

Parte 1

Il linguaggio JavaScript

JS

Ilya Kantor

Costruito a 22 gennaio 2023

L'ultima versione di questo tutorial è reperibile a <https://it.javascript.info>.

Lavoriamo costantemente per migliorare il tutorial. Se trovi qualsiasi errore, segnalalo sul nostro [github](#).

- [Introduzione](#)
 - [Introduzione a JavaScript](#)
 - [Manuali e Specifiche](#)
 - [Code editor](#)
 - [Developer console](#)
- [Le basi JavaScript](#)
 - [Hello, world!](#)
 - [Struttura del codice](#)
 - [Le tecniche moderne, "use strict"](#)
 - [Variabili](#)
 - [Tipi di dato](#)
 - [Interazioni: alert, prompt, confirm](#)
 - [Conversione di tipi](#)
 - [Operatori di base](#)
 - [Confronti](#)
 - [Operatori condizionali: if, ?'](#)
 - [Operatori logici](#)
 - [Nullish coalescing operator ??'](#)
 - [Cicli: while e for](#)
 - [L'istruzione "switch"](#)
 - [Funzioni](#)
 - [Function expression](#)
 - [Arrow functions, le basi](#)
 - [Specialità di JavaScript](#)
- [Qualità del codice](#)
 - [Debugging in the browser](#)
 - [Stile di programmazione](#)
 - [Commenti](#)
 - [Codice ninja](#)
 - [Test automatici con Mocha](#)
 - [Polyfills e transpilers](#)
- [Oggetti: le basi](#)
 - [Oggetti](#)
 - [Oggetti: riferimento e copia](#)
 - [Garbage collection \("Spazzatura"\)](#)
 - [Metodi degli oggetti, "this"](#)
 - [Costruttore, operatore "new"](#)

- Concatenamento opzionale '?.'
- Il tipo Symbol
- Conversione da oggetto a primitivi
- Tipi di dato
 - Metodi dei tipi primitivi
 - Numeri
 - Stringhe
 - Array
 - Metodi per gli array
 - Iteratori
 - Map e Set
 - WeakMap e WeakSet
 - Object.keys, values, entries
 - Assegnamento di destrutturazione
 - Date e time
 - Metodi JSON, toJSON
- Gestione avanzata delle funzioni
 - Ricorsione e pila
 - Parametri resto e operatore di espansione
 - Variable scope, closure
 - Il vecchio "var"
 - Oggetto globale
 - Oggetto funzione, NFE
 - La sintassi "new Function"
 - Pianificazione: setTimeout e setInterval
 - *Decorators* e forwarding, call/apply
 - Function binding
 - Arrow functions rivisitate
- Configurazione delle proprietà dell'oggetto
 - Attributi e descrittori di proprietà
 - Proprietà getters e setters
- Prototypes, inheritance
 - Prototypal inheritance
 - F.prototype
 - Native prototypes
 - Metodi di prototype, objects senza __proto__
- Classi
 - Sintassi base delle classi
 - Ereditarietà delle classi
 - Proprietà e metodi statici
 - Proprietà e metodi privati e protetti

- Estendere le classi built-in
- Verifica delle classi: "instanceof"
- Mixins
- Gestione degli errori
 - Gestione degli errori, "try...catch"
 - Errori personalizzati, estendere la classe Error
- Promises, async/await
 - Introduzione: callbacks
 - Promise
 - Concatenamento di promise (promise chaining)
 - Gestione degli errori con le promise
 - Promise API
 - Promisification
 - Microtasks
 - Async/await
- Generators, iterazioni avanzate
 - I generatori
 - Iteratori e generatori asincroni
- Moduli
 - Moduli, introduzione
 - Export e Import
 - Dynamic imports
- Miscellaneous
 - Proxy e Reflect
 - Eval: eseguire una stringa di codice
 - Currying
 - Il tipo Reference
 - BigInt

Impareremo JavaScript, iniziando dalle basi e passando a concetti avanzati come OOP.

Ci concentreremo principalmente sul linguaggio, con un minimo di annotazioni riguardo gli ambienti di sviluppo.

Introduzione

Studieremo il linguaggio JavaScript e l'ambiente di sviluppo.

Introduzione a JavaScript

Vediamo cosa rende JavaScript così speciale, cosa è possibile ottenere tramite il suo utilizzo e tutte le tecnologie che possono essere applicate per renderlo adatto ad ogni necessità.

Cos'è JavaScript?

JavaScript è stato creato con lo scopo di “dare vita alle pagine web”.

I programmi che sfruttano questo linguaggio vengono chiamati *script*. Possono essere scritti direttamente nel documento HTML ed eseguiti in automatico al caricamento della pagina.

Gli script vengono scritti ed eseguiti come testo semplice. Per questo non richiedono alcuna fase di preparazione o compilazione per essere eseguiti.

Sotto questo aspetto, JavaScript è molto differente da un altro linguaggio chiamato [Java](#).

Perché si chiama JavaScript?

In origine JavaScript aveva un altro nome: “LiveScript”. In quel periodo Java era molto popolare, per questo si è pensato che identificare Javascript come il “fratello minore” di Java potesse aiutare alla sua diffusione.

Evolvendosi, JavaScript è diventato un linguaggio completamente indipendente, le cui specifiche sono definite da [ECMAScript](#), e adesso non ha quasi nulla in comune con Java.

Attualmente, JavaScript può essere eseguito non solo nei browser, ma anche nei server web e in altri ambienti che supportano il [motore JavaScript](#) (JavaScript engine).

Il browser ha un suo motore JavaScript integrato, chiamato alle volte “JavaScript Virtual Machine”.

Esistono altri motori JavaScript, tra cui:

- [V8](#) – per Chrome e Opera.
- [SpiderMonkey](#) – per Firefox.
- ...Ci sono altri codenames come “Chakra” per IE, “JavaScriptCore”, “Nitro” e “SquirrelFish” per Safari, etc.

I nomi citati sopra possono essere utili da ricordare, poiché si possono trovare spesso in articoli che trattano di sviluppo web. Anche noi li useremo. Ad esempio, se “una caratteristica X è supportata da V8”, probabilmente funzioneranno senza problemi in Chrome e Opera.

i Come funzionano questi motori?

Il funzionamento di questi motori è complicato, ma i concetti alla base sono semplici.

1. I motori (integriti nei browser) leggono (“analizzano”) lo script.
2. Successivamente convertono (“compilano”) lo script nel linguaggio della macchina.
3. Infine il “codice macchina” viene eseguito, molto rapidamente.

Il motore ottimizza il codice ad ogni passaggio del processo, anche durante l'esecuzione dello script già compilato, quando ne analizza il flusso dati. Nonostante tutto l'esecuzione dello script risulta essere molto veloce.

Cosa può fare JavaScript a livello browser?

JavaScript, al giorno d'oggi, è un linguaggio di programmazione “sicuro”. Non consente alcun accesso di basso livello alla memoria o alla CPU. Questo perché è stato creato con lo scopo di funzionare nei browser, che non richiedono questi tipi di privilegi.

Le capacità di JavaScript dipendono molto dall'ambiente in cui lo si esegue. Ad esempio, [Node.js ↗](#) supporta funzioni che consentono a JavaScript di scrivere/leggere file, eseguire richieste web, etc.

Integrato nel browser Javascript può fare qualsiasi cosa legata alla manipolazione della pagina, all'interazione con l'utente e con il server.

Ad esempio, è possibile:

- Aggiungere HTML alla pagina, cambiare il contenuto esistente, modificare lo stile.
- Reagire alle azioni dell'utente, click del mouse, movimenti del cursore, input da tastiera.
- Inviare richieste al server tramite la rete, caricare e scaricare file (con l'ausilio di [AJAX ↗](#) e [COMET ↗](#)).
- Prelevare e impostare cookies, interrogare l'utente, mostrare messaggi.
- Memorizzare i dati client-side (“memorizzazione locale”).

Cosa NON può fare JavaScript a livello browser?

Per la sicurezza dell'utente, le possibilità di JavaScript nel browser sono limitate. L'intento è di prevenire che una pagina “maligna” tenti di accedere alle informazioni personali o di danneggiare i dati degli utenti.

Esempi di queste restrizioni possono essere:

- JavaScript, in una pagina web, non può leggere o scrivere in qualsiasi file nell'hard disk, né copiare o eseguire programmi. Non ha accesso diretto alle funzioni del sistema operativo.

I moderni browser gli consentono di lavorare con i file, sempre con un accesso limitato e comunque solo se il comando proviene da utente, come il “dropping” di un file nella finestra del browser, o con la selezione tramite il tag `<input>`.

Ci sono anche funzionalità che consentono di interagire con la camera/microfono e altri dispositivi, ma in ogni caso richiedono il permesso esplicito dell'utente. Quindi una pagina con

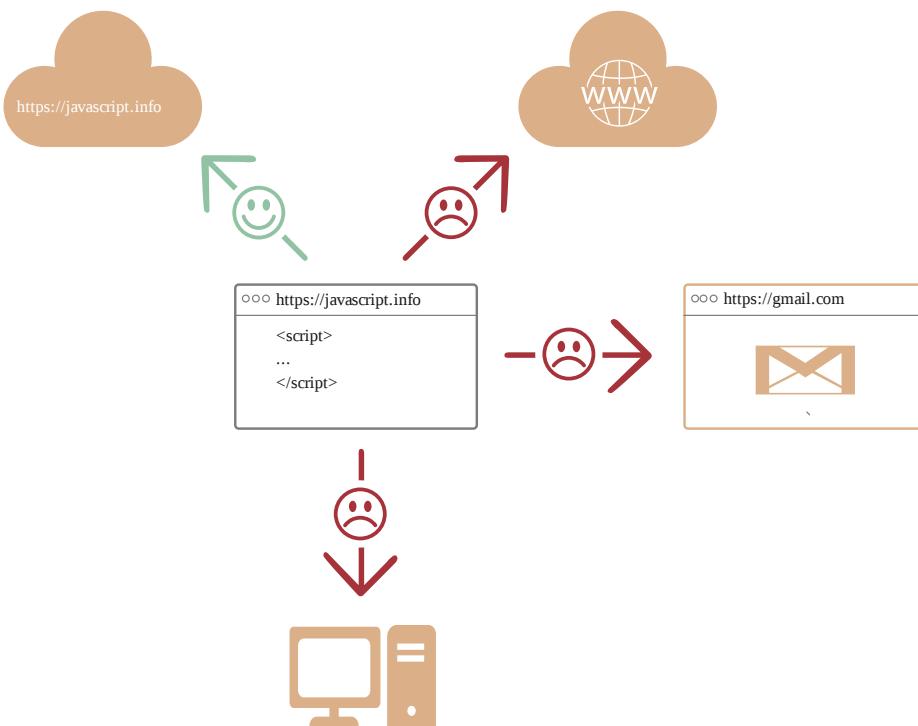
JavaScript abilitato non può attivare la web-cam di nascosto, osservare i nostri comportamenti e inviare informazioni alla CIA ↗.

- Pagine o schede diverse generalmente non sono a conoscenza dell'esistenza delle altre. In certi casi, tuttavia, può capitare; ad esempio quando una finestra ne apre un'altra tramite JavaScript. Ma anche in questo caso, il codice JavaScript non può accedere all'altra pagina se non appartiene allo stesso sito (stesso dominio, protocollo o porta).

Questa viene definita la “Same Origin Policy” (“Politica di Appartenenza alla Stessa Origine”). Per poter aggirare questo limite, *entrambe le pagine* devono contenere uno speciale codice JavaScript che consente di gestire lo scambio di dati.

Questa limitazione è sempre dovuta alla sicurezza dell'utente. Una pagina proveniente da `http://anysite.com` che è stata aperta da un utente, ad esempio, non deve essere in grado di accedere ad un'altra scheda del browser con l'URL `http://gmail.com` e rubarne le informazioni.

- JavaScript può facilmente comunicare con il server da cui la pagina proviene. Ma la sua abilità di ricevere dati da altri siti/domini è limitata. Sebbene sia possibile, sono richieste esplicite autorizzazioni (passate tramite HTTP headers) dall'indirizzo remoto. Ancora una volta, una limitazione dovuta alla sicurezza.



Queste limitazioni non si pongono se JavaScript viene eseguito fuori dal browser, ad esempio in un server. I browser moderni permettono l'installazione di plugin ed estensioni che consentono di estendere vari permessi.

Cosa rende JavaScript unico?

Ci sono almeno *tre* cose che rendono JavaScript così unico:

- Completa integrazione con HTML/CSS.
- Operazioni semplici vengono eseguite semplicemente.

- Supportato dai maggiori browser ed integrato di default.

JavaScript è l'unica tecnologia in ambiente browser che combina queste tre caratteristiche.

Questo rende JavaScript unico. Ed è il motivo per cui è lo strumento più diffuso per creare interfacce web.

Quando si ha in programma di imparare una nuova tecnologia, è fondamentale verificare le sue prospettive. Quindi diamo uno sguardo alle nuove tendenze che includono nuovi linguaggi e tecnologie.

Linguaggi “oltre” JavaScript

La sintassi di JavaScript non soddisfa le necessità di tutti. Alcune persone necessitano di caratteristiche differenti.

Questo è prevedibile, poiché i progetti e i requisiti sono diversi da persona a persona.

Recentemente, per questo motivo, sono nati molti nuovi linguaggi che vengono *convertiti* in JavaScript prima di essere eseguiti nel browser.

Gli strumenti moderni rendono la conversione molto veloce e pulita, consentendo agli sviluppatori di programmare in un altro linguaggio e di auto-convertirlo *under the hood*.

Esempi di alcuni linguaggi:

- [CoffeeScript ↗](#) è un linguaggio che introduce una sintassi semplificata che consente di scrivere codice più leggibile. Amato dagli sviluppatori provenienti da Ruby.
- [TypeScript ↗](#) si occupa di aggiungere la “tipizzazione”, per semplificare lo sviluppo e supportare sistemi più complessi. È stato sviluppato da Microsoft.
- [Flow ↗](#) anche esso aggiunge la tipizzazione dei dati, ma in un modo differente. Sviluppato da Facebook.
- [Dart ↗](#) è un linguaggio autonomo che possiede il suo motore, che esegue in ambienti esterni al browser (come mobile apps). È stato introdotto da Google come alternativa a JavaScript, ma attualmente i browser richiedono la conversione in JavaScript, proprio come i precedenti.
- [Brython ↗](#) è un *transpiler*, scritto in Python, che consente di scrivere applicazioni in quest'ultimo senza utilizzare JavaScript.
- [Kotlin ↗](#) è un moderno, conciso e sicuro linguaggio di programmazione mirato ai browsers o a Node.

Ce ne sono molti altri. Ovviamente, per comprendere cosa stiamo facendo, se utilizziamo uno di questi linguaggi dovremmo altresì conoscere JavaScript.

Riepilogo

- JavaScript è stato creato specificamente per i browser, ma attualmente viene utilizzato con efficacia in molti altri ambienti.
- Attualmente, per quanto riguarda lo sviluppo del web, JavaScript si trova in una posizione unica grazie ad una completa integrazione con HTML/CSS.

- Ci sono molti linguaggi che possono essere “convertiti” in JavaScript; essi provvedono le stesse funzionalità e risolvono gli stessi problemi. E’ fortemente consigliato di leggere brevemente le funzionalità di alcuni di essi, dopo aver studiato e compreso JavaScript.

Manuali e Specifiche

Questo libro è un *tutorial*. L’obiettivo è quello di aiutarti ad apprender il linguaggio gradualmente. Una volta che avrai familiarizzato con le basi avrai bisogno di ulteriori risorse.

Specifiche

[La specifica ECMA-262 ↗](#) contiene informazioni più dettagliate, approfondite e formalizzate riguardanti JavaScript. E’ la definizione stessa del linguaggio.

Iniziare a studiare dalla specifica può risultare difficile. Se avete bisogno di una fonte affidabile e formale riguardante i dettagli del linguaggio, la specifica è il posto in cui cercare. Ma non è una risorsa comoda da consultare per i problemi di tutti i giorni.

Ogni anno viene rilasciata una nuova specifica. Di queste pubblicazioni, è possibile trovare l’ultima bozza a [https://tc39.es/ecma262/ ↗](https://tc39.es/ecma262/).

Per leggere delle più recenti caratteristiche, incluse quelle considerate “quasi standard” (definite “stage 3”), potete consultare [https://github.com/tc39/proposals ↗](https://github.com/tc39/proposals).

