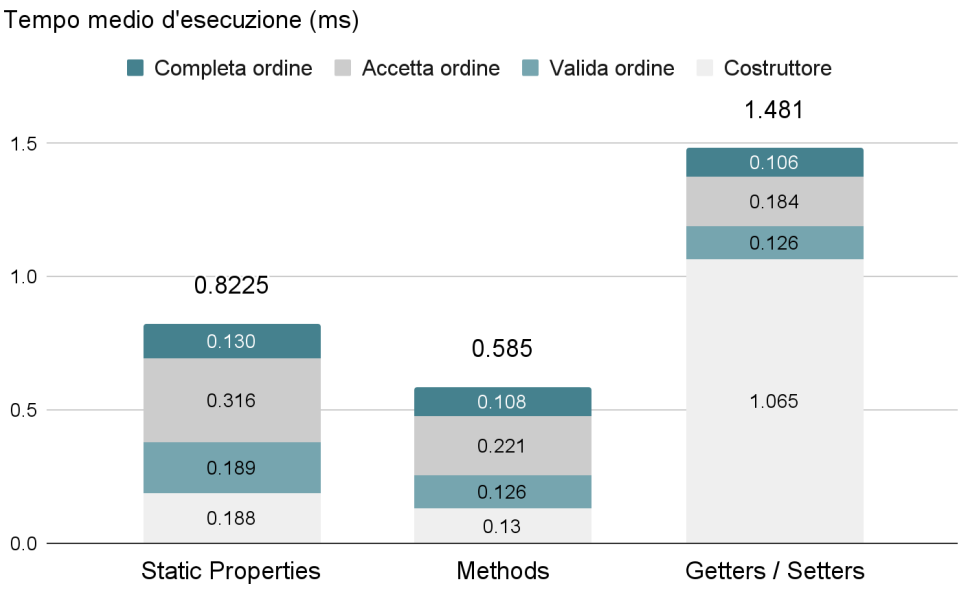

Grafico [5.2.8]: Tempo - Static properties vs Methods vs Getters/Setters

- Uso di Array vs uso di Map

Linee di codice: **80 - 78**

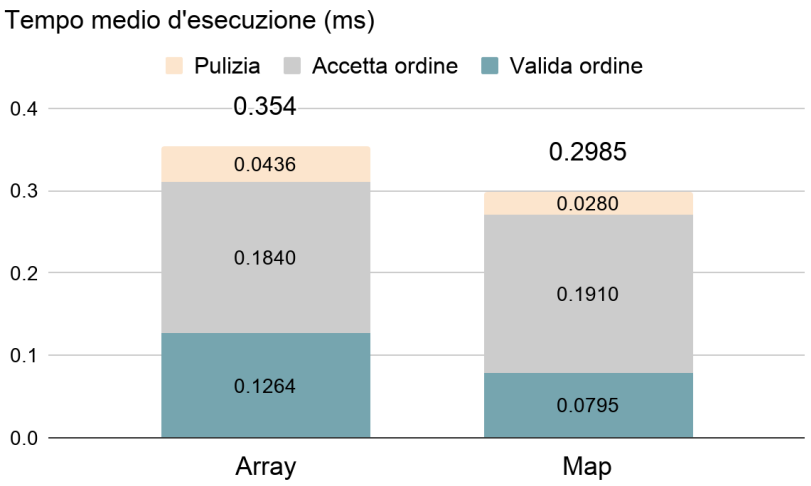

Grafico [5.2.9]: Tempo - Array vs Map

Nonostante, come detto, la struttura interna di una Map sia un Array di Array, essa risulta essere vantaggiosa anche facendone un uso limitato. Soprattutto i suoi metodi *Has*, usato nella validazione dell'ordine, e *Clear*, usato nella pulizia della struttura dati, sembrano essere ottimizzati. Il *forEach*, tuttavia, si direbbe molto simile in termini di efficienza a quello degli Array.

5.3 Le compatibilità tra browser

Di seguito viene riportata una tabella di compatibilità tra browser per ogni feature illustrata. Per la realizzazione della stessa sono stati utilizzati i riferimenti: [JK21], [MDN21a] , [MDN21g], [MDN21i], [MDN21k], [MDN21n], [MDN21o], [MDN21p].
Come si noterà le funzionalità introdotte sono generalmente ben supportate in tutti i browser più comuni (è stata considerata la versione più recente per ognuno), a parte nel caso di Internet Explorer la cui evoluzione è stata soppiantata dall'introduzione di Microsoft Edge nel 2015 [W21c].

Legenda:

= non supportato

= supportato, ma non al 100%

= non sono supportate funzionalità non appartenenti allo standard

= supportato

Riferimenti in tabella:

A = Non supporta l' "optional catch binding".

B = Non supporta i nomi di proprietà dinamici.

C = Non supporta " export * " come namespace.

D = Non supporta "Promise.any()" (ES2021).

E = Non supporta "new Map (Iterable)", "entries", "keys", uguaglianza dell'uso di -0 e 0 come chiave, concatenamento di "set()", "values" .

F = Non supporta "spread parameters" dopo l'Optional chaining.

	Internet Explorer 11	Edge 90	Firefox 88	Chrome 92	Opera 76	Safari 14.1	IOS 14	Samsung 13	Opera Mobile 63
Try (ES3)	**A***								
Getters/Setters (ES5)	**B***								
Modules (ES2015)						**C***	**C***		
Promises (ES2015)					**D***			**D***	**D***
Map (ES2015)	**E***								
Spread operator (ES2015)									
Async/Await (ES2017)									
Flat() (ES2018)									
Optional chaining (ES2020)		**F***		**F***	**F***			**F***	**F***

Tabella [5.3]: Le compatibilità tra browser

5.4 Riepilogo

In questo capitolo sono stati riportati i rendimenti di tutti i costrutti sotto analisi, ognuno secondo la relativa implementazione precedentemente esposta.
Ricapitolando per ogni gruppo di confronto:

- Async/Await è risultato il più veloce con un buon compromesso sull'uso del JS Heap;
- Il Flat è il più ottimizzato e conveniente nella necessità di spacchettamento e unione di Array;
- L'Optional chaining è probabilmente il costrutto che ha apportato il maggior margine di miglioramento;
- I Getters/Setters sono i meno significativi a livello di progresso, soprattutto considerando lo svantaggio visto intorno al costruttore;
- Map è un'ottima soluzione nel caso di gestione dati in corrispondenze chiave-valore con frequenti "look-up" e pulizie.

In generale, si può quindi concludere che da un punto di vista quantitativo la feature più recente sia risultata anche essere quella più efficiente.

Capitolo 6

Conclusioni

Il lavoro eseguito e qui esposto è stato sicuramente uno solo dei percorsi possibili tra la vastità delle feature di ECMAScript, ma lo ritengo comunque un punto di partenza alla scoperta dello stesso e di come il raggiungimento di alcune funzionalità sia mutato negli anni a favore dell'efficienza e/o della chiarezza del codice. Seguo dunque con un giudizio finale e personale per ogni oggetto di studio.

L'utilizzo dei Moduli è stato di enorme aiuto nella gestione del codice: poterlo dividere in file, quindi in modo logico, mi ha permesso di mantenere una visione più chiara e una gestione più veloce del tutto. È vero che a livello di efficienza è risultato un po' più lento, per le richieste multiple al server rispetto ad una singola, però è da dire che nel mondo reale vengono ormai sempre utilizzati dei "Bundler" di Moduli che li riadattano in un unico file.

Per quanto riguarda l'uso delle Promesse l'ho trovato preferibile rispetto a quello di semplici Callback, soprattutto nello scenario creato di Callback-Hell. La gestione a livello di codice mi è risultata più semplice anche solo per la comprensione a vista d'occhio di dove un blocco avesse fine o per la possibilità di gestire gli errori in un unico punto finale. Il fatto che provochino uno sbalzo di Byte maggiore all'interno del JS Heap non mi ha sorpreso una volta ricordato che ogni Promessa è un Oggetto quindi una nuova allocazione nello stesso. Nella pratica risultano comunque gestite più velocemente grazie alla coda differita all'interno del *JS Engine*, come spiegato in *Appendice A* nella sezione tre. Quindi sicuramente le considero un passo in avanti in tema di asincronia. L'introduzione di Async/Await nel linguaggio non fa altro che convincermi di quanto appena detto apportando ancora più "user-friendliness", compattezza e velocità d'esecuzione.

Lo Spread operator in generale è un costrutto che trovo molto utile, ma in questo caso specifico non mi è risultato particolarmente comodo: per poter unire gli Array desiderati li ho dovuti specificare uno ad uno e questo ha afflitto la comprensione del codice. Personalmente, nonostante la piccola perdita di velocità, continuerei a far impiego di Reduce/Concat. Il metodo Flat tuttavia lo ritengo assolutamente il migliore per questo caso d'uso, sia a livello di scrittura del codice sia di prestazioni.

Anche il Try l'ho trovato un po' disagevole, questa volta però per il modo in cui l'ho implementato. Nonostante permetta la gestione di tutte le eccezioni, per quelle prevedibili mi atterrei a semplici controlli condizionali diretti evitando di scatenarle, cosa risultata inefficiente. Probabilmente sarebbe stato meglio compiere gli "If-checks" nel blocco Try e finire il meno possibile nel caso di errore, mantenendo il blocco Catch come ancora di salvezza eventuale. L'Optional chaining, d'altra parte, penso sia un mirabile arricchimento per il linguaggio: non solo è compatto e funzionale, ma si è persino distinto per il rendimento.