Inoltre, se state sviluppando in ambiente browser, ci sono ulteriori specifiche che verranno analizzate nella [seconda parte](#) del tutorial.

Manuali

- **MDN (Mozilla) JavaScript Reference** è il manuale principale, corredata di spiegazioni teoriche, esempi ed altre informazioni utili. E’ ottimo per avere informazioni dettagliate riguardo le funzioni e altre caratteristiche del linguaggio.

Può essere consultato a [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference ↗](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference).

Tuttavia, spesso è meglio fare una ricerca su internet. E’ sufficiente cercare “MDN”, seguito dal termine da ricercare, e.g. [https://google.com/search?q=MDN+parseInt ↗](https://google.com/search?q=MDN+parseInt) per ricercare la funzione `parseInt`, oppure frasi come “RegExp MSDN” o “RegExp MSDN jscript”.

Può essere consultato al link [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference ↗](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference).

- **MSDN** – Manuale Microsoft con molte informazioni, tra cui su JavaScript (a cui viene fatto riferimento con il termine JScript). Se si ha bisogno di ottenere qualche informazione specifica per Internet Explorer, meglio consultare la guida: [http://msdn.microsoft.com/ ↗](http://msdn.microsoft.com/).

Tabelle di compatibilità

JavaScript è un linguaggio che muta costantemente, con nuove funzionalità che vengono aggiunte regolarmente.

Per verificare il loro supporto da parte dei browser, si possono consultare:

- <http://caniuse.com> – per le tabelle di supporto di ogni caratteristica, ad esempio per visualizzare le funzioni di crittografia: <http://caniuse.com/#feat=cryptography>.
- <https://kangax.github.io/compat-table> – una tabella con le caratteristiche del linguaggio e i motori che le supportano.

Tutte le risorse elencate finora sono utili nello sviluppo di tutti i giorni, in quanto contengono ottime informazioni riguardo ai dettagli del linguaggio, il loro supporto ecc.

Ti consiglio quindi di ricordartele (in alternativa puoi consultare questa pagina), nel caso dovessi avere bisogno di informazioni dettagliate riguardo a qualche caratteristica particolare.

Code editor

Un code editor è il posto in cui i programmatori passano la maggior parte del loro tempo.

Ci sono due principali tipi di code editor: IDE ed editor semplici. Molte persone si trovano bene a sceglierne uno per entrambe le categorie.

IDE

Il termine [IDE](#) (Integrated Development Environment) descrive un potente editor che copre lo sviluppo dell'intero progetto. Come anche il nome suggerisce, non è un semplice editor, ma un "ambiente di sviluppo" scalabile, con molte funzionalità.

Un IDE carica il progetto (possono essere molti file), consente la navigazione tra i file, fornisce il completamento automatico basandosi sull'intero progetto (non sul singolo file), può essere integrato con sistemi di gestione di versione (come [git](#)), un ambiente dedicato al test e altre funzionalità a livello del progetto.

Se non hai ancora considerato di scegliere un IDE, dai un'occhiata a queste alternative:

- [Visual Studio Code](#) (*cross-platform*, gratuito).
- [WebStorm](#) (*cross-platform*, a pagamento).

Per Windows, c'è anche l'editor "Visual Studio", da non confondere con "Visual Studio Code". "Visual Studio" è un potente editor (a pagamento) disponibile solo per Windows, ottimo per le piattaforme .NET. E' disponibile anche una versione gratuita: ([Visual Studio Community](#)).

Molti IDE sono a pagamento, ma offrono un periodo di prova gratuito. Solitamente il loro costo è trascurabile se paragonato allo stipendio di una sviluppatore qualificato; è quindi importante scegliere il migliore in base alle proprie esigenze.

Editor semplici

Gli "editor Semplici" non sono potenti come gli IDE ma sono molto veloci, eleganti e semplici.

Sono principalmente utilizzati per aprire un file e modificarlo rapidamente.

La principale differenza tra gli editor semplici e un IDE è che quest'ultimo lavora a vari livelli del progetto, carica molti più dati quando viene aperto, analizza la struttura del progetto e così via. Un editor semplice è molto più veloce poiché necessita solo del file.

In pratica, tuttavia, gli editor semplici possono avere molti plugin, tra cui la sintassi a livello directory e l'autocompletamento, quindi non ci sono delle differenze ben definite tra un editor semplice e un IDE.

Meritano attenzione le seguenti opzioni:

- [Atom ↗](#) (*cross-platform*, gratuito).
- [Visual Studio Code ↗](#) (*cross-platform*, gratuito).
- [Sublime Text ↗](#) (*cross-platform*, con prova gratuita).
- [Notepad++ ↗](#) (Windows, gratuito).
- [Vim ↗](#) e [Emacs ↗](#) sono particolarmente carini se si sanno utilizzare.

Non intestarditevi

Gli editor elencati sopra sono quelli che io e i miei amici, che considero buoni sviluppatori, abbiamo utilizzato senza problemi per molto tempo.

Ci sono altri grandi editor nel nostro grande mondo. Scegli quello che più ti si addice.

La scelta di un editor, come pure di altri strumenti, è individuale e dipende dai progetti, dalle abitudini e preferenze personali.

Developer console

Il codice è incline a contenere errori. E' molto probabile che tu commetta errori... Di cosa sto parlando? Sicuramente commetterai errori, sempre che tu sia umano, e non un [robot ↗](#).

In un browser però, di default l'utente non può vedere gli errori. Quindi, se qualcosa non funziona nello script, non saremo in grado di capire quale sia il problema e sistemarlo.

Per poter visualizzare gli errori e ricevere altre informazioni utili riguardo gli script, i browser integrano degli "strumenti di sviluppo", in inglese "developer tools" o più semplicemente "DevTools".

Molti sviluppatori preferiscono utilizzare Chrome o Firefox poiché questi browser incorporano i migliori strumenti per lo sviluppo. Anche gli altri browser hanno gli strumenti per lo sviluppo, talvolta con caratteristiche speciali, ma più che altro inseguono le caratteristiche di Chrome e Firefox. In genere gli sviluppatori hanno un browser "preferito" e utilizzano gli altri solo quando un problema è specifico di quel browser.

Gli strumenti per lo sviluppo sono molto potenti e possiedono molte funzionalità. Per iniziare, dobbiamo capire come accedervi, come individuare gli errori e come eseguire comandi JavaScript.

Google Chrome

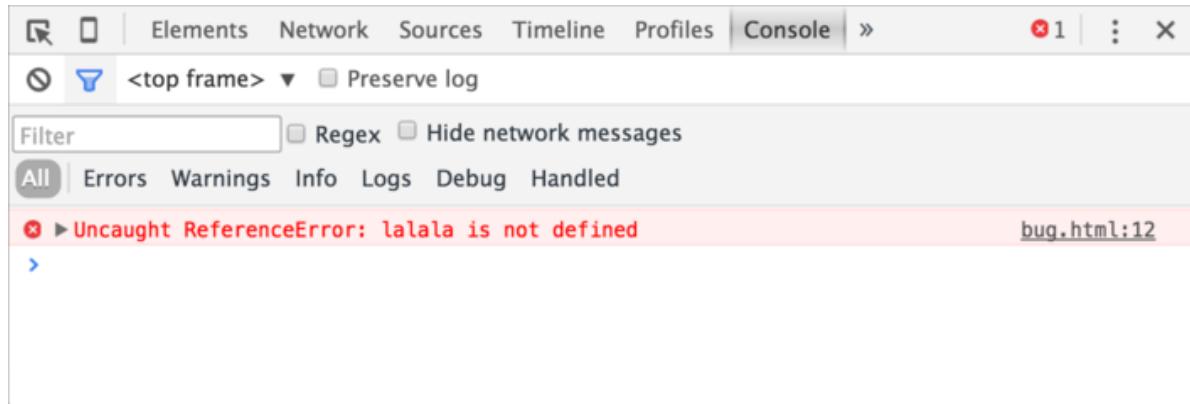
Apri la pagina [bug.html](#).

C'è un errore nel codice JavaScript. E' nascosto agli occhi di un normale utente, quindi dobbiamo aprire gli strumenti di sviluppo per trovarlo.

Premi `F12`, oppure, se sei su Mac, utilizza `Cmd+Opt+J`.

Gli strumenti di sviluppo, di default, si apriranno nella scheda Console.

Assomiglierà a qualcosa di simile a questo:



Il look esatto degli strumenti di sviluppo dipenderà dalla tua versione di Chrome. Nel tempo potrebbe cambiare un po', ma dovrebbe essere comunque molto simile.

- Qui possiamo notare il messaggio d'errore in rosso. In questo caso, lo script contiene il comando "lalala" non riconosciuto.
- Sulla destra, c'e il link cliccabile della sorgente `bug.html:12` con il numero della linea in cui si è verificato l'errore.

Sotto il messaggio d'errore, c'e il simbolo blu `>`. Questo indica la "riga di comando" in cui possiamo digitare dei comandi JavaScript. Premendo `Enter` il comando viene (`Shift+Enter` per inserire comandi multi-linea).

Ora possiamo vedere gli errori, e questo è sufficiente per iniziare. Ritorneremo sugli strumenti di sviluppo più avanti e analizzeremo il debugging più in profondità nel capitolo [Debugging in the browser](#).

i Input multi-riga

Di solito, quando inseriamo una riga di codice nella console e premiamo il tasto `Enter`, questa viene eseguita.

Per inserire più righe premi `Shift+Enter`, in questo modo puoi inserire lunghe porzioni di codice Javascript.

Firefox, Edge, and others

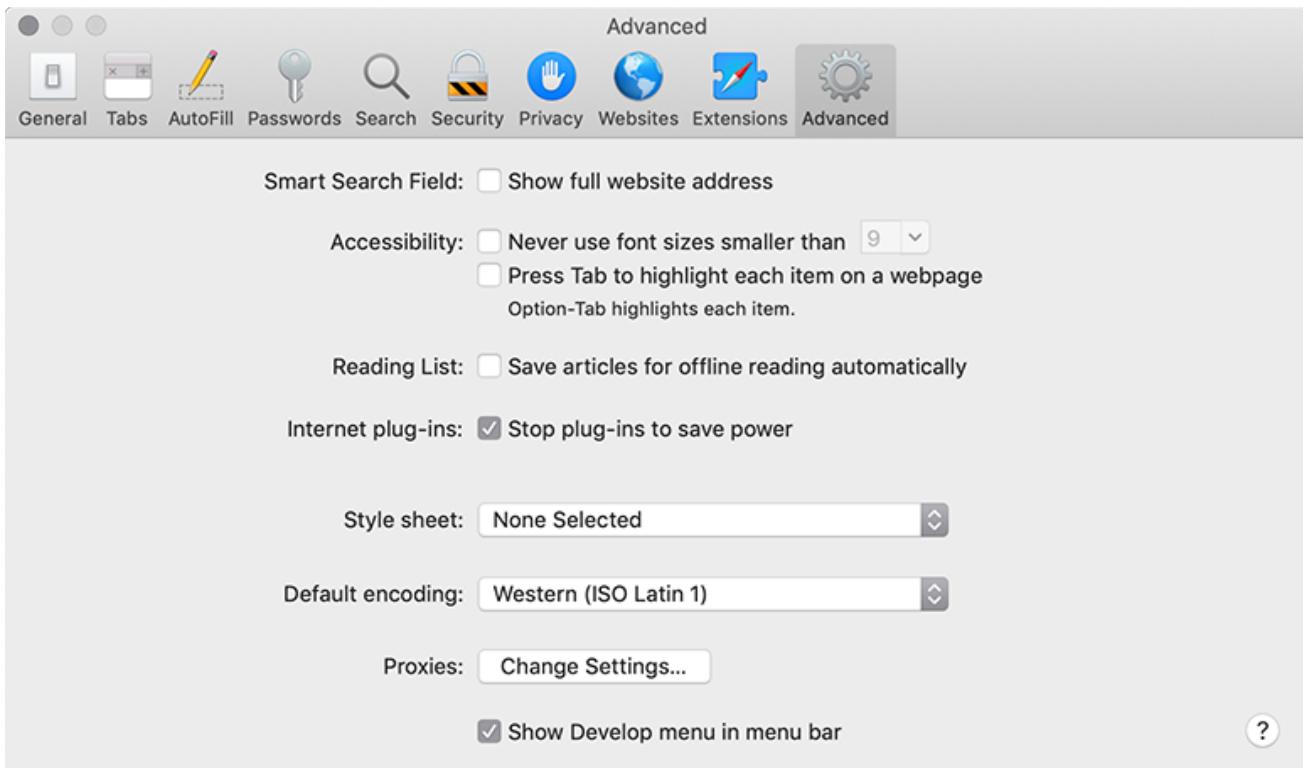
Molti altri browser utilizzano `F12` per aprire gli strumenti di sviluppo.

Anche l'aspetto è molto simile. Quando avrai imparato come utilizzare uno di questi strumenti (puoi iniziare con quelli di Chrome), potrai facilmente utilizzarne anche gli altri.

Safari

Safari (Mac browser, non supportato da Windows/Linux) è un pò speciale in questo ambito. E' necessario attivare prima il "Menu di Sviluppo".

Apri le Impostazioni e vai sul pannello "Avanzate". In basso troverai un'opzione da spuntare:



Adesso tramite `Cmd+Opt+C` puoi attivare e disattivare la console. Inoltre noterai che un nuovo menu “Sviluppo” è apparso. Esso contiene molti comandi e opzioni.

Riepilogo

- Gli strumenti di sviluppo (Developer Tools o DevTools) ci consentono di individuare gli errori, eseguire comandi, esaminare variabili e molto altro.
- Possono essere attivati con `F12` nella maggior parte dei browser in Windows e Linux. Chrome su Mac `Cmd+Opt+J`, Safari: `Cmd+Opt+C` (avendolo precedentemente abilitato).

Ora in nostro ambiente di sviluppo è pronto. Nella prossima sezione inizieremo ad analizzare JavaScript.

Le basi JavaScript

Impareremo le basi dello sviluppo di uno script.

Hello, world!

Il seguente tutorial tratta del core (nucleo) di JavaScript, il quale è indipendente dalla piattaforma.

Abbiamo bisogno di un ambiente di lavoro per eseguire i nostri script, e il fatto che questo tutorial sia online, rende il browser un ottima scelta. Cercheremo di mantenere al minimo l'utilizzo dei comandi specifici per browser (come `alert`), così non dovrà perdere la testa se deciderai di spostarti in altri ambienti (come Node.js). In ogni caso, ci concentreremo sulle caratteristiche JavaScript specifiche per il browser nella [prossima parte](#) del tutorial.

Quindi prima di tutto, vediamo come inserire uno script in una pagina web. Per ambienti server-side (come Node.js), è sufficiente eseguirli con un comando come `"node my.js"`.

Il tag “script”

I programmi JavaScript possono essere inseriti in qualunque parte di un documento HTML, con l'utilizzo del tag `<script>`.

Ad esempio:

```
<!DOCTYPE HTML>
<html>

<body>

<p>Before the script...</p>

<script>
  alert( 'Hello, world!' );
</script>

<p>...After the script.</p>

</body>

</html>
```

Il tag `<script>` contiene codice JavaScript che viene automaticamente eseguito quando il browser processa il tag.

La segnatura moderna

Il tag `<script>` ha un paio di attributi che vengono utilizzati raramente, ma è comunque possibile trovarli nei vecchi codici:

L'attributo `type` : `<script type=...>`

Il precedente standard HTML, HTML4, richiedeva che lo script avesse una proprietà `type`. Solitamente era `type="text/javascript"`. Ora non è più richiesto. Inoltre, lo standard attuale HTML, ha completamente cambiato il suo significato. Ora può essere utilizzato per i moduli JavaScript. Ma questo è un argomento avanzato, parleremo dei moduli più avanti nel tutorial.

L'attributo `language` : `<script language=...>`

Questo attributo aveva lo scopo di mostrare il linguaggio utilizzato dallo script. Ora questo linguaggio non ha più molto senso, poiché JavaScript è il linguaggio utilizzato di default. Quindi non ha più senso utilizzarlo.

I commenti prima e dopo gli script.

Nei vecchi libri e tutorial, potrete trovare commenti all'interno del tag `<script>`, come questo:

```
<script type="text/javascript"><!--
  ...
--></script>
```

Questo trucco non viene più utilizzato. Questi commenti avevano lo scopo di nascondere il codice JavaScript ai vecchi browser che non erano in grado di elaborare il tag `<script>`. Poiché i browser rilasciati negli ultimi 15 anni non hanno più questo problema, questo tipo di commenti possono aiutarti ad identificare codici molto vecchi.