Per quanto concerne i Getters/Setters, malgrado l'inefficienza risultata intorno alla funzione costruttore, ritengo siano una feature interessante: unisce l'agevolezza e l'intuizione d'uso di una proprietà statica alla funzionalità offerta da un metodo, aggiungendo un livello di astrazione che personalmente ho apprezzato. Quindi in definitiva in un caso come quello che ho creato, per cui la costruzione degli oggetti è avvenuta prima della visualizzazione della pagina, la preferisco.

In ultimo Map è stata una bella scoperta, non solo per quanto ottenuto dalle valutazioni: il fatto che sia composta da associazioni chiave-valore assicura l'unicità delle stesse, a differenza di un Array di Array standard, inoltre la possibilità di avere come chiavi non solo stringhe o numeri la distingue da un Oggetto rendendola, in contesti similari a quello ideato, più comoda anche grazie ai metodi di cui è corredata.

Reputo infine che, se si volessero ottenere confronti quantitativi ottimali tra i diversi gruppi di feature, bisognerebbe costituire ulteriori scenari e studi mirati, cercando di impiegare al massimo le loro potenzialità (soprattutto nel caso di Map che sarebbe da confrontare anche con un *Object* e da esplorare negli altri suoi metodi). È normale che analisi specifiche e distinte avrebbero permesso di ottenere più dati da discutere. Lo scopo di questa dissertazione tuttavia era quello di fornire una panoramica d'insieme riguardo l'evoluzione di ECMAScript, attraverso alcune delle sue feature più importanti dal punto di vista teorico e pratico. L'avanzamento degli anni sembra essere stato accompagnato sia da da una maggiore

leggibilità e comodità a livello di scrittura di codice che da un generale progresso nei tempi d'esecuzione, in linea con la necessità sempre crescente di sistemi interattivi e veloci. Mi sento quindi di poter affermare che lo studio fin qui condotto abbia avuto, a mio parere, riscontri positivi.

Appendici

Appendice A

Viene introdotta questa appendice con lo scopo di fornire un'idea riguardo il funzionamento dell'esecuzione di codice JavaScript e poterne chiarire, auspicabilmente, alcuni concetti[51] che lo caratterizzano.

JavaScript in pratica: JS Runtime

Una *JS Runtime* è data dalla composizione di tutte le parti necessarie all'esecuzione di codice JavaScript. La più consueta è quella fornita dai browser; ne esistono diversi ma includono tutti un ambiente per il linguaggio, ottenuto a partire da un'architettura di base comune [AV21], [JU18], [RM19].

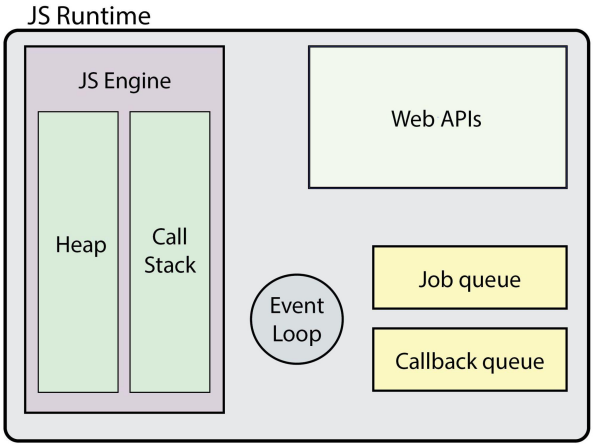

Immagine [Append.1]: Le parti di una JS runtime

[51] Ad esempio: "perché assegnare un Oggetto preesistente non crea veramente un nuovo elemento?" Oppure, "da cosa è reso possibile il codice asincrono?"

Il nucleo fondante è il *JS Engine*[52] che è il luogo in cui avviene effettivamente l'esecuzione.

Il codice viene prima di tutto sottoposto a parsing, ovvero letto e strutturato in un albero di sintassi astratta[53] [SB20]. Questa struttura dati viene poi compilata e tradotta in codice macchina grazie a cui è resa possibile l'esecuzione vera e propria. Una cosa interessante è che il primo eseguibile prodotto non è ottimizzato per favorire la velocità di disposizione del codice: esiste un processo di ottimizzazione, ripetibile più volte, che riporta quest'ultimo in una fase di compilazione e lo riserve via via più raffinato. Il tipo di compilazione qui accennato è chiamato "just-in-time" perché non è una compilazione pura, difatti essa avviene sì in una volta sola ma poi segue immediatamente l'esecuzione [AS20].

L'elaborazione del codice dà luogo innanzi tutto al contesto globale[54] utilizzato dal codice top-level. Quest'ultimo è il primo ad essere processato dalla CPU nell'ambiente creato, successivamente l'attenzione viene rivolta alle funzioni e poi alle Callback: per ognuna viene creato il proprio contesto che ne contiene le variabili di ambiente, la "Scope-chain" e il "This"[55].

1. Il contesto di esecuzione

Prima dell'esecuzione di una funzione vengono determinate tutte le dichiarazioni di variabile in essa presenti e per ognuna di queste viene introdotta una proprietà tra le variabili d'ambiente. Questo ha come ripercussione un fenomeno, detto *hoisting* [RM17], per cui è possibile utilizzare alcuni nomi prima della loro dichiarazione. "Alcuni" perchè in ES2015 sono state inserite due nuove tipologie di variabili, *let* e *const*, che sono invece mantenute in una zona protetta, la *Temporal Dead Zone* (o TDZ) [KP20], fino alla loro inizializzazione: un tentato accesso anticipato a queste sfocia in un errore di referenza, mentre nel caso di *var* vi è l'assegnazione di default a "undefined" e nelle funzioni al loro stesso corpo.

[52] Ogni runtime ha la propria Engine, la più famosa è V8 di Google.
[53] Struttura ordinata di relazioni tra le componenti significative del programma, secondo quanto definito dal linguaggio, che costituiscono la rappresentazione del codice nell'engine.
[54] Il contesto d'esecuzione di un pezzo di codice ne contiene le informazioni necessarie, appunto, all'esecuzione. Quello globale è unico ed è condiviso da tutte le parti.
[55] Non nel caso delle "arrow function"che usano quello della funzione genitore più vicina.

Per ogni variabile viene determinata a livello lessicale, tramite l'uso di blocchi " {} " e funzioni, la regione in cui può essere acceduta, chiamata *scope*. Questo può essere globale, di funzione o di blocco in base a dove è posizionata la dichiarazione. Ciascuna funzione, inoltre, ha visibilità sulle variabili presenti nelle aree di scope in cui è contenuta. La Scope-chain è deputata proprio al mantenimento della catena di referenze fra livelli di scope successivi, che definisce tutto ciò che può essere acceduto in un certo punto del codice [RM17].

Per ultimo in ogni contesto vi è una variabile speciale, il This, che contiene il valore calcolato dinamicamente del detentore della funzione stessa: per un metodo è l'Oggetto che lo contiene, per una funzione semplice è "undefined" [RM17] .

2. Call Stack e Heap

Una volta avvenute le operazioni precedenti, avviene l'esecuzione vera e propria per mezzo della *Call Stack* e del *JS Heap*.

Lo Heap è una memoria libera in cui vengono mantenuti tutti gli oggetti che servono al programma per questioni di spazio[56], mentre la Call Stack è il luogo di esecuzione vero e proprio in cui viene mantenuto l'ordine attraverso i contesti esecutivi [FG20b]: ogni chiamata di funzione porta al posizionamento del suo contesto in cima allo Stack fino alla sua terminazione. [RM19]

In particolare è da chiarire come vengano trattate diversamente le variabili in base al fatto che il loro valore sia un tipo primitivo[57] o un *Object*: si è infatti detto che questi ultimi sono nello Heap.

2.1 La gestione della memoria

Per ciascuna dichiarazione di variabile viene creato nello Stack un identificatore univoco connesso all'indirizzo del valore associato. Nel caso venga assegnato un valore già esistente in memoria, l'identificatore viene collegato ad esso e un'eventuale modifica del valore comporta un cambio di riferimento[58][FG20b].

L'identificatore di un dato strutturato è associato all'indirizzo che ne individua la posizione nello Heap. Ciò implica che una modifica sul contenuto dell'Oggetto non provoca un

[56] Un Oggetto potenzialmente può essere molto grande.
[57] Uno fra Number String, Boolean, Undefined, Null, Symbol, BigInt.
[58] Nella call stack il valore salvato in un certo indirizzo non può essere cambiato.

cambiamento entro la Call Stack, che mantiene la connessione immutata, bensì nell'altra memoria [SK12]. Questo è il motivo per cui se si assegna un Oggetto a più variabili e se ne muta, ad esempio, le proprietà tramite una di esse questo si ripercuote anche sulle altre[59]: continuano infatti tutte ad essere collegate allo stesso indirizzo in corrispondenza del quale avviene l'effettiva trasformazione dei dati.