Script esterni

Se abbiamo molto codice JavaScript, possiamo inserirlo in un file separato.

Il file dello script viene integrato nel codice HTML tramite l'attributo `src`:

```
<script src="/path/to/script.js"></script>
```

Questo `/path/to/script.js` è il percorso assoluto al file che contiene lo script a partire dalla root del sito. Ad esempio, `src="script.js"` significherebbe un file `"script.js"` che si trova nella cartella corrente.

E' anche possibile fornire un percorso relativo a partire dalla pagina corrente. Per esempio `src="script.js"` significa che il file `"script.js"` si trova nella cartella corrente.

Possiamo anche fornire un URL. Ad esempio:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js"></script>
```

Per integrare più script, possiamo utilizzare più volte il tag:

```
<script src="/js/script1.js"></script>
<script src="/js/script2.js"></script>
...

```

Da notare:

Come regola da seguire, solo gli script molto semplici vanno inseriti all'interno dell'HTML. Quelli più complessi vanno inseriti in file separati.

Il beneficio di inserire gli script in file separati è che il browser andrà a scaricarli e li memorizzerà nella sua [cache](#).

Così facendo, le altre pagine che vorranno utilizzare lo stesso script lo preleveranno dalla cache invece che riscaricarlo. Quindi il file verrà scaricato una sola volta.

Questo risparmierà traffico e renderà il caricamento delle pagine più veloce.

 **Se `src` è impostato, il contenuto all'interno di `script` verrà ignorato.**

Quindi un tag `<script>` non può avere sia `src` che codice incorporato.

Questo non funziona:

```
<script src="file.js">
  alert(1); // il contenuto viene ignorato, perché src è impostato
</script>
```

Dobbiamo scegliere fra le due possibilità: script esterno `<script src="...">` o il semplice tag `<script>` con all'interno il codice.

L'esempio precedente può essere diviso in due script:

```
<script src="file.js"></script>
<script>
  alert(1);
</script>
```

Riepilogo

- Possiamo usare il tag `<script>` per aggiungere codice JavaScript alla pagina.
- Gli attributi `type` e `language` non sono più richiesti.
- Uno script in un file esterno può essere inserito con `<script src="path/to/script.js"></script>`.

C'è ancora molto da imparare riguardo gli script browser e la loro interazione con le pagine web. Ma tenete a mente che questa parte del tutorial è dedicata al linguaggio JavaScript, quindi non dobbiamo distrarci da questo obiettivo. Andremo ad utilizzare il browser come piattaforma in cui eseguire JavaScript, che è molto comodo, ma è solo uno dei tanti modi.

✓ Esercizi

Mostra un alert

importanza: 5

Crea un pagina che mostra un messaggio "I'm JavaScript".

Provalo tramite la sandbox, oppure sul tuo hard disk, non ha importanza, assicurati solo che funzioni.

[Demo in una nuova finestra ↗](#)

[Alla soluzione](#)

Mostra un alert con uno script esterno

importanza: 5

Prendi la soluzione dell'esercizio precedente [Mostra un alert](#). Modificalo estraendo il contenuto dello script, inseriscilo in un file esterno `alert.js`, salvato nella stessa cartella.

Apri la pagina ed assicurati che l'alert venga visualizzato.

[Alla soluzione](#)

Struttura del codice

La prima cosa che studieremo riguarda la struttura del codice.

Istruzioni

Le istruzioni sono dei costrutti sintattici e comandi che permettono di eseguire azioni.

Abbiamo già visto un'istruzione `alert('Hello, world!')`, che mostra il messaggio "Hello world!".

All'interno del codice possiamo avere tutte le istruzioni che desideriamo. Le istruzioni possono essere separate da un punto e virgola.

Ad esempio, qui dividiamo il messaggio in due alert:

```
alert('Hello'); alert('World');
```

Di solito ogni istruzione viene scritta in una riga separata per rendere il codice molto più leggibile:

```
alert('Hello');
alert('World');
```

Punto e virgola

Un punto e virgola può essere omesso nella maggior parte dei casi quando si interrompe una riga.

Questo funzionerà ugualmente:

```
alert('Hello')
alert('World')
```

In questo caso, JavaScript interpreta la fine della riga come un punto e virgola "implicito". Viene anche chiamata [inserimento automatico del punto e virgola ↗](#).

In molti casi la nuova riga viene interpretata come un punto e virgola implicito. Ma "in molti casi" non significa "sempre"!

Ci sono casi in cui la nuova riga non implica una punto e virgola, per esempio:

```
alert(3 +  
1  
+ 2);
```

Il codice stampa `6` perché, in questo caso, JavaScript non inserisce un punto e virgola. E' abbastanza ovvio che se la riga finisce con un `"+"`, allora è un "espressione incompleta", quindi il punto e virgola sarebbe errato. Per questo, nell'esempio sopra, tutto funziona come dovrebbe.

Ma ci sono casi in cui JavaScript "fallisce" nell'interpretare un punto e virgola, dove invece sarebbe necessario.

Gli errori di questo tipo sono molto difficili da trovare e sistemare.

Un esempio di errore

Se sei curioso di vedere un esempio concreto di questo tipo di errore, dai un occhiata al seguente codice:

```
alert("Hello");  
[1, 2].forEach(alert);
```

Non c'e bisogno di pensare al significato delle parentesi `[]` e al `forEach`. Li studieremo più avanti, per ora è sufficiente sapere il risultato: mostrerà `1` e poi `2`.

Adesso andiamo a rimuovere il punto e virgola dopo `alert`:

```
alert("Hello")  
[1, 2].forEach(alert);
```

La differenza rispetto al codice precedente è solo un carattere, il punto e virgola al termine della prima line è sparito.

Se eseguiamo il codice, verrà mostrato solo il primo `Hello` (ci sarà un errore, ma per visualizzarlo bisogna aprire la console). Non verranno più mostrati i numeri.

Questo perché JavaScript non inserisce il punto e virgola prima della parentesi quadra `[...]`. Quindi il codice viene interpretato come un singolo comando.

Ecco come il motore interpreta il codice:

```
alert("Hello")[1, 2].forEach(alert);
```

Sembra strano, vero? Questo comportamento in questo caso è errato. E' necessario mettere un punto e virgola dopo `alert` affinché il codice funzioni correttamente.

Questo accade anche in altre situazioni.

E' consigliato quindi, di inserire il punto e virgola fra ogni istruzione, anche se vengono scritte in righe diverse. Questa è una regola largamente adottata dalla community. Ripetiamolo

nuovamente – è possibile omettere il punto e virgola la maggior parte delle volte. Ma è più sicuro – specialmente per un novizio – inserirlo al termine di ogni istruzione.

Commenti

Con il passare del tempo, i programmi sono diventati sempre più complessi. Ed è diventato necessario aggiungere *commenti* per poter descrivere i comportamenti del codice.

I commenti possono essere messi in qualsiasi punto dello script. Infatti non hanno alcun effetto sull'esecuzione del codice, poiché il motore JavaScript semplicemente li ignora.

I commenti su una singola linea si inseriscono con due caratteri di slash `//`.

Il resto della linea è il commento. Può occupare un'intera riga oppure essere posta in seguito ad un'istruzione.

Vediamo un esempio:

```
// Questo commento occupa un'intera riga
alert('Hello');

alert('World'); // Questo commento segue un'istruzione
```

I commenti multilinea incominciano con un singolo carattere di slash ed un asterisco `/*` e finiscono con un asterisco ed un carattere di slash `*/`.

Come nell'esempio:

```
/* Un esempio con due messaggi.
Questo è un commento multilinea.
*/
alert('Hello');
alert('World');
```

Il contenuto dei commenti viene ignorato, quindi se inseriamo codice al suo interno `/* ... */` non verrà eseguito.

Qualche volta può essere utile per bloccare temporaneamente qualche porzione di codice:

```
/* Commentiamo il codice
alert('Hello');
*/
alert('World');
```

Usa le scorciatoie da tastiera!

In molti editor una linea di codice può essere commentata con la combinazione da tastiera dei tasti `Ctrl+/` e una combinazione simile a `Ctrl+Shift+/` – per i commenti multilinea (selezionate prima una parte di codice e poi utilizzate la combinazione di tasti). Su Mac dovrebbe funzionare la combinazione `Cmd` al posto di `Ctrl`.

I commenti annidati non sono supportati!

Non si possono inserire `/* ... */` all'interno di altri `/* ... */`.

Questo codice genererebbe un errore:

```
/*
  /* commento annidato ?!?
*/
alert( 'World' );
```

Non abbiate paura di utilizzare i commenti nel codice.

I commenti aumenteranno il peso finale dello script, ma questo non sarà un problema. Ci sono tantissimi strumenti che possono minimizzare("minify") il codice prima di pubblicarlo nel server. Questi strumenti rimuovono i commenti, quindi non appariranno nello script che verrà eseguito. Perciò i commenti non hanno alcun effetto negativo sul codice prodotto.

Inoltre più avanti nel tutorial ci sarà un capitolo [Stile di programmazione](#) che illustrerà come scrivere commenti in maniera ottimale.

Le tecniche moderne, "use strict"

Per molto tempo JavaScript si è evoluto senza problemi di compatibilità. Nuove funzionalità venivano aggiunte al linguaggio, ma quelle vecchie non cambiavano.

Questo ha consentito al vecchio codice di non diventare obsoleto. Ma lo svantaggio è stato che così facendo gli errori e le decisioni imperfette fatte dai creatori di JavaScript, rimarranno nel linguaggio per sempre.

Così è stato fino al 2009 quando è apparsa ECMAScript 5 (ES5). Ha aggiunto nuove funzionalità al linguaggio e ne ha modificate alcune già esistenti. Per far sì che il vecchio codice continui a funzionare, molte modifiche vengono disattivate di default. Diventa quindi necessario abilitarle esplicitamente con la direttiva `"use strict"`.

"use strict"

La direttiva è simile ad una stringa: `"use strict"` o `'use strict'`. Quando viene inserita all'inizio dello script, da quel momento l'intero script funziona nello stile "moderno".

Per esempio:

```
"use strict";

// questo codice viene eseguito secondo gli standard moderni
...;
```

Impareremo presto le funzioni (un modo per raggruppare le istruzioni).

Guardando avanti, notiamo che `"use strict"` può essere applicato all'inizio di una funzione (la maggior parte) piuttosto che all'intero script. La modalità strict sarà quindi attiva solo all'interno

di quella funzione. Solitamente si utilizza nell'intero script.

⚠ Assicurati che "use strict" sia all'inizio

Assicurati `"use strict"` sia all'inizio dello script, altrimenti la modalità script non verrà abilitata.

Qui vediamo un esempio in cui non verrà attivata la modalità strict:

```
alert("some code");
// "use strict" qui sotto viene ignorato -- la dichiarazione deve stare sempre in cima

"use strict";

// strict mode non è attiva
```

Solo i commenti possono apparire prima di `"use strict"`.

⚠ Non c'e nessun modo per annullare `use strict`

Non esiste nessuna direttiva `"no use strict"` o simile, che possa riportare lo script alla vecchia modalità.

Una volta abilitata la modalità strict, non c'e ritorno.

Browser console

In futuro, quando utilizzerete la console integrata in un browser, dovete tenere a mente che di default non vale `use strict`.

In certe situazioni, `use strict` fa veramente la differenza, quindi potreste ottenere dei risultati indesiderati.

Potete provare con `Shift+Enter` per inserire più righe di codice, con `use strict` in cima, come nell'esempio:

```
'use strict'; <Shift+Enter for a newline>
// ...your code
<Enter to run>
```

Funziona nella maggior parte dei browser, tra cui Firefox e Chrome.

Nel caso in cui non funzioni, la miglior strada da seguire è quella di assicurarsi che `use strict` venga inserito, in questo modo:

```
"use strict";
// codice
...
```

Dovremmo utilizzare “use strict”?

Può sembrare una domanda ovvia, ma non lo è.

Potrebbero consigliarvi di iniziare tutti gli script con `"use strict"` ... Ma sapete cosa c'è di bello?

JavaScript moderno supporta le “classi” e i “moduli” – delle strutture avanzate del linguaggio (a cui arriveremo più avanti), che abilitano `use strict` in automatico. Quindi non è necessario inserire la direttiva `"use strict"` se utilizziamo queste funzionalità.

Quindi, per ora `"use strict"`; è un ospite ben accetto nei vostri script. Più avanti, quando il vostro codice sarà suddiviso in classi e moduli, potrete ometterlo.

Al momento, è sufficiente una conoscenza generica di `use strict`.

Nei prossimi capitoli, quando impareremo nuove funzionalità del linguaggio, vedremo più nel dettaglio le differenze tra la “strict mode” e la “vecchia modalità”. Fortunatamente, non sono molte e sono anche molto utili.

Tutti gli esempi in questo tutorial assumono che la “strict mode” sia attiva, tranne in alcuni rarissimi casi (in cui sarà specificato).

Variabili

La maggior parte delle volte, le applicazioni JavaScript necessitano di lavorare con informazioni. Vediamo due esempi:

1. Un negozio online – le informazioni possono riguardare i beni venduti e il carrello.
2. Un'applicazione di messaggistica – le informazioni possono riguardare utenti, messaggi e molto altro.

Le variabili vengono utilizzate per memorizzare informazioni.

Variabile

Una [variabile ↗](#) è uno “spazio di memoria con nome” utilizzato per salvare dati. Possiamo usare le variabili per memorizzare informazioni extra, visitatori e altri dati.

Per creare una variabile in JavaScript, dobbiamo utilizzare la parola chiave `let`.

L’istruzione sotto crea (in altre parole: *dichiara*) una variabile identificata dal nome “messaggio”:

```
let message;
```

Adesso possiamo inserirci dei dati utilizzando l’operatore di assegnazione `=`:

```
let message;  
  
message = 'Hello'; // memorizzazione della stringa
```

La stringa è adesso salvata nell'area di memoria associata alla variabile. Possiamo accedervi utilizzando il nome della variabile:

```
let message;  
message = 'Hello!';  
  
alert(message); // mostra il contenuto della variabile
```

Per essere precisi, potremmo unire la dichiarazione e l'assegnazione in una singola riga:

```
let message = 'Hello!'; // definisce la variabile e gli assegna il valore  
  
alert(message); // Hello!
```

Possiamo anche dichiarare più variabili in una riga:

```
let user = 'John', age = 25, message = 'Hello';
```

Questo potrebbe risultare più breve, ma è sconsigliato. Per mantenere una migliore leggibilità è meglio dichiarare solamente una variabile per riga.

L'alternativa a più righe è un po più lunga, ma più facile da leggere:

```
let user = 'John';  
let age = 25;  
let message = 'Hello';
```

Alcune persone scrivono variabili multiple in questo modo:

```
let user = 'John',  
age = 25,  
message = 'Hello';
```

...O anche con la virgola su nuova riga:

```
let user = 'John'  
, age = 25  
, message = 'Hello';
```

Tecnicamente, tutte queste varianti fanno la stessa cosa. Quindi è una questione di gusto personale ed estetico.

i `var` piuttosto che `let`

Nei vecchi script potresti trovare: `var` piuttosto che `let`:

```
var message = 'Hello';
```

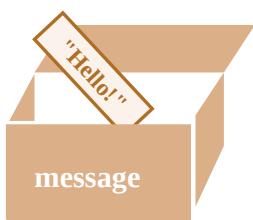
La parola chiave `var` è quasi la stessa cosa di `let`. Dichiara comunque una variabile, ma in un maniera leggermente diversa, “vecchio stile”.

Ci sono delle sottili differenze tra `let` e `var`, ma per ora non hanno importanza. Le copriremo in dettaglio più avanti, nel capitolo [Il vecchio "var"](#).

Un'analogia con il mondo reale

Possiamo comprendere meglio il concetto di “variabile” se la immaginiamo come una scatola per dati, con appiccicata un’etichetta univoca.

Per esempio, la variabile `message` può essere immaginata come una scatola con etichetta “`message`” con il valore “`Hello!`” al suo interno:

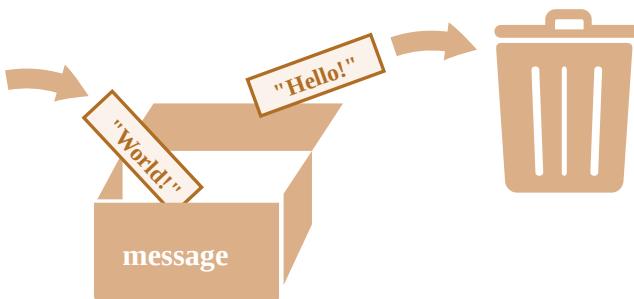


Possiamo inserire qualsiasi valore all'interno della scatola.

Possiamo anche cambiarlo. Il valore può cambiare tutte le volte che ne abbiamo bisogno:

```
let message;  
  
message = 'Hello!';  
  
message = 'World!'; // il valore è cambiato  
  
alert(message);
```

Quando il valore viene cambiato, il dato vecchio viene rimosso dalla variabile:



Possiamo anche dichiarare due variabili e copiare i dati da un all'altra.