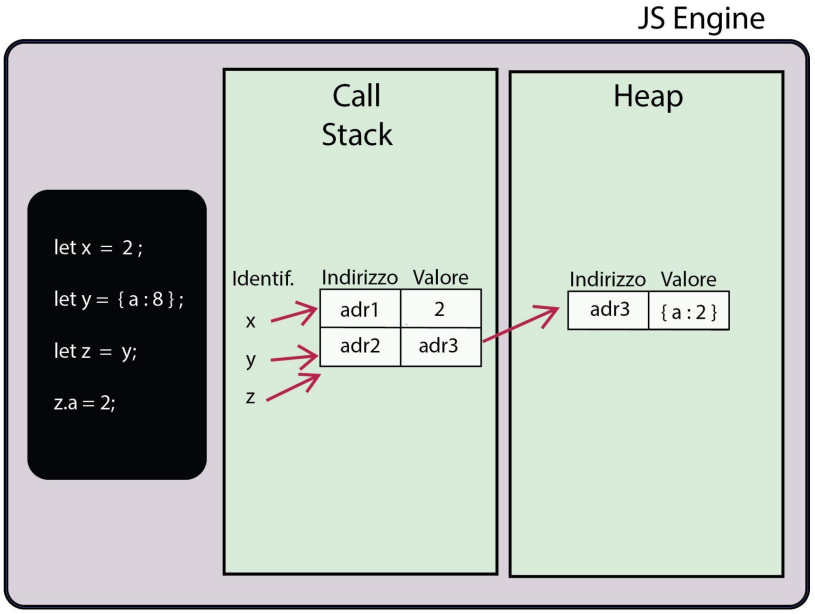

Immagine [Append.2]: La gestione della memoria nella JS Engine

3. L'asincronia

A livello di lettura di codice l'asincronia implica il fatto che l'ordine delle righe non determina quello di elaborazione. In particolare, quando l'esecuzione di una linea di codice sottende computazioni arbitrariamente lunghe o lente, essa verrà portata a termine solo dopo che gli incarichi saranno portati a termine in altro luogo.

L'asincronia è molto importante in JavaScript perché si tratta di un linguaggio *Single-Threaded*: vi è un unico flusso di esecuzione nella CPU quindi, per non bloccarlo, i compiti che richiedono una quantità di tempo rilevante vengono spostati sullo sfondo e reimmessi nel corso principale solo quando conclusi [FG20a].

[59] É anche il motivo per cui in un Oggetto si può cambiare il valore di una proprietà anche quando dichiarata `const`.

Le procedure di natura asincrona sono un insieme definito: i timer, alcuni eventi[60] del *DOM*[61] e le chiamate *Ajax*[62]. Queste ultime sono le più importanti e utilizzate in quanto permettono la comunicazione asincrona con un server facendo uso di un Oggetto chiamato *XMLhttpRequest*[63] [JD14] .

L'asincronia nella pratica è permessa dall'esistenza dell'*Event-Loop*: un processo esterno all'engine che regola l'interazione tra la call-stack e due code di compiti completati. Quando sulla pila di esecuzione è il turno di una procedura asincrona, questa e la Callback associata vengono spostate nella parte della JS Runtime denominata delle *Web APIs*[64] [RM19]. In questo ambiente l'incarico viene eseguito fino all'emissione di un evento di completamento. A quel punto la Callback lì registrata viene posizionata in una delle due code presenti: se consecutiva a una chiamata Ajax tramite Promesse nella *Job-Queue*[65] [FC18], [FG20a], nella *Callback-Queue*[66] altrimenti.

Quando lo Stack è vuoto ha luogo un *Event-Loop tick*: l'Event-Loop rimuove il primo task, se presente, dalla Callback-Queue alla Call-Stack per farlo eseguire. Dopo di che viene controllata la Job-Queue e ne vengono fatte processare tutte le Callback presenti, anche quelle che vengono aggiunte durante questo procedimento. Una volta concluse, se la cima della Call-Stack è ancora vuota, ha luogo un nuovo "tick" [JA14] .

In conseguenza di questo tipo di gestione si dice che i Microtask abbiano priorità sui Task: potenzialmente la Job-Queue potrebbe bloccare l'altra [CM20], anche se nella realtà ciò non accade. Questo è il motivo, ad esempio, per cui in JavaScript non bisogna dare per scontato che un Timer duri esattamente i millisecondi dichiarati: non solo esistono delle precedenze di risoluzione, ma inoltre la sua Callback finirebbe nella coda a "minore priorità".

[60] Quelli che comportano l'esecuzione di un qualche lavoro, come il caricamento di un'immagine.
[61] "Document Object Model": la rappresentazione di un documento HTML/XML nel browser.
[62] Asynchronous JavaScript and XML: un insieme di tecnologie che offrono funzionalità asincrone nel browser.
[63] Standardizzato dal WHATWG. Un'API sotto forma di Oggetto che permette al browser di creare richieste Ajax e di elaborarne le risposte asincrone.
[64] Presenta le API che il browser specifico mette a disposizione di JavaScript.
[65] Anche conosciuta come Microtask-Queue.
[66] Anche conosciuta come Task-Queue. Essa contiene anche le Callback associate a eventi del DOM che non implicano procedure asincrone, ma che vengono comunque eseguite nell'ambiente delle web APIs.

Elenchi

Esempi

Immagini

Codice

Grafici

Tabelle

Bibliografia

[AC20a] Andrea Chiarelli. *"JavaScript: What's New in ECMAScript 2020"*. 18 Maggio 2020. URL: https://auth0.com/blog/javascript-whats-new-es2020/#New-Operators.

[AC20b] Alligator.io Community. *"Exploring Async/Await Functions in JavaScript"*. 4 Settembre 2020. URL: https://www.digitalocean.com/community/tutorials/js-async-functions.

[AG19] Ayush Gupta. *"In Javascript how to empty an array"*. 16 Settembre 2019. URL: https://www.tutorialspoint.com/in-javascript-how-to-empty-an-array.

[AN20] Alexander Nnakwue. *"The evolution of asynchronous programming in JavaScript"*. 16 Aprile 2020. URL: https://blog.logrocket.com/evolution-async-programming-javascript/.

[AO12] Addy Osmani. *"Writing Modular JavaScript With AMD, CommonJS & ES Harmony"*. 2012. URL: https://addyosmani.com/writing-modular-js/.

[AR14] Axel Rauschmayer. *"Speaking JavaScript: An In-Depth Guide for Programmers"*. O'Reilly Media; Edizione 1, 25 Febbraio 2014. URL: http://speakingjs.com/es5/ch14.html#_exception_handling_in_javascript.

[AR18] Axel Rauschmayer. *"Exploring ES6, Upgrade to the next version of JavaScript"*. Ecmanauten, Ultima modifica: 2 Marzo 2018. URL: https://exploringjs.com/es6/ch_about-es6.html#sec_ecmascript-history.

[AS20] Allan Sendagi. *"Inside the Javascript Engine: Compiler and Interpreter Is Javascript compiled or interpreted?"*. 29 Gennaio 2020. URL: https://medium.com/@allansendagi/inside-the-javascript-engine-compiler-and-interpreter-c8faa638b0d9.

[AV21] Ayush Verma *"Event Loop in JavaScript - Call Stack, Web APIs, Event Queue, Micro-tasks, Macro-tasks"*. Ultima modifica: 13 Aprile 2021. URL: https://towardsdev.com/event-loop-in-javascript-672c07618dc9. (Ultimo accesso: 24 Aprile 2021).

[BE08] Brendan Eich. *"ECMAScript Harmony"*. 13 Agosto 2008. URL: https://mail.mozilla.org/pipermail/es-discuss/2008-August/006837.html.

[BI15] Ben Ilegbodu. *"History of ECMAScript"*. 29 Luglio 2015. URL: https://www.benmvp.com/blog/learning-es6-history-of-ecmascript.

[CM20] Carl Mungazi. *"JavaScript job queue and microtasks"*. Ultima modifica: 7 Aprile 2020. URL: https://careersjs.com/magazine/javascript-job-queue-microtask/. (Ultimo accesso: 24 Aprile 2021).

[DE21] Daniel Ehrenberg. *"Private methods and getter/setters for JavaScript classes"*. TC39 Proposal, Ultima modifica: 14 Aprile 2021. URL: https://github.com/tc39/proposal-private-methods. (Ultimo accesso: 23 Aprile 2021).

[EIa] ECMA International. *"Home"*. URL: https://www.ecma-international.org. (Ultimo accesso: 12 Aprile 2021).

[EIb] ECMA International. *"TC39"*. URL:
https://www.ecma-international.org/technical-committees/tc39.
(Ultimo accesso: 9 Novembre 2020).

[EI09] Ecma International. *"ECMAScript Language Specification"*. Standard ECMA-262 ECMAScript®, Edizione 5, Dicembre 2009. URL:
https://www.ecma-international.org/wp-content/uploads/ECMA-262_5th_edition_december_2009.pdf.

[EI17] Ecma International. *"ECMAScript Language Specification"*. Standard ECMA-262 ECMAScript®, Edizione 8, Giugno 2017. URL:
https://262.ecma-international.org/8.0/#sec-object.entries.

[FC18] Flavio Copes. *"The JavaScript Event Loop"*. 18 Aprile 2018. URL:
https://flaviocopes.com/javascript-event-loop/.

[FG20a] Felix Gerschau. *"JavaScript Event Loop and Call Stack Explained"*. 2020. URL:
https://felixgerschau.com/javascript-event-loop-call-stack/.

[FG20b] Flavio Gerschau. *"JavaScript's Memory Management Explained"*. 2020. URL:
https://felixgerschau.com/javascript-memory-management/.

[GG19] GeeksforGeeks. *"Map in JavaScript"*. Ultimo aggiornamento: 13 Maggio 2019. URL:
https://www.geeksforgeeks.org/map-in-javascript/. (Ultimo accesso: 23 Aprile 2021).

[IK20] Ilya Kantor. *"Property getters and setters"*. 28 Maggio 2020. URL:
https://javascript.info/property-accessors#smarter-getters-setters.

[JA14] Jake Archibald *"Tasks, microtasks, queues and schedules"*. 17 Agosto 2015. URL:
https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/.

[JD14] John Duckett. *"JAVASCRIPT & JQUERY"*. Versione originale Copyright © 2014 by John Wiley & Sons. Inc., Indianapolis, Indiana; versione italiana Copyright © 2017 Apogeo - IF - Idee editoriali Feltrinelli S.r.L. Finito di stampare: Settembre 2019.