```
let hello = 'Hello world!';

let message;

// copia 'Hello world' da hello in message
message = hello;

// ora le due variabili contengono gli stessi dati
alert(hello); // Hello world!
alert(message); // Hello world!
```

Dichiare una variabile più di una volta farà scattare un errore

Una variabile dovrebbe essere dichiarata una volta sola.

La ripetizione della dichiarazione di una stessa variabile porterà ad un errore:

```
let message = "This";

// 'let' ripetuto genererà un errore
let message = "That"; // SyntaxError: 'message' has already been declared
```

Quindi, dovremmo dichiarare una variabile una volta sola, e farne riferimento senza la parola chiave `let`.

Linguaggi funzionali

Può essere interessante sapere che esistono anche linguaggi di programmazione [funzionale](#) che vietano di cambiare il valore di una variabile. Per esempio, [Scala](#) o [Erlang](#).

In questo tipo di linguaggi, una volta che il valore viene memorizzato “dentro la scatola”, ci rimane per sempre. Se abbiamo bisogno di memorizzare qualcosa altro, il linguaggio ci forza a creare una nuova scatola (dichiarare una nuova variabile). Non possiamo quindi riutilizzare quelle vecchie.

Anche se potrebbero sembrare un po’ strano a prima vista, questi linguaggi sono veramente capaci di sviluppare grandi cose. Inoltre, ci sono certe situazioni come calcoli paralleli in cui questi limiti portano dei benefici. Studiare un linguaggio di questo tipo (anche se non abbiamo intenzione di utilizzarlo a breve) è consigliato per allargare le proprie conoscenze.

Nomi delle variabili

In JavaScript ci sono solo due limitazioni per il nome delle variabili:

1. Il nome deve contenere solo lettere, numeri, simboli `$` e `_`.
2. Il primo carattere non può essere un numero.

Esempi di nomi validi:

```
let userName;
```

```
let test123;
```

Quando il nome contiene più parole, viene utilizzato il [camelCase](#). La logica è: le parole vanno una dopo l'altra, ogni parola inizia con lettere maiuscola: `myVeryLongName`.

Una cosa interessante è che – il simbolo del dollaro `'$'` e l'underscore `'_'` possono essere utilizzati nei nomi. Sono dei semplici simboli, come le lettere, senza alcun significato speciale.

Ad esempio questi nomi sono validi:

```
let $ = 1; // dichiarata una variabile con nome "$"
let _ = 2; // qui una variabile con nome "_"

alert($ + _); // 3
```

Questi invece non lo sono:

```
let 1a; // non può cominciare con una stringa

let my-name; // '-' non è consentito nei nomi
```

La questione delle lettere

Le variabili `apple` ed `AppLE` sono distinte.

Le lettere non latine sono permesse, ma sono sconsigliate

E' possibile utilizzare qualsiasi alfabeto, compreso quello cirillico o addirittura i geroglifici:

```
let имя = '...';
let 我 = '...';
```

Tecnicamente, non ci sono errori, questo tipo di nomi sono permessi, ma la tradizione internazionale è di utilizzare l'alfabeto inglese per il nome delle variabili. Anche se stiamo scrivendo un piccolo script, questo potrebbe infatti avere una lunga vita. Persone di altre nazionalità potrebbero aver bisogno di leggerlo.

Nomi riservati

C'è una [lista di parole riservate ↗](#), che non possono essere utilizzate come nomi di variabili, perché vengono utilizzate dal linguaggio stesso.

Per esempio, le parole `let`, `class`, `return`, `function` sono riservate.

Questo codice provocherà un errore di sintassi:

```
let let = 5; // non è possibile chiamare una variabile "let", errore!
let return = 5; // nemmeno "return", errore!
```

Un assegnazione senza `use strict`

Normalmente, abbiamo bisogno di definire variabili prima di utilizzarle. Ma una volta, era possibile definire una variabile semplicemente assegnandogli un valore, senza `let`. Questo è ancora possibile se non utilizziamo `use strict`. E' necessario per mantenere la compatibilità con i vecchi script.

```
// da notare: non si utilizza "use strict" in questo esempio

num = 5; // la variabile "num" se non esiste già

alert(num); // 5
```

Questa però è una pessima pratica, che causerebbe un errore in strict mode:

```
"use strict";

num = 5; // errore: num non è definita
```

Costanti

Per dichiarare una variabile costante (immutabile), dobbiamo utilizzare `const` invece di `let`:

```
const myBirthday = '18.04.1982';
```

Le variabili dichiarate con `const` vengono chiamate “costanti”. Non possono cambiare valore. Se tentassimo di farlo verrebbe sollevato un errore:

```
const myBirthday = '18.04.1982';

myBirthday = '01.01.2001'; // errore, non è possibile riassegnare la costante!
```

Quando il programmatore è sicuro che il valore della variabile non cambierà mai, può utilizzare `const` per soddisfare questa esigenza, rendendolo così esplicito.

Le costanti maiuscole

Una pratica molto diffusa è di utilizzare le variabili costanti come alias di valori difficili da ricordare, e che sono noti prima dell'esecuzione.

Questo tipo di costanti vengono identificate con lettere maiuscole e underscore.

Come in questo esempio, creiamo delle costanti nel cosiddetto formato “web” (esadecimale):

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";

// ...quando abbiamo bisogno di prelevare un colore
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

Benefici:

- `COLOR_ORANGE` è più facile da ricordare di `"#FF7F00"`.
- E' più facile commettere errori scrivendo `"#FF7F00"` piuttosto che `COLOR_ORANGE`.
- Quando leggiamo il codice, `COLOR_ORANGE` è molto più significativo di `#FF7F00`.

Quando dovremmo utilizzare lettere maiuscole per una costante, e quando invece trattarle come normali variabili? Facciamo un pò di chiarezza.

Essere una “costante” significa che il valore non potrà mai cambiare. Ci sono costanti che sono note prima dell'esecuzione (come la codifica esadecimale del colore rosso), e ci sono quelle che vengono *calcolate* durante l'esecuzione, ma non cambieranno più dopo che gli sarà stato assegnato un valore.

Per esempio:

```
const pageLoadTime = /* tempo necessario da una pagina web per caricare */;
```

Il valore di `pageLoadTime` non è noto prima del caricamento della pagina, quindi viene trattato come una normale variabile. Ma rimane comunque una costante, perché non potrà più cambiare dopo che gli sarà stato assegnato un valore.

In altre parole, i nomi delle costanti in maiuscolo vengono utilizzati con variabili dal valore noto prima dell'esecuzione.

Dare i giusti nomi alle cose

Parlando di variabili, c'è un'altra cosa estremamente importante.

Il nome di una variabile dovrebbe sempre essere pulito, ovvio e descrittivo del suo contenuto.

Dare i giusti nomi alle variabili è una delle abilità più importanti (e difficili) nella programmazione. Una rapida occhiata ai nomi delle variabili può rivelare se il codice è stato scritto da un

principiante o da uno sviluppatore esperto.

In un progetto reale, la maggior parte del tempo lo si perde a modificare ed estendere del codice già esistente, piuttosto che riscriverne uno nuovo. E quando ritorneremo sul codice, dopo aver fatto qualcos'altro, sarà molto più facile trovare informazioni se sono ben descritte. In altre parole, quando le variabili utilizzano dei nomi efficaci.

Quindi è utile spendere del tempo a pensare il giusto nome per una variabile, prima di dichiararla. Questo approccio vi ripagherà.

Alcune regole da seguire:

- Utilizzare nomi leggibili da persone, come `userName` o `shoppingCart`.
- Evitate abbreviazioni o nomi brevi come `a`, `b`, `c`, senza che abbiano veramente senso.
- Rendete il nome il più descrittivo e preciso possibile. Esempi di pessimi nomi sono `data` e `value`. Questo tipo di nomi non dicono niente. Si possono utilizzare eccezionalmente se il contesto rende esplicito il significato.
- Definire delle regole personali o con il team. Se il visitatore del sito viene chiamato "user" allora dovremmo chiamare la relativa variabile come `currentUser` o `newUser`, non `currentVisitor` o `newManInTown`.

Sembra facile? Infatti lo è, ma trovare dei buoni nomi che siano precisi e descrittivi nella pratica non è sempre così semplice.

Nuovo o Riciclo?

Come ultima cosa. Ci sono alcuni programmati un po' pigri, che invece di dichiarare nuove variabili tendono a riutilizzare quelle già esistenti.

Il risultato che si ottiene, è che le variabili sono come delle scatole in cui si possono mettere varie cose, senza cambiare l'etichetta. Cosa ci sarà dentro in un dato momento? Chi lo sa... Siamo costretti a controllare manualmente.

Questo genere di programmati risparmiano qualche bit nella dichiarazione delle variabili ma perdono dieci volte il tempo risparmiato per fare debugging del codice.

Una variabile in più non è necessariamente un male.

I browser moderni e JavaScript minimizzano ed ottimizzano il codice abbastanza bene, quindi non ci saranno problemi di performance. Usare variabili differenti, per valori differenti può addirittura aiutare il motore JavaScript nell'ottimizzazione.

Riepilogo

Possiamo dichiarare variabili per memorizzare dati. Possono essere dichiarate con `var`, `let` o `const`.

- `let` – è una dichiarazione delle variabili più moderna.
- `var` – è una dichiarazione delle variabili più vecchia-scuola. Normalmente non si dovrebbe utilizzare, spiegheremo le sottili differenze da `let` nel capitolo [Il vecchio "var"](#), giusto per esserne a conoscenza.
- `const` – è simile a `let`, ma non consente di cambiare il valore della variabile.

Le variabili dovrebbero avere dei nomi che ci consentono di capire facilmente cosa c'è dentro.

✓ Esercizi

Lavorare con le variabili

importanza: 2

1. Dichiarare due variabili: `admin` e `name`.
2. Assegnare il valore "John" a `name`.
3. Copiare il valor da `name` su `admin`.
4. Mostra il valore di `admin` tramite `alert` (deve stampare "John").

[Alla soluzione](#)

Scegliere il giusto nome

importanza: 3

1. Create una variabile con il nome del nostro pianeta. Come chiameresti questo tipo di variabile?
2. Create una variabile che memorizza il nome del visitatore corrente. Come chiameresti questa variabile?

[Alla soluzione](#)

Costanti maiuscole?

importanza: 4

Analizziamo il seguente codice:

```
const birthday = '18.04.1982';
const age = someCode(birthday);
```

Abbiamo una costante `birthday` che indica una data e `age` che viene calcolata da `birthday` tramite un algoritmo (non viene fornito per brevità, e perchè non è importante per descrivere l'argomento).

Sarebbe giusto utilizzare lettere maiuscole per `birthday`? E per `age`? O anche per entrambe?

```
const BIRTHDAY = '18.04.1982'; // make uppercase?
const AGE = someCode(BIRTHDAY); // make uppercase?
```

[Alla soluzione](#)

Tipi di dato

Un valore in JavaScript ha sempre un tipo specifico. Ad esempio, string o number.

Ci sono otto tipi di base in JavaScript. In questo articolo, vedremo i loro aspetti generali, mentre nei prossimi capitoli parleremo di ognuno di essi in maniera più dettagliata.

Una variabile in JavaScript può contenere qualsiasi dato. Quindi è possibile avere una variabile di tipo string ed in un secondo momento potrebbe contenere un valore numerico:

```
// nessun errore
let message = "hello";
message = 123456;
```

I linguaggi di programmazione che lo consentono sono detti “dinamicamente tipati”, questo significa che ci sono tipi di dato, ma le variabili non sono legate ad un tipo.

Number

```
let n = 123;
n = 12.345;
```

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

Con i valori di tipo number si hanno diverse operazioni a disposizione, ad esempio la moltiplicazione `*`, la divisione `/`, l'addizione `+`, la sottrazione `-` e molte altre.

Oltre ai normali valori numeri, esistono anche i “valori numerici speciali” che appartengono sempre al tipo numerico: `Infinity`, `-Infinity` e `Nan`.

- `Infinity` rappresenta il concetto matematico Infinito ↪ ∞ . È un valore speciale che è più grande di qualsiasi altro numero.

Possiamo ottenerlo come risultato tramite la divisione per zero:

```
alert( 1 / 0 ); // Infinity
```

O inserendolo direttamente nel codice:

```
alert( Infinity ); // Infinity
```

- `Nan` rappresenta un errore di calcolo. È il risultato di un'operazione non corretta o indefinita, ad esempio:

```
alert( "not a number" / 2 ); // NaN, è una divisione errata
```

`Nan` è “fisso”. Qualsiasi operazione su `Nan` restituirà `Nan`:

```
alert( "not a number" / 2 + 5 ); // NaN
```

Quindi, se è presente un `Nan` da qualche parte nell'operazione matematica, questo si propagherà fino al risultato.

Le operazioni matematiche sono sicure

In JavaScript le operazioni matematiche sono sicure. Possiamo fare qualsiasi cosa: dividere per zero, trattare stringhe non numeriche come numeri, etc.

Lo script non si interromperà mai con un errore fatale. Nel peggiore dei casi otterremo un `Nan` come risultato.

I numeri con valore speciale appartengono formalmente al tipo “numerico”. Ovviamente non sono numeri nel vero senso della parola.

Vedremo di più su come lavorare con i numeri nel capitolo [Numeri](#).

BigInt

In JavaScript, il tipo “number” non può rappresentare valori interni più grandi di $(2^{53}-1)$ (che equivale a `9007199254740991`), o minori di $-(2^{53}-1)$. Questa è una limitazione tecnica dovuta alla loro rappresentazione interna.

Per la maggior parte degli scopi, questo intervallo è sufficiente, ma in alcuni casi potremmo aver bisogno di numeri molto grandi, ad esempio per la crittografia o timestamp con precisione al microsecondo.

Il tipo `BigInt` è stato aggiunto di recente al linguaggio, e consente di rappresentare numeri interi di lunghezza arbitraria.

Un valore di tipo `BigInt` viene creato aggiungendo `n` alla fine del numero:

```
// la "n" alla fine del numero indica che è un BigInt
const bigInt = 1234567890123456789012345678901234567890n;
```

Poiché i `BigInt` sono utilizzati raramente, non li analizzeremo in questo articolo, ma li vedremo più in dettaglio nel capitolo dedicato [BigInt](#).

Problemi di compatibilità

Attualmente, `BigInt` sono supportati da Firefox/Chrome/Edge/Safari, ma non da IE.

Potete sempre verificare [la tabella di compatibilità di MDN BigInt](#) per sapere quali versioni dei browser li supportano.

String

Una stringa in JavaScript deve essere tra apici.

```
let str = "Hello";
let str2 = 'Single quotes are ok too';
let phrase = `can embed another ${str}`;
```

In JavaScript, ci sono 3 tipi di apici.

1. Apici doppi: "Hello".
2. Apice singolo: 'Hello'.
3. Backtick: `Hello`.

Gli apici doppi e singoli sono apici “semplici”. In JavaScript li tratta allo stesso modo.

Il backtick è un tipo di apice utilizzato per definire stringhe con “funzionalità estese”. Ci consente di integrare variabili ed espressioni in una stringa racchiudendola tra \${...}, per esempio:

```
let name = "John";

// variabile integrata
alert(`Hello, ${name}!`); // Hello, John!

// espressione integrata
alert(`the result is ${1 + 2}`); // il risultato è 3
```

L'espressione all'interno di \${...} viene valutata ed il risultato diventa parte della stringa.

Possiamo metterci qualsiasi cosa: una variabile come name oppure un'espressione aritmetica come 1 + 2 o qualcosa di più complesso.

Nota che questo è possibile sono tramite l'uso del backtick. Gli altri apici non lo consentono!