[JI20] JavaScipt.info *"Modules, introduction"*. 1 Novembre 2020. URL: `https://javascript.info/modules-intro.`

[JK21] Juriy Zaytsev e contributori vari. *"ECMAScript compatibility table"*. GitHub, Copyright (c) 2010-2013 Juriy Zaytsev. Ultima modifica: 29 Aprile 2021. URL: `http://kangax.github.io/compat-table/.` (Ultimo accesso: 3 Maggio 2021).

[JU18] Jamie Uttariello. *"The Javascript Runtime Environment ...and, you know, how Javascript works."*. 21 Aprile 2018. URL: `https://olinations.medium.com/the-javascript-runtime-environment-d58fa2e60dd0.`

[KP20] Kealan Parr. *"What is the Temporal Dead Zone (TDZ) in JavaScript?"*. 6 Ottobre 2020. URL: `https://www.freecodecamp.org/news/what-is-the-temporal-dead-zone/.`

[LH20] Love Huria. *"ECMAScript - Past, Current and the future"*. 25 Novembre 2020. URL: `https://lhuria94.github.io/ecmascript-past-present-and-future/.`

[MA17] Michael Aranda. *"What's the difference between JavaScript and ECMAScript?"*. 28 Ottobre 2017. URL: `https://www.freecodecamp.org/news/whats-the-difference-between-javascript-and-ecmascript-cba48c73a2b5/#javascript.`

[MDN20] MDN Web Docs. *"Window:DOMContentLoaded event"*. Ultima modifica: 18 Dicembre 2020. URL: `https://developer.mozilla.org/en-US/docs/Web/API/Window/DOMContentLoaded_event.` (Ultimo accesso: 1 Maggio 2021).

[MDN21a] MDN Web Docs. *"try...catch"*. Ultima modifica: 6 Aprile 2021. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch.` (Ultimo accesso: 23 Aprile 2021).

[MDN21b] MDN Web Docs. *"Details of the object model"*. Ultima modifica: 15 Aprile 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model. (Ultimo accesso: 23 Aprile 2021).

[MDN21c] MDN Web Docs. *"Property accessors"*. Ultima modifica: 12 Aprile 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_accessors. (Ultimo accesso: 23 Aprile 2021).

[MDN21d] MDN Web Docs. *"Import"*. Ultima modifica: 14 Aprile 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import. (Ultimo accesso: 23 Aprile 2021).

[MDN21e] MDN Web Docs. *"Export"*. Ultima modifica: 19 Febbraio 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export. (Ultimo accesso 23 Aprile 2021).

[MDN21f] MDN Web Docs. *"Promise.prototype.then()"*. Ultima modifica: 6 Aprile 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then. (Ultimo accesso: 23 Aprile 2021).

[MDN21g] MDN Web Docs. *"Map"*. Ultima modifica: 15 Aprile 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map. (Ultimo accesso: 23 Aprile 2021).

[MDN21h] MDN Web Docs. *"Iteration protocols"*. Ultima modifica: 19 Febbraio 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols. (Ultimo accesso: 23 Aprile 2021).

[MDN21i] MDN Web Docs. *"Spread syntax (...)"*. Ultima modifica: 6 Aprile 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax. (Ultimo accesso: 23 Aprile 2021).

[MDN21j] MDN Web Docs. *"Array.prototype.flatMap()"*. Ultima modifica: 19 Aprile 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/flatMap. (Ultimo accesso: 23 Aprile 2021).

[MDN21k] MDN Web Docs. *"Optional chaining (?.)"*. Ultima modifica: 14 Aprile 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional_chaining. (Ultimo accesso: 23 Aprile 2021).

[MDN21l] MDN Web Docs. *"performance.measure()"*. Ultima modifica: 27 Aprile 2021. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance/measure. (Ultimo accesso: 1 Maggio 2021).

[MDN21m] MDN Web Docs. *"performance.mark()"*. Ultima modifica: 19 Febbraio 2021. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance/mark. (Ultimo accesso: 1 Maggio 2021).

[MDN21n] MDN Web Docs. *"Array.prototype.flat()"*. Ultima modifica: 28 Aprile 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/flat. (Ultimo accesso: 3 Maggio 2021).

[MDN21o] MDN Web Docs. *"getter"*. Ultima modifica: 28 Aprile 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get. (Ultimo accesso: 3 Maggio 2021).

[MDN21p] MDN Web Docs. *"setter"*. Ultima modifica: 28 Aprile 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/set. (Ultimo accesso: 3 Maggio 2021).

[MDN21q] MDN Web Docs. *"Performance.memory"*. Ultima modifica: 19 Febbraio 2021. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance/memory. (Ultimo accesso: 3 Maggio 2021).