```
alert("the result is ${1 + 2}"); // il risultato è ${1 + 2} (le virgolette non fanno nulla)
```

Copriremo meglio le stringhe nel capitolo [Stringhe](#).

i Non c'è il tipo carattere.

In alcuni linguaggi, c'è un tipo “carattere” per i caratteri singoli. Per esempio, nel linguaggio C ed in Java c'è char.

In JavaScript, non è presente questo tipo. C'è solamente il tipo: string, che può essere utilizzato per contenere un singolo carattere o più di uno.

Tipo boolean (tipo logico)

Il tipo boolean ha solo due valori: true e false.

Questo tipo viene tipicamente utilizzato per memorizzare valori “si/no”: true significa “Sì, corretto”, e false significa “No, scorretto”.

Per esempio:

```
let nameFieldChecked = true; // si, il campo nome è spuntato
```

```
let ageFieldChecked = false; // no, il campo age non è spuntato
```

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

```
let isGreater = 4 > 1;

alert( isGreater ); // true (il risultato del confronto è "si")
```

Copriremo i valori booleani più in dettaglio nel capitolo [Operatori logici](#).

Il valore “null”

Il valore speciale `null` non appartiene a nessun tipo di quelli descritti fino ad ora.

Fa parte di un altro tipo, che contiene solo il valore `null`:

```
let age = null;
```

In JavaScript `null` non è un “riferimento ad un oggetto inesistente” o un “puntatore nullo” come in altri linguaggi.

E’ solamente un valore speciale utilizzato per indicare il valore “nullo”, “vuoto” o “valore sconosciuto”.

Il codice sopra indica che `age` è sconosciuto o vuoto per qualche motivo.

Il valore “undefined”

Il valore speciale `undefined` è un tipo a se stante. Fa da tipo a se stesso, proprio come `null`.

Il significato di `undefined` è che “il valore non è assegnato”.

Se una variabile viene dichiarata, ma non assegnata, il suo valore è esattamente `undefined`:

```
let age;

alert(age); // mostra "undefined"
```

Tecnicamente, è possibile assegnare `undefined` a qualsiasi variabile:

```
let age = 100;

// change the value to undefined
age = undefined;

alert(age); // "undefined"
```

...Ma non è comunque consigliabile farlo. Normalmente, si utilizza `null` per descrivere un valore “vuoto” o “sconosciuto” della variabile, e `undefined` viene utilizzato solo per i controlli,

per verificare se la variabile è stata assegnata.

Object e Symbol

Il tipo `object` è un tipo speciale.

Tutti gli altri tipi descritti sono definiti “primitivi”, perché i loro valori possono contenere solo una cosa (può essere una stringa, un numero o altro). Invece, gli oggetti vengono utilizzati per memorizzare una collezione di dati ed entità più complesse. Li tratteremo nel capitolo [Oggetti](#) dopo avere appreso abbastanza sui tipi primitivi.

Il tipo `symbol` viene utilizzato per creare identificatori unici per gli oggetti. Li abbiamo citati per completezza, ma è meglio studiarli dopo aver compreso gli oggetti.

L'operatore `typeof`

L'operatore `typeof` ritorna il tipo dell'argomento. È utile quando vogliamo lavorare con valori di tipi differenti, o per eseguire controlli rapidi.

Sono supportate due sintassi:

1. Come operatore: `typeof x`.
2. Come funzione: `typeof(x)`.

In altre parole, funziona sia con le parentesi che senza. Il risultato è lo stesso.

Una chiamata a `typeof x` ritorna una stringa con il nome del tipo:

```
typeof undefined // "undefined"  
typeof 0 // "number"  
typeof 10n // "bigint"  
typeof true // "boolean"  
typeof "foo" // "string"  
typeof Symbol("id") // "symbol"  
typeof Math // "object" (1)  
typeof null // "object" (2)  
typeof alert // "function" (3)
```

Le ultime tre linee potrebbero richiedere una spiegazione ulteriore:

1. `Math` è un oggetto integrato che fornisce operazioni matematiche avanzate. Lo studieremo nel capitolo [Numeri](#). Qui ha il semplice scopo di rappresentare un oggetto.
2. Il risultato di `typeof null` è `"object"`. Questo è un errore del linguaggio, ufficialmente riconosciuto e mantenuto per retro-compatibilità. Ovviamente, `null` non è un oggetto. È un valore speciale che fa da tipo a se stesso. Quindi, nuovamente, questo è un errore del linguaggio.

3. Il risultato di `typeof alert` è "function", poiché `alert` è una funzione del linguaggio. Studieremo le funzioni nel prossimo capitolo, e vedremo che non c'è nessun tipo "funzione" nel linguaggio. Le funzioni appartengono al tipo oggetto. Ma `typeof` le tratta differentemente. Formalmente, è errato, ma molto utile nella pratica.

Riepilogo

Ci sono 7 tipi base in JavaScript.

- `number` per numeri di qualsiasi tipo: interi o in virgola mobile.
- `bigint` viene utilizzato per definire interi di lunghezza arbitraria.
- `string` per stringhe. Una stringa può contenere uno o più caratteri, non esiste nessun tipo `character`.
- `boolean` per `true/false`.
- `null` per valori sconosciuti – un valore a parte che contiene solo il valore `null`.
- `undefined` per valori non assegnati – un tipo a parte che ha il solo valore `undefined`.
- `object` per strutture dati più complesse.
- `symbol` per identificatori unici.

L'operatore `typeof` ci consente di vedere quale tipo è memorizzato nella variabile.

- Due forme: `typeof x` o `typeof(x)`.
- Ritorna una stringa con il nome del tipo, come `"string"`.
- Il valore `null` ritorna `"object"` – è un errore del linguaggio, infatti non è un oggetto.

Nel prossimo capitolo ci concentreremo nei tipi primitivi e quando avremo preso familiarità, passeremo agli oggetti.

✓ Esercizi

Gli apici nelle stringhe

importanza: 5

Qual'è l'output dello script?

```
let name = "Ilya";

alert(`hello ${1}`); // ?

alert(`hello ${"name"} `); // ?

alert(`hello ${name}`); // ?
```

[Alla soluzione](#)

Interazioni: alert, prompt, confirm

Questa parte del tutorial ha l'intenzione di coprire JavaScript così per "com'è", senza le caratteristiche specifiche di ogni ambiente.

Ma continueremo comunque ad utilizzare il browser come ambiente di test. Per farlo, abbiamo bisogno di conoscere un paio di funzioni utili per l'interazione con l'interfaccia utente. In questo capitolo prenderemo familiarità con le funzioni browser `alert`, `prompt` e `confirm`.

alert

Sintassi:

```
alert(message);
```

Questo mostra un messaggio e mette in pausa l'esecuzione dello script finché l'utente non preme il pulsante "OK".

Ad esempio:

```
alert("Hello");
```

La finestra che appare con il messaggio si chiama *modal window*. La parola "modal" significa che l'utente non potrà interagire con il resto della pagina, premere altri bottoni etc, fino a che non avrà interagito con la finestra. In questo esempio – quando premerà "OK".

prompt

La funzione `prompt` accetta due argomenti:

```
result = prompt(title, [default]);
```

Questo mostrerà una modal window con un messaggio testuale, un campo di input ed il bottone OK/CANCEL.

title

Il testo da mostrare all'utente.

default

Un secondo parametro opzionale, che rappresenta il valore iniziale del campo input.

Le parentesi quadre nella forma [...]

Le parentesi quadre intorno a `default` indicano che il parametro è opzionale, non richiesto.

L'utente potrà scrivere nel campo input del prompt e successivamente premere OK. O in alternativa potrà cancellare l'input premendo su CANCEL o la combinazione di tasti `Esc`.

La chiamata ad un `prompt` ritorna il testo del campo input o `null` se è stato premuto cancel.

Ad esempio:

```
let age = prompt('How old are you?', 100);
alert(`You are ${age} years old!`); // Tu hai 100 anni!
```

IE: inserisce sempre un valore di default

Il secondo parametro è opzionale. Ma se non inseriamo nulla, Internet Explorer inserirà il testo "undefined" nel prompt.

Provate ad eseguire il seguente codice su Internet Explorer:

```
let test = prompt("Test");
```

Quindi, per farlo funzionare ugualmente su IE, è consigliato fornire sempre il secondo argomento:

```
let test = prompt("Test", ''); // <-- per IE
```

confirm

La sintassi:

```
result = confirm(question);
```

La funzione `confirm` mostra una modal window con un `domanda` e due bottoni: OK e CANCEL.

Il risultato è `true` se viene premuto OK altrimenti è `false`.

Ad esempio:

```
let isBoss = confirm("Are you the boss?");
alert( isBoss ); // true se viene premuto OK
```

Riepilogo

Abbiamo studiato 3 funzioni specifiche dei browser per interagire con l'utente:

`alert`

mostra un messaggio.

`prompt`

mostra un messaggio che richiede all'utente di inserire un input. Ritorna il testo inserito, o in alternativa, se viene premuto Cancel o il tasto `Esc`, ritorna `null`.

`confirm`

mostra un messaggio e attende che l'utente prema “OK” o “Cancel”. Ritorna `true` nel caso in cui venga premuto “OK”, `false` altrimenti.

Tutti questi metodi sono dei modal window: quindi interrompono l'esecuzione dello script e non consentono all'utente di interagire con il resto della pagina finché la modal non viene chiusa.

Ci sono due limitazioni che sono condivise da tutti i metodi visti sopra:

1. La posizione esatta della modal window viene decisa dal browser. Solitamente sta al centro.
2. Anche la grafica della modal window dipende dal browser. Non possiamo modificarla.

Questo è il prezzo da pagare per la semplicità. Ci sono altri modi per mostrare finestre di tipo modal più eleganti, con più informazioni o con maggiori possibilità di interazione con l'utente, ma se non ci interessa fare grandi cose, questi metodi possono essere utili.

✓ Esercizi

Una semplice pagina

importanza: 4

Create una semplice pagina web che chiede un nome, e successivamente lo ritorna.

[Esegui la demo](#)

[Alla soluzione](#)

Conversione di tipi

Nella maggior parte dei casi, operatori e funzioni convertono automaticamente il valore nel tipo corretto.

Ad esempio, `alert` converte automaticamente un valore qualsiasi in una stringa, per poterla mostrare. Le operazioni matematiche convertono i valori in numeri.

Ci sono invece casi in cui è necessario convertire esplicitamente i valori per poter evitare errori.

Non parliamo ancora di oggetti

In questo capitolo non parleremo ancora di oggetti ma ci dedicheremo ai tipi primitivi. Approfondiremo gli oggetti e la loro conversione dopo averli studiati, nel capitolo [Conversione da oggetto a primitivi](#).

Conversione di stringhe

La conversione a `String` è utile quando abbiamo bisogno del formato stringa di un valore.

Ad esempio, `alert(value)` effettua questa conversione per mostrare il valore.

Possiamo anche utilizzare la funzione `String(value)`, per ottenere lo stesso risultato:

```
let value = true;
alert(typeof value); // boolean

value = String(value); // ora value è una stringa contenente "true"
alert(typeof value); // string
```

La conversione in stringa è quella più semplice. Il valore `false` diventa la stringa `"false"`, mentre `null` diventa `"null"` etc.

Conversione numerica

La conversione a `Number` viene applicata automaticamente nelle funzioni ed espressioni matematiche.

Ad esempio, quando la divisione `/` viene applicata ad un tipo non numerico:

```
alert( "6" / "2" ); // 3, le stringhe sono state converite a numeri
```

Possiamo utilizzare la funzione `Number(value)` per convertire esplicitamente un valore `value`:

```
let str = "123";
alert(typeof str); // string

let num = Number(str); // diventa il numero 123

alert(typeof num); // number
```

Una conversione esplicita è solitamente richiesta quando leggiamo un valore da una risorsa di tipo stringa, come un form testuale, ma ci aspettiamo l'inserimento di un numero.

Se la stringa non risulta essere un numero valido, il risultato della conversione sarà `Nan`, ad esempio:

```
let age = Number("an arbitrary string instead of a number");

alert(age); // NaN, conversione fallita
```

Le regole di conversione numerica:

Valore	Diventa...
<code>undefined</code>	<code>NaN</code>
<code>null</code>	<code>0</code>
<code>true e false</code>	<code>1 e 0</code>

Valore	Diventa...
string	Gli spazi bianchi all'inizio e alla fine vengono rimossi. Se la stringa rimanente è vuota, il risultato sarà <code>0</code> . Altrimenti, il numero viene "letto" dalla stringa. Un errore restituirà <code>Nan</code> .

Esempi:

```
alert( Number(" 123   ") ); // 123
alert( Number("123z") );    // NaN (errore nella lettura del numero "z")
alert( Number(true) );     // 1
alert( Number(false) );    // 0
```

Nota che `null` e `undefined` si comportano diversamente: `null` diventa zero, mentre `undefined` diventa `NaN`.

i L'addizione '+' concatena le stringhe

Quasi tutte le operazioni matematiche convertono valori in numeri. Con un importante eccezione per l'addizione `+`. Se uno degli operandi è una stringa, allora anche gli altri vengono convertiti in stringhe.

E successivamente li concatena (unisce):

```
alert( 1 + '2' ); // '12' (stringa a destra)
alert( '1' + 2 ); // '12' (stringa a sinistra)
```

Questo accade solo quando almeno uno degli argomenti è di tipo stringa. Altrimenti, i valori vengono convertiti in numeri

Boolean Conversion

La conversione a `Boolean` è quella più semplice.

Questa si verifica con le operazioni logiche (più avanti incontreremo i test di condizione ed altri tipi di operazione logiche), ma può anche essere richiamato manualmente con la funzione `Boolean(value)`.

Le regole di conversione:

- I valori che sono intuitivamente "vuoti", come lo `0`, una stringa vuota, `null`, `undefined` e `Nan` diventano `false`.
- Gli altri valori diventano `true`.

Ad esempio:

```
alert( Boolean(1) ); // true
alert( Boolean(0) ); // false

alert( Boolean("hello") ); // true
alert( Boolean("") ); // false
```

⚠️ Da notare: una stringa contenente "0" viene valutata come true

Alcuni linguaggi (come il PHP) trattano "0" come false. Diversamente in JavaScript una stringa vuota è sempre true.

```
alert( Boolean("0") ); // true  
alert( Boolean(" ") ); // spazi vuoti, valgono true (qualsiasi stringa non vuota viene interpretata come true)
```

Riepilogo

I tre tipi di conversioni più utilizzati sono: a *string*, a *number* e a *boolean*.

Conversione a stringa – Avviene quando stampiamo qualcosa a schermo, può essere richiamato con `String(value)`. La conversione a stringa è solitamente ovvia per i valori primitivi.

Conversione numerica – Utilizzata nelle operazioni matematiche, può essere richiamata esplicitamente con `Number(value)`.

La conversione segue le seguenti regole:

Value	Becomes...
undefined	NaN
null	0
true / false	1 / 0
string	La stringa viene letta per "com'è", gli spazi bianchi agli estremi vengono ignorati. Una stringa vuota diventa 0. Un errore restituisce NaN.

Conversione booleana – Avviene nelle operazioni logiche, può anche essere richiamato esplicitamente con `Boolean(value)`.

Segue le regole:

Value	Becomes...
0, null, undefined, NaN, ""	false
qualsiasi altro valore	true

La maggior parte di queste regole sono facili da capire e memorizzare. Gli errori più comuni che commettono le persone sono:

- `undefined` convertito a `Number` vale `NaN`, non `0`.
- `"0"` e le stringhe che contengono solamente spazi `" "` vengono interpretate come true.

Qui non abbiamo coperto gli oggetti, ci ritorneremo più avanti nel capitolo [Conversione da oggetto a primitivi](#) che è dedicato esclusivamente agli oggetti, dopo che avremmo imparato più cose basilari su JavaScript.

Operatori di base

Molti operatori matematici già li conosciamo dalle scuole. Tra di essi ci sono l'addizione `+`, la moltiplicazione `*`, la sottrazione `-` e coi via.

In questo capitolo inizieremo con gli operatori semplici, quindi ci concentreremo sugli aspetti specifici di Javascript che non vengono trattati a scuola.

Termini: “unario”, “binario”, “operando”

Prima di iniziare, cerchiamo di capire la terminologia.

- *Un operando* – è ciò a cui si applica l'operatore. Ad esempio nella moltiplicazione `5 * 2` ci sono due operandi: l'operando sinistro è il numero `5`, e l'operando di destra è il numero `2`. A volte gli operandi vengo anche chiamati “argomenti”.
- Un operatore è *unario* se ha un singolo operando. Ad esempio, la negazione `-` inverte il segno di un numero:

```
let x = 1;  
  
x = -x;  
alert( x ); // -1, viene applicata la negazione unaria
```

- Un operatore è *binario* se ha due operandi. Lo stesso operatore “meno” esiste nella forma binaria:

```
let x = 1, y = 3;  
alert( y - x ); // 2, la sottrazione binaria sottrae i valori
```

Formalmente, negli esempi precedenti abbiamo due diversi operatori che condividono lo stesso simbolo: la negazione (operatore unario che inverte il segno) e la sottrazione (operatore binario che esegue la sottrazione).

Operatori matematici

Sono supportati seguenti operatori matematici:

- Addizione `+`,
- Sottrazione `-`,
- Moltiplicazione `*`,
- Divisione `/`,
- Resto `%`,
- Potenza `**`.

I primi quattro sono piuttosto semplici, mentre `%` e `**` richiedono qualche spiegazione.

Resto %

L'operatore resto `%`, diversamente da quello che si può pensare, non è legato alla percentuale.

Il risultato di `a % b` è il **resto ↗** della divisione intera di `a` diviso `b`.

Ad esempio:

```
alert( 5 % 2 ); // 1, è il resto dell'operazione 5 diviso 2
alert( 8 % 3 ); // 2, è il resto dell'operazione 8 diviso 3
```

Elevamento a Potenza **

The exponentiation operator `a ** b` raises `a` to the power of `b`. L'operatore di elevamento a potenza `a ** b` eleva a potenza `a` usando `b` come esponente.

In nomenclatura matematica viene scritto come a^b .

Ad esempio:

```
alert( 2 ** 2 ); // 22 = 4
alert( 2 ** 3 ); // 23 = 8
alert( 2 ** 4 ); // 24 = 16
```

Come in matematica, l'esponente può essere anche un valore numerico non intero.

Ad esempio, la radice quadrata può essere vista come un elevamento a potenza con esponente `1/2`:

```
alert( 4 ** (1/2) ); // 2 (potenza 1/2 equivale alla radice quadrata)
alert( 8 ** (1/3) ); // 2 (potenza 1/3 equivale alla radice cubica)
```

Concatenazione di stringhe, operatore binario +

Adesso diamo un'occhiata alle caratteristiche degli operatori in JavaScript, che vanno oltre l'aritmetica scolastica.

Soltanente l'operatore di somma `+` viene utilizzato per sommare due numeri.

Ma se l'operatore binario `+` viene applicato a delle stringhe, queste verranno unite (concatenate):

```
let s = "my" + "string";
alert(s); // mystring
```

Nota che se almeno uno degli operandi è una stringa, anche gli altri verranno convertiti in stringa.

Ad esempio:

```
alert( '1' + 2 ); // "12"
alert( 2 + '1' ); // "21"
```

Come puoi vedere, non è importante se la stringa è il primo il secondo operando. La regola è semplice: se uno degli operandi è una stringa, anche gli altri vengono convertiti a stringa.

Comunque, queste operazioni vengono eseguite da sinistra verso destra. Se ci sono due numeri prima di una stringa, prima vengono sommati e il risultato convertito in stringa:

Ora un esempio più complesso:

```
alert(2 + 2 + '1'); // "41" non "221"
```

Qui le operazioni vengono eseguite una di seguito all'altra, da sinistra verso destra. Il primo `+` somma i due numeri e restituisce `4`, quindi il successivo `+` concatena a quest'ultimo la stringa `1`, quindi sarebbe come fare `4 + '1' = 41`.

```
alert('1' + 2 + 2); // "122" non "14"
```

In questo esempio il primo operando è una stringa, quindi il compilatore tratterà anche i successivi operandi come stringhe. I `2` verranno concatenati alla stringa `1`: `'1' + 2 = 12` quindi `'12' + 2 = '122'`.

L'operatore binario `+` è l'unico che può lavorare con le stringhe in questo modo. Gli altri operatori aritmetici funzionano solo con i numeri. Infatti convertono sempre i loro operandi in numeri.

Questo è un esempio per la sottrazione e la divisione:

```
alert( 6 - '2'); // 4, converte la stringa '2' in numero
alert( '6' / '2'); // 3, converte entrambi gli operandi in numeri
```

Conversione numerica, operatore unario `+`

L'operatore `+` esiste in due forme. La forma binaria che abbiamo utilizzato sopra, e quella unaria.

L'operatore unario `+` viene applicato ad un singolo valore. Nel caso questo sia un numero, non succede nulla. Se invece non è un numero, questo viene convertito in un operando di tipo numerico.

Ad esempio:

```
// Nessun effetto sui numeri
let x = 1;
alert( +x ); // 1

let y = -2;
alert( +y ); // -2

// Converte i valori non numerici
alert( +true ); // 1
alert( +'0' ); // 0
```

Si ottiene lo stesso risultato di `Number(...)`, ma in un modo più veloce.

La necessità di convertire stringhe in numeri si presenta molto spesso. Ad esempio, se stiamo prelevando un valore da un campo HTML, questo solitamente sarà di tipo stringa. Come procedere in caso volessimo sommare questi valori?

La somma binaria li concatenerebbe come stringhe:

```
let apples = "2";
let oranges = "3";

alert( apples + oranges ); // "23", la somma binaria concatena le stringhe
```

Se vogliamo trattarli come numeri, dobbiamo prima convertirli e successivamente sommarli:

```
let apples = "2";
let oranges = "3";

// entrambi i valori vengono convertiti in numeri prima della somma binaria
alert( +apples + +oranges ); // 5

// la variante più lunga
// alert( Number(apples) + Number(oranges) ); // 5
```

Dal punto di vista di matematico l'abbondanza `+` può sembrare errato, ma dal punto di vista di un programmatore non c'è nulla di strano: i `+` unari vengono applicati per primi e si occupa di convertire le stringhe in numeri, successivamente il `+` binario esegue la somma.

Perché il `+` unario viene applicato prima di quello binario? Come adesso vedremo, questo accade per via della sua *precedenza più alta*.

Precedenza degli operatori

Se un'espressione ha più di un operatore, l'ordine d'esecuzione viene definito dalla loro *precedenza*, in altre parole, c'è una priorità implicita tra gli operatori.

Fin dalle scuole sappiamo che la moltiplicazione nell'espressione `1 + 2 * 2` viene eseguita prima dell'addizione. E' proprio questo che si intende con precedenza. La moltiplicazione sta dicendo di avere *una precedenza più alta* rispetto all'addizione.

Le parentesi, sovrascrivono qualsiasi precedenza, quindi se non siamo soddisfatti dell'ordine d'esecuzione, possiamo utilizzarle: `(1 + 2) * 2`.

Ci sono molti operatori in JavaScript. Ogni operatore ha un suo grado di precedenza. Quello con il grado più elevato viene eseguito per primo. Se il grado di precedenza è uguale, l'esecuzione andrà da sinistra a destra.

Un estratto della [tabella delle precedenze ↗](#) (non è necessario che ve la ricordiate, ma tenete a mente che gli operatori unari hanno una precedenza più elevata rispetto ai corrispondenti binari):

Precedence	Name	Sign
...
17	unary plus	+

Precedence	Name	Sign
17	unary negation	-
16	exponentiation	**
15	multiplication	*
15	division	/
13	addition	+
13	subtraction	-
...
3	assignment	=
...

Come possiamo vedere, la “somma unaria”(unary plus) ha una priorità di 17 , che è maggiore del 13 dell’addizione(+ binario). Questo è il motivo per cui l’espressione "+apples + +oranges" esegue prima il + unario, e successivamente l’addizione.

Assegnazione

Da notare che anche l’assegnazione = è un operatore. Viene infatti elencato nella tabella delle precedenze con una priorità molto bassa: 3 .

Questo è il motivo per cui quando assegniamo un valore ad una variabile, come x = 2 * 2 + 1 , i calcoli vengono eseguiti per primi, e successivamente viene valutato l’operatore = , che memorizza il risultato in x .

```
let x = 2 * 2 + 1;
alert( x ); // 5
```

L’assegnazione = restituisce un valore

Il fatto che il simbolo = sia un operatore e non un costrutto “magico” del linguaggio, ha delle interessanti implicazioni.

Tutti gli operatori in Javascript restituiscono un valore. Questo è ovvio per + e - , ma è altrettanto vero per = .

La chiamata x = value scrive value in x e quindi lo restituisce.

Di seguito una dimostrazione di come usare un assegnamento come parte di una espressione più complessa:

```
let a = 1;
let b = 2;

let c = 3 - (a = b + 1);

alert( a ); // 3
alert( c ); // 0
```

Nell'esempio qui sopra, il risultato dell'espressione `(a = b + 1)` viene assegnato ad `a` (esso è `3`). Quindi questo viene utilizzato per le successive valutazioni.

E' un codice divertente, non trovate? Dovremmo sapere come funziona perché è possibile trovarlo in alcune librerie Javascript.

Comunque, per favore evitate di scrivere codice simile. Questo genere di scorciatoie rende il codice poco chiaro e leggibile.

Concatenare assegnazioni

Un'altra caratteristica interessante è l'abilità di concatenare assegnazioni:

```
let a, b, c;  
  
a = b = c = 2 + 2;  
  
alert( a ); // 4  
alert( b ); // 4  
alert( c ); // 4
```

Le assegnazioni concatenate vengono valutate da destra a sinistra. Prima viene valutata l'espressione più a destra `2 + 2` e successivamente viene valutata l'assegnazione a sinistra: `c`, `b` e `a`. Alla fine, tutte le variabili condivideranno lo stesso valore.

Ancora una volta, per favorire la leggibilità è meglio dividere il codice su più linee:

```
c = 2 + 2;  
b = c;  
a = c;
```

Questo è più facile da leggere, specialmente quando si scorre velocemente il codice.

Modifica sul posto

Spesso abbiamo bisogno di applicare un operatore ad una variabile ed assegnare il risultato alla variabile stessa.

Per esempio:

```
let n = 2;  
n = n + 5;  
n = n * 2;
```

Questa sintassi può essere abbreviata usando `+=` and `*=`:

```
let n = 2;  
n += 5; // ora n = 7 (equivale a n = n + 5)  
n *= 2; // ora n = 14 (equivale a n = n * 2)  
  
alert( n ); // 14
```

Gli operatori “modifica-e-assegna” esistono per tutti gli operatori matematici e sui bit: `/=`, `-=`, etc.

Questi operatori hanno la stessa precedenza delle normali assegnazioni, quindi vengono eseguiti dopo la maggior parte degli altri calcoli.

```
let n = 2;  
  
n *= 3 + 5;  
  
alert( n ); // 16 (prima viene valutata la parte destra, equivale a n *= 8)  
  
## Incremento/Decremento  
  
<!-- Can't use -- in title, because the built-in parser turns it into a 'long dash' - -->
```

L'incremento o il decremento di un numero di uno è una delle operazioni numeriche più comuni.

Quindi, ci sono speciali operatori dedicati a questo:

- ****Incremento**** `++`` incrementa la variabile di `1`:

```
```js run no-beautify  
let counter = 2;
counter++; // sarebbe come fare counter = counter + 1, ma in maniera più breve
alert(counter); // 3
```
```

- ****Decremento**** `--`` decremente la variabile di `1`:

```
```js run no-beautify  
let counter = 2;
counter--; // equivale a counter = counter - 1, ma è più breve da scrivere
alert(counter); // 1
```
```

```
```warn
```

L'Incremento/decremento possono essere applicati solo a variabili. Se tentiamo di utilizzarli co

Gli operatori `++` e `--` possono essere inseriti sia prima che dopo la variabile.

- Quando l'operatore viene messo dopo la variabile, viene detto “forma post-fissa”: `counter++`.
- La “forma pre-fissa” si ottiene inserendo l'operatore prima della variabile: `++counter`.

Entrambi questi metodi portano allo stesso risultato: incrementano `counter` di `1`.

C'è qualche differenza? Si, ma possiamo notarla solo se andiamo ad utilizzare il valore di ritorno di `++/-`.

Facciamo chiarezza. Come sappiamo, tutti gli operatori restituiscono un valore. Incremento e decremento non fanno eccezione. La forma pre-fissa restituisce il nuovo valore, mentre la forma post-fissa restituisce il vecchio valore (prima dell'incremento/decremento).

Per vedere le differenze ecco un esempio:

```
let counter = 1;
```

```
let a = ++counter; // (*)
```

```
alert(a); // 2
```

Nella riga (\*) la chiamata pre-fissa di `++counter` incrementa `counter` e ritorna il nuovo valore che è 2. Quindi `alert` mostra 2.

Adesso proviamo ad utilizzare la forma post-fissa:

```
let counter = 1;
let a = counter++; // (*) abbiamo sostituito ++counter con counter++

alert(a); // 1
```

Nella riga (\*) la forma *post-fissa* `counter++` incrementa `counter`, ma ritorna il *vecchio* valore (prima dell'incremento). Quindi `alert` mostra 1.

Per ricapitolare:

- Se il risultato di un incremento/decremento non viene utilizzato, non ci sarà differenza qualsiasi forma venga utilizzata:

```
let counter = 0;
counter++;
++counter;
alert(counter); // 2, le righe sopra fanno la stessa cosa
```

- Se l'intenzione è di incrementare il valore e utilizzare il valore, allora si utilizza la forma pre-fissa:

```
let counter = 0;
alert(++counter); // 1
```

- Se si ha intenzione di incrementare, ma utilizzare il valore precedente, allora sarà necessario utilizzare la forma post-fissa:

```
let counter = 0;
alert(counter++); // 0
```

## Incremento/decremento contro gli operatori

Gli operatori `++/-` possono essere utilizzati nello stesso modo all'interno di un'espressione. La loro precedenza sarà più alta rispetto alla maggioranza degli altri operatori aritmetici.

Ad esempio:

```
let counter = 1;
alert(2 * ++counter); // 4
```

Confrontatelo con:

```
let counter = 1;
alert(2 * counter++); // 2, perché counter++ ritorna il "vecchio" valore
```

Sebbene sia tecnicamente permesso, questa sintassi rende il codice meno leggibile. Una linea che esegue più operazioni `++` non è mai un bene.

Mentre leggiamo il codice, una rapido scorrimento con lo sguardo in “verticale” può facilmente farci perdere una parte di codice, come ad esempio `counter++`, e potrebbe quindi non essere ovvio che la variabile incrementa.

E' consigliato utilizzare lo stile “una linea – un’azione”:

```
let counter = 1;
alert(2 * counter);
counter++;
```

## Operatori sui bit

Gli operatori sui bit trattano gli argomenti come numeri interi rappresentati in 32-bit e lavorano sulla loro rappresentazione binaria.

Questi operatori non sono specifici di JavaScript, ma supportati in molti linguaggi di programmazione.

La lista degli operatori:

- AND (`&`)
- OR (`|`)
- XOR (`^`)
- NOT (`~`)
- LEFT SHIFT (`<<`)
- RIGHT SHIFT (`>>`)
- ZERO-FILL RIGHT SHIFT (`>>>`)

Questi operatori vengono utilizzati molto raramente, quando abbiamo bisogno di lavorare con i numeri al più basso livello (bit per bit). Non avremo bisogno di questi operatori molto presto, poiché lo sviluppo web ne fa un uso limitato, ma in alcune aree speciali, come la crittografia, sono utili. In caso di necessità potete leggere l'articolo [operatori BitWise](#) su MDN.

## Virgola

L'operatore virgola `,` è uno degli operatori più rari ed inusuali. Qualche volta viene utilizzato per scrivere codice più breve, quindi è necessario capirlo bene per sapere cosa sta succedendo.

L'operatore virgola ci consente di valutare diverse espressioni, dividendole con `,`. Ogni espressione viene valutata, ma viene restituito solo il risultato dell'ultima.

Ad esempio:

```
let a = (1 + 2, 3 + 4);
alert(a); // 7 (il risultato di 3 + 4)
```

Qui la prima espressione `1 + 2` viene valutata, ed il suo risultato viene scartato, successivamente viene eseguito `3 + 4` e il suo risultato viene restituito.

### La virgola ha una precedenza molto bassa

L'operatore virgola ha una precedenza molto bassa, più bassa di `=`, quindi le parentesi sono importanti nell'esempio sopra.