[MDN21r] MDN Web Docs. *"Working with objects"*. Ultima modifica: 29 Marzo 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#defining_getters_and_setters. (Ultimo accesso: 6 Maggio 2021).

[MDN21s] MDN Web Docs. *"Object.defineProperties()"*. Ultima modifica: 5 Maggio 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties. (Ultimo accesso: 6 Maggio 2021).

[MS18] Maja Shavin. *"ES6 - Map vs Object - What and when?"*. 1 Febbraio 2018. URL: https://medium.com/front-end-weekly/es6-map-vs-object-what-and-when-b80621932373.

[MWJ06] Mark Wilton Jones. *"Efficient Javascript"*. 2 Novembre 2006. URL: https://dev.opera.com/articles/efficient-javascript/?page=2#trycatch.

[PR19] Preethi Ranjit. *"Guide to Using Getter and Setters in JavaScript for Developers"*. Ultima modifica: 11 Aprile 2019. URL: https://www.hongkiat.com/blog/getters-setters-javascript/#overwrite-prevention. (Ultimo accesso: 23 Aprile 2021).

[RM17] Rupesh Mishra. *"Execution context, Scope chain and JavaScript internals"*. 18 Maggio 2017. URL: https://medium.com/@happymishra66/execution-context-in-javascript-319dd72e8e2c#:~:text=Execution%20context%20(EC)%20is%20defined,to%20at%20a%20particular%20time.

[RM19] Rupesh Mishra. *"JavaScript Internals: JavaScript engine, Run-time environment & setTimeout Web API"*. 19 Giugno 2019. URL: https://blog.bitsrc.io/javascript-internals-javascript-engine-run-time-environment-settimeout-web-api-eeed263b1617.

[SB20] Supratik Basu. *"JavaScrip Compiler Optimization Techniques--only for Experts"*. 19 Aprile 2020. URL: https://codeburst.io/javascript-compiler-optimization-techniques-only-for-experts-58d6f5f958ca.

[SG19] Selva Ganesh. *"JavaScript: What's new in ECMAScript 2019 (ES2019)/ES10?"*. 8 Febbraio 2019. URL: https://medium.com/@selvaganesh93/javascript-whats-new-in-ecmascript-2019-es2019-es10-35210c6e7f4b.

[SK12] Shivprasad Koirala. *"Six important .NET concepts: Stack, heap, value types, reference types, boxing, and unboxing"*. 14 Maggio 2012. URL: https://www.codeproject.com/Articles/76153/Six-important-NET-concepts-Stack-heap-value-types#Stack%20and%20Heap.

[SO15]Autori vari. *"javascript - assigning setter / getter for js object"*. StackOverflow. Ultima modifica: 9 Maggio 2015. URL: https://stackoverflow.com/questions/30140280/assigning-setter-getter-for-js-object. (Ultimo accesso: 6 Maggio 2021).

[SO17]Autori vari. *"Why isn't Try/Catch used more often in Javascript?"*. StackOverflow. Ultima modifica: 17 Giugno 2017. URL: https://stackoverflow.com/questions/12609527/why-isnt-try-catch-used-more-often-in-javascript?noredirect=1&lq=1. (Ultimo accesso: 4 Maggio 2021).

[SP16] Sebastian Peyrott. *"A Rundown of JavaScript 2015 features"*. 16 Novembre 2016. URL: https://auth0.com/blog/a-rundown-of-es6-features/.

[TC3921] Shu-yu Guo, Michael Ficarra, Kevin Gibbons. *"ECMAScript® 2022 Language Specification"*. TC39, Draft ECMA-262, 6 Aprile 2021. URL: https://tc39.es/ecma262/#sec-overview.

[THG21] Todd H Gardner. *"Monitoring JavaScript Memory"*. © 2013-2021 TrackJS LLC, URL: https://trackjs.com/blog/monitoring-javascript-memory/. (Ultimo accesso: 4 Maggio 2021).

[W21a] Wikipedia. *"ECMAScript"*. Ultima modifica: 22 Aprile 2021. URL: https://en.wikipedia.org/wiki/ECMAScript#History. (Ultimo accesso: 23 Aprile 2021).

[W21b] Wikipedia. *"VBScript"*. Ultima modifica: 26 Marzo 2021. URL: https://en.wikipedia.org/wiki/VBScript#:~:text=VBScript%20(%22Microsoft%20Visual%20Basic%20Scripting,and%20other%20advanced%20programming%20constructs. (Ultimo accesso: 23 Aprile 2021).

[W21c] Wikipedia. *"Microsoft Edge"*. Ultima modifica: 29 Aprile 2021. URL: https://en.wikipedia.org/wiki/Microsoft_Edge. (Ultimo accesso: 3 Maggio 2021).

[ZL17] Zell Liew. *"Promise in JavaScript"*. 14 Giugno 2017. URL: https://zellwk.com/blog/js-promises/.