Senza parentesi: `a = 1 + 2, 3 + 4` verrebbe valutato `+` prima, sommando i numeri in `a = 3, 7`, poi viene valutato l'operatore di assegnazione `=` che assegna `a = 3`, e successivamente il numero `7` dopo la virgola, che viene ignorato.

Perché dovremmo avere bisogno di un operatore che non ritorna nulla tranne l'ultima parte?

Qualche volta le persone lo utilizzano in costrutti più complessi che seguono più azioni in una sola riga.

Ad esempio:

```
// tre operazioni in un'unica riga
for (a = 1, b = 3, c = a * b; a < 10; a++) {
 ...
}
```

Questo “trick” viene utilizzato in molti framework JavaScript, per questo l'abbiamo menzionato. Ma solitamente non migliora la leggibilità del codice, quindi dovremmo pensarci bene prima di scrivere questo tipo di espressioni.

## Esercizi

### La forma post-fissa e pre-fissa

importanza: 5

Quale sarà il valore finale delle variabili `a`, `b`, `c` e `d` dopo l'esecuzione del codice sotto?

```
let a = 1, b = 1;

let c = ++a; // ?
let d = b++; // ?
```

[Alla soluzione](#)

---

## Risultato dell'assegnazione

importanza: 3

Quale sarà il risultato di `a` e `x` dopo l'esecuzione del codice sotto?

```
let a = 2;

let x = 1 + (a *= 2);
```

[Alla soluzione](#)

---

## Conversioni di tipo

importanza: 5

Qual'è il risultato di questa espressione?

```
"" + 1 + 0
"" - 1 + 0
true + false
6 / "3"
"2" * "3"
4 + 5 + "px"
"$" + 4 + 5
"4" - 2
"4px" - 2
" -9 " + 5
" -9 " - 5
null + 1
undefined + 1
" \t \n" - 2
```

Pensaci bene, scrivi le risposte su un foglio e poi compara le tue risposte con le soluzioni.

[Alla soluzione](#)

---

## Sistema l'addizione

importanza: 5

Questo codice chiede all'utente di inserire due numeri, quindi ne mostra la somma.

Funziona in maniera errata. L'output nell'esempio qui sotto è `12` (per i valori inseriti di default).

Perché? Prova a correggerlo, il risultato deve essere `3`.

```
let a = prompt("First number?", 1);
let b = prompt("Second number?", 2);

alert(a + b); // 12
```

[Alla soluzione](#)

## Confronti

Molti operatori di confronto già li conosciamo dalla matematica:

- Maggiore/minore: `a > b`, `a < b`.
- Maggiore/minore o uguale: `a >= b`, `a <= b`.
- Uguaglianza: `a == b` (da notare che il doppio simbolo `=` indica un test di uguaglianza, mentre il simbolo unico `a = b` rappresenta un'assegnazione).
- Disuguaglianza. In matematica la notazione è `≠`, mentre in JavaScript viene scritto come `a != b`.

In questo articolo impareremo più approfonditamente i vari tipi di confronto, come vengono gestiti in JavaScript, incluse alcune importanti peculiarità

## Il risultato è booleano

Tutti gli operatori di confronto restituiscono un valore booleano.

- `true` – significa “si”, “corretto” o “vero”.
- `false` – significa “no”, “sbagliato” o “falso”.

Ad esempio:

```
alert(2 > 1); // true (corretto)
alert(2 == 1); // false (sbagliato)
alert(2 != 1); // true (corretto)
```

Il risultato di un confronto può essere assegnato ad una variabile, proprio come qualsiasi altro valore:

```
let result = 5 > 4; // assegna il valore del confronto
alert(result); // true
```

## Confronto di stringhe

Per vedere quale stringa è maggiore di un'altra, viene utilizzato il cosiddetto ordine “dizionario” o “lessicografico”.

In altre parole, le stringhe vengono confrontate lettera per lettera.

Ad esempio:

```
alert('Z' > 'A'); // true
alert('Glow' > 'Glee'); // true
alert('Bee' > 'Be'); // true
```

L'algoritmo per confrontare due stringhe è semplice:

1. Confronta il primo carattere di entrambe le stringhe.
2. Se il primo carattere della prima stringa è maggiore(o minore) di quello della seconda stringa, allora la prima stringa è maggiore(o minore) della seconda. Abbiamo quindi finito.
3. Altrimenti, se il primo carattere di entrambe le stringhe è identico, viene comparato il secondo nella stessa maniera.
4. Viene ripetuto questo procedimento fino alla fine di una delle due stringhe.
5. Se entrambe le stringhe hanno la medesima lunghezza, allora sono uguali. Altrimenti la stringa più lunga è quella maggiore.

Nell'esempio sopra, il confronto `'Z' > 'A'` porta ad un risultato al primo passo dell'algoritmo.

Mentre il secondo confronto tra `"Glow"` e `"Glee"` richiede più passi poiché le stringhe vengono confrontate carattere per carattere:

1. `G` è uguale a `G`.
2. `l` è uguale a `l`.
3. `o` è maggiore di `e`. Qui si ferma. La prima stringa è maggiore.

#### **i Non un vero e proprio dizionario, ma un ordine Unicode**

L'algoritmo di confronto esaminato sopra è molto simile a quello utilizzato nei dizionari cartacei o nelle agende telefoniche, ma non è esattamente lo stesso.

Ad esempio, le lettere maiuscole e minuscole contano. La lettera maiuscola `"A"` non è uguale alla lettera minuscola `"a"`. Qual'è la più grande? La maggiore è quella minuscola. Come mai? Perché le lettere minuscole hanno un indice maggiore nella tabella di encoding utilizzata JavaScript (Unicode). Ritorneremo nei dettagli specifici e alle conseguenze nel capitolo [Stringhe](#).

## Confronti tra tipi diversi

Quando compariamo valori che appartengono a tipi differenti, questi vengono convertiti in numeri.

Ad esempio:

```
alert('2' > 1); // true, la stringa '2' diventa il numero 2
alert('01' == 1); // true, la stringa '01' diventa il numero 1
```

Per i valori di tipo booleano, `true` diventa `1` e `false` diventa `0`, quindi:

```
alert(true == 1); // true
alert(false == 0); // true
```

### Una conseguenza divertente

E' possibile, contemporaneamente, che:

- Due valori sono uguali.
- Uno dei due è `true` come booleano e l'altro è `false` come booleano.

Ad esempio:

```
let a = 0;
alert(Boolean(a)); // false

let b = "0";
alert(Boolean(b)); // true

alert(a == b); // true!
```

Dal punto di vista di JavaScript questo è abbastanza normale. Un controllo di uguaglianza converte sempre utilizzando la conversione numerica (quindi `"0"` diventa `0`), mentre la conversione esplicita a booleano `Boolean` utilizza altre regole.

## Uguaglianza stretta

Il normale controllo di uguaglianza `==` ha dei problemi. Non distingue tra `0` e `false`:

```
alert(0 == false); // true
```

La stessa cosa accade con una stringa vuota:

```
alert("" == false); // true
```

Questo perché operandi di tipo diverso vengono convertiti in numerici dall'operatore di uguaglianza `==`. Una stringa vuota, così come un booleano `false`, diventa zero.

Come possiamo fare se vogliamo distinguere tra `0` e `false`?

**Un operatore di uguaglianza stretta `===` controlla l'uguaglianza senza conversione di tipo.**

In altre parole, se `a` e `b` sono di tipo differente, allora `a === b` restituirà subito `false` senza tentare di convertirli.

Proviamolo:

```
alert(0 === false); // false, perché il tipo è differente
```

Esiste anche un operatore di “disuguaglianza stretta” `!==`, analogo all’operatore `!=`.

L’operatore di uguaglianza stretta è un pò più lungo da scrivere, ma rende ovvio il controllo e lascia meno spazio ad errori.

## Confronto con null e undefined

C’è un comportamento poco intuitivo quando `null` o `undefined` vengono comparati con gli altri valori.

### Per un controllo di uguaglianza stretta `==`

Questi valori sono diversi, perché non appartengono allo stesso tipo.

```
```js run
alert( null == undefined ); // false
```

```

### Per un controllo non stretto `==`

Qui c’è una regola speciale. Questi due sono una “dolce coppia”: tra di loro sono uguali (riferito a `==`), ma non lo sono con nessun altro valore.

```
```js run
alert( null == undefined ); // true
```

```

### Per confronti matematici `<` `>` `<=` `>=`

I valori `null/undefined` vengono convertiti in numeri: `null` diventa `0`, mentre `undefined` diventa `NaN`.

Adesso notiamo una cosa divertente quando proviamo ad applicare queste regole. E, cosa più importante, come non cadere in tranelli a causa di queste caratteristiche.

### Un risultato strano: `null` vs `0`

Proviamo a confrontare `null` con lo zero:

```
alert(null > 0); // (1) false
alert(null == 0); // (2) false
alert(null >= 0); // (3) true
```

Matematicamente questo è strano. L’ultimo risultato dice che “`null` è maggiore o uguale a zero”. Quindi uno dei confronti sopra dovrebbe essere corretto, eppure risultano entrambi falsi.

La ragione è che il controllo di uguaglianza `==` e di confronto `>` `<` `>=` `<=` lavorano diversamente. Il confronto converte `null` in valore numerico, quindi lo tratta come `0`. Questo è il motivo per cui (3) `null >= 0` è vero e (1) `null > 0` è falso.

D’altra parte, il controllo di uguaglianza `==` per `undefined` e `null` viene eseguito, come detto, senza alcuna conversione. Essi sono uguali tra di loro, ma diversi a qualsiasi altro valore. Questo è il motivo per cui (2) `null == 0` è falso.

## Undefined è incomparabile

Il valore `undefined` non può essere confrontato con nessun altro valore:

```
alert(undefined > 0); // false (1)
alert(undefined < 0); // false (2)
alert(undefined == 0); // false (3)
```

Perché non va bene neanche lo zero? E' sempre falso!

Otteniamo questi risultati perché:

- I confronti (1) e (2) restituiscono `false` perché `undefined` viene convertito a `NaN`, e `NaN` è un valore numerico speciale che restituisce `false` in tutte le operazioni di confronto.
- Il confronto di uguaglianza (3) restituisce `false`, perché `undefined` è uguale solo a `null`, `undefined`, ma a nessun altro valore.

## Evitare i problemi

Perché abbiamo studiato questi esempi? Dovremmo ricordarci queste peculiarità tutte le volte? Beh, in realtà no. Questo tipo di trucchetti diventeranno familiari col tempo, ma c'è un modo sicuro per evitare i problemi che possono sorgere:

- Trattate tutti i confronti con `undefined/null`, ad eccezione del confronto stretto `==`, con particolare attenzione.
- Non utilizzate i confronti `>=` `>` `<` `<=` con variabili che potrebbero essere `null/undefined`, se non siete più che sicuri di ciò che state facendo. Se una variabile può avere questo tipo di valore, è buona norma eseguire un controllo separatamente.

## Riepilogo

- Gli operatori di confronto restituiscono sempre un valore booleano.
- Le stringhe vengono confrontate lettera per lettera seguendo l'ordine "lessicografico".
- Quando valori di tipo differente vengono confrontati, questi vengono convertiti in numeri (ad eccezione del controllo di uguaglianza stretto).
- I valori `null` e `undefined` sono `==` solo tra di loro, e a nessun altro valore.
- Va prestata attenzione quando si utilizzano gli operatori di confronto come `>` o `<` con variabili che potrebbero contenere `null/undefined`. Controllare separatamente l'assegnazione di `null/undefined` è una buona idea.

## ✓ Esercizi

### Confronti

importanza: 5

Quale sarà il risultato di queste espressioni?

```
5 > 4
"apple" > "pineapple"
"2" > "12"
undefined == null
undefined === null
```

```
null == "\n0\n"
null === +"\\n0\\n"
```

[Alla soluzione](#)

## Operatori condizionali: if, '?'

Qualche volta abbiamo bisogno di eseguire certe azioni solo nel caso valgano determinate condizioni.

Per questo c'è l'istruzione `if` e l'operatore condizionale di valutazione a cui ci riferiremo, per semplicità, con "l'operatore punto di domanda" `?`.

### L'istruzione "if"

L'istruzione `if( . . . )` valuta una condizione (racchiusa nelle parentesi); se il risultato è `true`, esegue il codice che segue `if`.

Ad esempio:

```
let year = prompt('In which year was ECMAScript-2015 specification published?', '');
if (year == 2015) alert('You are right!');
```

Nell'esempio sopra, la condizione è un semplice controllo di uguaglianza: `year == 2015`, ma potrebbe essere qualcosa di molto più complesso.

Se dobbiamo eseguire più di un'istruzione, queste vanno raggruppate tramite parentesi graffe:

```
if (year == 2015) {
 alert("That's correct!");
 alert("You're so smart!");
}
```

E' consigliabile raggruppare sempre il codice all'interno delle parentesi graffe `{}`, quando si usa un `if`, anche se contiene una sola istruzione. La leggibilità ne guadagna.

## Conversione booleana

L'istruzione `if ( ... )` valuta la condizione tra le parentesi e converte il risultato al tipo booleano.

Ricordiamo le regole di conversione viste nel capitolo [Conversione di tipi](#):

- Il numero `0`, una stringa vuota `""`, `null`, `undefined` e `NaN` diventano `false`. Per questo vengono chiamati valori "falsi".
- Gli altri valori diventano `true`, quindi vengono chiamati "veri".

Quindi, il codice nell'esempio sotto, non verrà mai eseguito:

```
if (0) { // 0 è falso
 ...
}
```

...Nel seguente esempio, invece, verrà sempre eseguito:

```
if (1) { // 1 è vero
 ...
}
```

Possiamo anche passare un valore già valutato in precedenza, come qui:

```
let cond = (year == 2015); // l'uguaglianza diventa vera o falsa

if (cond) {
 ...
}
```

## La clausola “else”

L'istruzione `if` può essere seguita da un blocco opzionale “else”. Questo viene eseguito quando la condizione è falsa.

Ad esempio:

```
let year = prompt('In which year was the ECMAScript-2015 specification published?', '');

if (year == 2015) {
 alert('You guessed it right!');
} else {
 alert('How can you be so wrong?'); // qualsiasi valore tranne 2015
}
```

## Condizione multiple: “else if”

Qualche volta vorremmo testare diverse varianti di una condizione. Per questo esiste la clausola `else if`.

Ad esempio:

```
let year = prompt('In which year was the ECMAScript-2015 specification published?', '');

if (year < 2015) {
 alert('Too early...');
} else if (year > 2015) {
 alert('Too late');
} else {
 alert('Exactly!');
}
```

Nel codice sopra JavaScript controlla anzitutto la prima condizione, `year < 2015`. Se risulta falsa va alla successiva condizione `year > 2015` ed esegue il codice dentro le parentesi graffe, altrimenti esegue il codice dentro al blocco `else`.

Ci possono essere molti blocchi `else if`. L'`else` finale è opzionale.

## Operatore condizionale ‘?’

Qualche volta abbiamo bisogno di assegnare un valore ad una variabile in base ad una certa condizione.

Ad esempio:

```
let accessAllowed;
let age = prompt('How old are you?', '');

if (age > 18) {
 accessAllowed = true;
} else {
 accessAllowed = false;
}

alert(accessAllowed);
```

Esiste un operatore “condizionale”, o “punto interrogativo”, che ci consente di farlo in maniera più breve e semplice.

L'operatore viene rappresentato dal punto interrogativo `?`. Il termine formale è “ternario”, perché richiede tre operatori. E' l'unico operatore in JavaScript che ne accetta così tanti.

La sintassi è:

```
let result = condition ? value1 : value2;
```

La `condition` viene valutata; se risulta vera, viene ritornato `value1`, altrimenti viene ritornato `value2`.

Ad esempio:

```
let accessAllowed = (age > 18) ? true : false;
```

Tecnicamente, potremmo omettere le parentesi attorno ad `age > 18`. L'operatore condizionale ha una precedenza molto bassa, viene eseguito dopo gli operatori di confronto `>`.

Il risultato dell'esempio sotto è uguale a quello precedente:

```
// l'operatore di confronto "age > 18" viene eseguito per primo
// (non c'è bisogno di racchiuderlo tra parentesi)
let accessAllowed = age > 18 ? true : false;
```

Ma le parentesi rendono il codice più leggibile, quindi è consigliabile utilizzarle.

### **i** Da notare:

Nell'esempio sopra sarebbe possibile omettere anche l'operatore ternario, perché l'operatore di confronto `>` ritorna già di suo `true/false`:

```
// stesso risultato (risulterà in `true` o `false`, a seconda del valore di `age`)
let accessAllowed = age > 18;
```

## Multipli operatori ‘?’

Una sequenza di operatori `?` consente di ritornare un valore che dipende da più condizioni.

Ad esempio:

```
let age = prompt('age?', 18);

let message = (age < 3) ? 'Hi, baby!' :
 (age < 18) ? 'Hello!' :
 (age < 100) ? 'Greetings!' :
 'What an unusual age!';

alert(message);
```

Potrebbe essere difficile, inizialmente, capirne la logica. Ma dopo averlo guardato da più vicino ci accorgeremo che è una semplice sequenza di condizioni.

1. Il primo operatore “?” controlla `age < 3`.
2. Se è vero – ritorna `'Hi, baby!'`, altrimenti – segue la colonna `" : "`, controlla `age < 18`.
3. Se questo è vero – ritorna `'Hello!'`, altrimenti – segue la colonna `" : "`, controlla `age < 100`.
4. Se questo è vero – ritorna `'Greetings!'`, altrimenti – segue la colonna `" : "`, ritorna `'What an unusual age!'`.

La stessa logica riscritta utilizzando `if..else`:

```
if (age < 3) {
 message = 'Hi, baby!';
} else if (age < 18) {
 message = 'Hello!';
} else if (age < 100) {
 message = 'Greetings!';
} else {
 message = 'What an unusual age!';
}
```

## Uso non tradizionale dell'operatore ‘?’

Qualche volta l'operatore `?` si utilizza per rimpiazzare l'istruzione `if`:

```
let company = prompt('Which company created JavaScript?', '');
(company == 'Netscape') ?
 alert('Right!') : alert('Wrong.');
```

In base alla valutazione della condizione `company == 'Netscape'`, viene eseguita la prima o la seconda parte (e il rispettivo `alert`).

Qui non assegniamo il risultato ad una variabile. L'idea è di eseguire un codice differente a seconda della condizione.

**Non è consigliabile utilizzare l'operatore ternario in questo modo.**

La notazione risulta essere molto più breve rispetto all'istruzione `if`; questo viene sfruttato da molti programmati, ma risulta meno leggibile.

Compariamo il codice sopra con una versione che utilizza `if` invece dell'operatore ternario `?:`:

```
let company = prompt('Which company created JavaScript?', '');
if (company == 'Netscape') {
 alert('Right!');
} else {
 alert('Wrong.');
}
```

I nostri occhi esaminano il codice verticalmente. I costrutti che si estendono per qualche riga risultano più semplici da capire piuttosto di un'unica istruzione che si estende orizzontalmente.

L'idea dell'operatore ternario `?:` è di ritornare, in base a una condizione, un valore piuttosto di un altro. Va quindi utilizzato solo in questo tipo di situazioni. Per eseguire diversi codici è consigliabile utilizzare il costrutto `if`.

## ✓ Esercizi

### if (una stringa con zero)

importanza: 5

L'`alert` verrà mostrato?

```
if ("0") {
 alert('Hello');
}
```

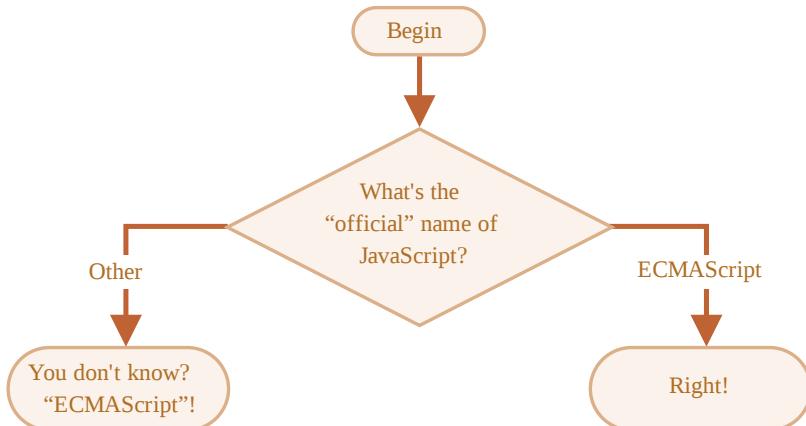
[Alla soluzione](#)

### Il nome di JavaScript

importanza: 2

Usando il costrutto `if..else`, scrivi un codice che chieda: ‘What is the “official” name of JavaScript?’

Se l’utente digita “ECMAScript”, ritorna “Right!”, altrimenti-- ritorna: “Didn’t know? ECMAScript!”



[Demo in una nuova finestra ↗](#)

[Alla soluzione](#)

## Mostra il segno

importanza: 2

Utilizzando `if..else`, scrivi un codice che preleva un numero tramite `prompt` e lo mostra con un `alert`:

- `1`, se il valore è maggiore di zero,
- `-1`, se è minore di zero,
- `0`, se è uguale a zero.

In questo esercizio supporremo che l’input sia sempre un numero.

[Demo in una nuova finestra ↗](#)

[Alla soluzione](#)

## Riscrivi 'if' con '?:'

importanza: 5

Riscrivi il seguente `if` utilizzando l’operatore ternario `'?'`:

```
let result;

if (a + b < 4) {
 result = 'Below';
} else {
 result = 'Over';
}
```

[Alla soluzione](#)

## Riscrivi 'if..else' con '?:'

importanza: 5

Riscrivi `if..else` utilizzando più volte l'operatore ternario `'?'`.

Per migliorare la leggibilità, è consigliato dividere il codice in più linee.

```
let message;

if (login == 'Employee') {
 message = 'Hello';
} else if (login == 'Director') {
 message = 'Greetings';
} else if (login == '') {
 message = 'No login';
} else {
 message = '';
}
```

[Alla soluzione](#)

## Operatori logici

In JavaScript ci sono quattro operatori logici: `||` (OR), `&&` (AND), e `!` (NOT), `??` (Nullish Coalescing). Qui abbiamo trattato i primi tre, l'operatore `??` sarà approfondito nel prossimo articolo.

Nonostante si chiamino “logici”, possono essere applicati a valori di qualsiasi tipo, non solo ai booleani (i risultati stessi possono essere di qualunque tipo).

Vediamoli nei dettagli.

### `|| (OR)`

L'operatore “OR” viene rappresentato da due linee verticali:

```
result = a || b;
```

Nella programmazione classica, l'OR logico è utilizzato per manipolare solo tipi booleani. Se almeno un argomento è `true`, allora il risultato sarà `true`, altrimenti sarà `false`.

In JavaScript questo operatore è un po' più potente. Ma prima vediamo come si comporta con i valori booleani.

Ci sono quattro combinazioni logiche possibili:

```
alert(true || true); // true
alert(false || true); // true
```

```
alert(true || false); // true
alert(false || false); // false
```

Come possiamo vedere, il risultato è sempre `true`, tranne nei casi in cui entrambi gli operandi sono `false`.

Se un operando non è di tipo booleano, allora viene momentaneamente convertito per la valutazione.

Ad esempio, il numero `1` viene considerato come `true`, il numero `0` come `false`:

```
if (1 || 0) { // funziona proprio come (true || false)
 alert('truthy!');
}
```

La maggior parte delle volte, OR `||` viene utilizzato in un `if` per verificare se *almeno una* delle condizioni è vera.

Ad esempio:

```
let hour = 9;

if (hour < 10 || hour > 18) {
 alert('The office is closed.');
}
```

Possiamo passare molteplici condizioni:

```
let hour = 12;
let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {
 alert('The office is closed.'); // l'Ufficio è chiuso
}
```

## OR "||" trova il primo valore vero

La logica descritta sopra è ovvia. Adesso proviamo ad addentrarci in qualche caratteristica "extra" di JavaScript.

Si può estendere l'algoritmo come segue.

Dati svariati operandi:

```
result = value1 || value2 || value3;
```

L'operatore OR `||` si comporta come segue:

- Valuta gli operandi da sinistra a destra.

- Ogni operando viene convertito a booleano. Se il risultato è `true`, il logical OR si ferma e ritorna il valore originale dell'operando.
- Se tutti gli operandi sono stati valutati e nessuno è `true`, ritorna l'ultimo operando.

Un valore viene ritornato nella sua forma originale, non nella sua conversione booleana.

In altre parole, una catena di OR `" || "` ritorna il primo valore vero; se invece non ce ne sono ritorna l'ultimo valore.

Ad esempio:

```
alert(1 || 0); // 1 (1 è vero)

alert(null || 1); // 1 (1 è il primo valore true)
alert(null || 0 || 1); // 1 (il primo valore true)

alert(undefined || null || 0); // 0 (tutti falsi, ritorna l'ultimo valore)
```

Questo ci permette alcuni utilizzi interessanti rispetto al “puro e classico OR booleano” boolean-only OR”.

## 1. Trovare il primo valore vero in una lista di variabili o espressioni.

Immaginiamo di avere diverse variabili, `firstName`, `lastName` e `nickName`, tutte opzionali (possono quindi essere `undefined` o avere valori falsi).

Possiamo utilizzare OR `||` per selezionare quella che contiene un valore e mostrarlo (oppure mostrare `"Anonymous"` se nessuna variabile è definita):

```
let firstName = "";
let lastName = "";
let nickName = "SuperCoder";

alert(firstName || lastName || nickName || "Anonymous"); // SuperCoder
```

Se tutte le variabili sono false, verrà mostrato `"Anonymous"`.

## 2. Valutazione a Corto-Circuito.

Gli operandi, oltre che valori, possono essere anche espressioni arbitrarie. L'operatore OR esegue la valutazione da sinistra a destra e si ferma al primo risultato vero, il quale viene ritornato. Il processo è chiamato “valutazione a corto-circuito” perché cerca di concludersi il prima possibile, senza dover elaborare tutti gli operandi.

Il logical OR è particolarmente utile quando il secondo argomento causerebbe un *side-effect* come l'assegnazione di una variabile o la chiamata a una funzione. Nell'esempio che segue solo il secondo messaggio verrà mostrato.

```
true || alert("not printed");
false || alert("printed");
```

Nella prima linea l'operatore OR trova subito un valore vero e ferma immediatamente la valutazione, quindi `alert` non viene eseguito. Si può utilizzare questa funzionalità per eseguire

un comando nel caso in cui la prima parte della condizione sia falsa.

## && (AND)

L'operatore AND viene rappresentato con `&&`:

```
result = a && b;
```

Nella programmazione classica AND ritorna `true` se entrambi gli operandi sono veri, altrimenti ritorna `false`:

```
alert(true && true); // true
alert(false && true); // false
alert(true && false); // false
alert(false && false); // false
```

Un esempio con `if`:

```
let hour = 12;
let minute = 30;

if (hour == 12 && minute == 30) {
 alert('The time is 12:30');
}
```

Proprio come per OR, anche per AND è consentito qualsiasi valore come operando:

```
if (1 && 0) { // valutato come true && false
 alert("won't work, because the result is falsy");
}
```

## AND “&&” trova il primo valore falso

Dati svariati operandi:

```
result = value1 && value2 && value3;
```

L'operatore AND `&&` si comporta come segue:

- Valuta gli operandi da sinistra a destra.
- Ogni operando viene convertito a booleano. Se il risultato è `false`, si ferma e ritorna il valore originale dell'operando.
- Se tutti gli operandi precedenti sono stati valutati e nessuno è `false`, ritorna l'ultimo operando.

In altre parole, AND ritorna il primo valore falso, altrimenti ritorna l'ultimo valore.

Le regole sono molto simili a quelle dell'OR. La differenza è che AND ritorna il primo valore *falso* mentre OR ritorna il primo valore *vero*.

Esempi:

```
// se il primo operando è vero,
// AND ritorna il secondo operando:
alert(1 && 0); // 0
alert(1 && 5); // 5

// se il primo operando è falso
// AND lo ritorna. Il secondo operando viene ignorato
alert(null && 5); // null
alert(0 && "no matter what"); // 0
```

Possiamo anche passare diversi valori in una sola riga. Nota come il primo valore falso viene ritornato non appena raggiunto:

```
alert(1 && 2 && null && 3); // null
```

Quando tutti i valori sono veri, viene ritornato l'ultimo valore:

```
alert(1 && 2 && 3); // 3, l'ultimo
```

### **i Precedenza di AND `&&` è maggiore dell'OR `||`**

La precedenza dell'operatore AND `&&` è maggiore di quella dell'OR `||`.

Quindi il codice `a && b || c && d` è analogo all'espressione: `(a && b) || (c && d)`.

### Non rimpiazzate `if` con `||` o `&&`

Talvolta, le persone utilizzano l'operatore AND `&&` come una “scorciatoia” dell'espressione `if`. Proprio come l'OR, anche AND `&&` può qualche volta rimpiazzare `if`.

Ad esempio:

```
let x = 1;

(x > 0) && alert('Greater than zero!');
```

Le azioni nella parte destra di `&&` vengono eseguite solamente se la valutazione non si ferma prima. Cioè: solo se `(x > 0)` è vera.

Il codice sopra è sostanzialmente analogo a:

```
let x = 1;

if (x > 0) alert('Greater than zero!');
```

La variante con `&&` sembra essere più corta. Ma l'istruzione `if` è più ovvia e tende ad essere più leggibile.

Quindi è consigliato usare ogni costrutto solo per i suoi scopi. Usate un `if` se volete imporre una condizione. Utilizzate invece `&&` se volete un AND.

## ! (NOT)

L'operatore booleano NOT viene rappresentato dal punto esclamativo `!`.

La sintassi è piuttosto semplice:

```
result = !value;
```

L'operatore accetta un solo argomento e si comporta come segue:

1. Converte l'operando a booleano: `true/false`.
2. Ritorna il valore inverso.

Ad esempio:

```
alert(!true); // false
alert(!0); // true
```

Un doppio NOT `!!` viene talvolta utilizzato per convertire un valore al tipo booleano:

```
alert(!!"non-empty string"); // true
alert(!!null); // false
```

Quello che accade è che il primo NOT converte l'operando a booleano e ritorna il suo inverso, e il secondo NOT lo *inverte nuovamente*. Il risultato è un valore di tipo booleano.

C'è un modo molto più lungo per fare la stessa cosa, usare la funzione `Boolean`, integrata in JavaScript:

```
alert(Boolean("non-empty string")); // true
alert(Boolean(null)); // false
```

La precedenza del NOT `!` è la più alta fra tutti gli operatori logici; viene sempre eseguita per prima e precede sia `&&` che `||`.

## ✓ Esercizi

---

### Qual è il risultato dell'OR?

importanza: 5

Quale sarà il risultato del codice sotto?

```
alert(null || 2 || undefined);
```

[Alla soluzione](#)

---

### Qual è il risultato dell'alert con l'OR?

importanza: 3

Cosa mostrerà il codice sotto?

```
alert(alert(1) || 2 || alert(3));
```

[Alla soluzione](#)

---

### Qual è il risultato dell'AND?

importanza: 5

Cosa mostrerà l'esecuzione di questo codice?

```
alert(1 && null && 2);
```

[Alla soluzione](#)

---

### Qual è il risultato degli alert?

importanza: 3

Cosa mostrerà questo codice?

```
alert(alert(1) && alert(2));
```

[Alla soluzione](#)

---

## Il risultato di OR AND OR

importanza: 5

Cosa verrà mostrato?

```
alert(null || 2 && 3 || 4);
```

[Alla soluzione](#)

---

## Controlla l'intervallo

importanza: 3

Scrivi una condizione `if` per controllare che `age` sia compresa tra `14` e `90` estremi inclusi.

“Inclusi” significa che `age` può valere anche `14` o `90`.

[Alla soluzione](#)

---

## Controlla l'intervallo fuori

importanza: 3

Scrivi una condizione `if` che controlli la variabile `age`; questa NON deve essere compresa tra `14` e `90` (inclusi).

Scrivi due varianti: la prima utilizzando `NOT !`, la seconda – senza.

[Alla soluzione](#)

---

## Un indovinello con "if"

importanza: 5

Quali di questi `alert` verranno eseguiti?

Quale sarà il risultato delle espressioni all'interno dei vari `if(...)`?

```
if (-1 || 0) alert('first');
if (-1 && 0) alert('second');
if (null || -1 && 1) alert('third');
```

[Alla soluzione](#)

---

## Controlla il login

importanza: 3

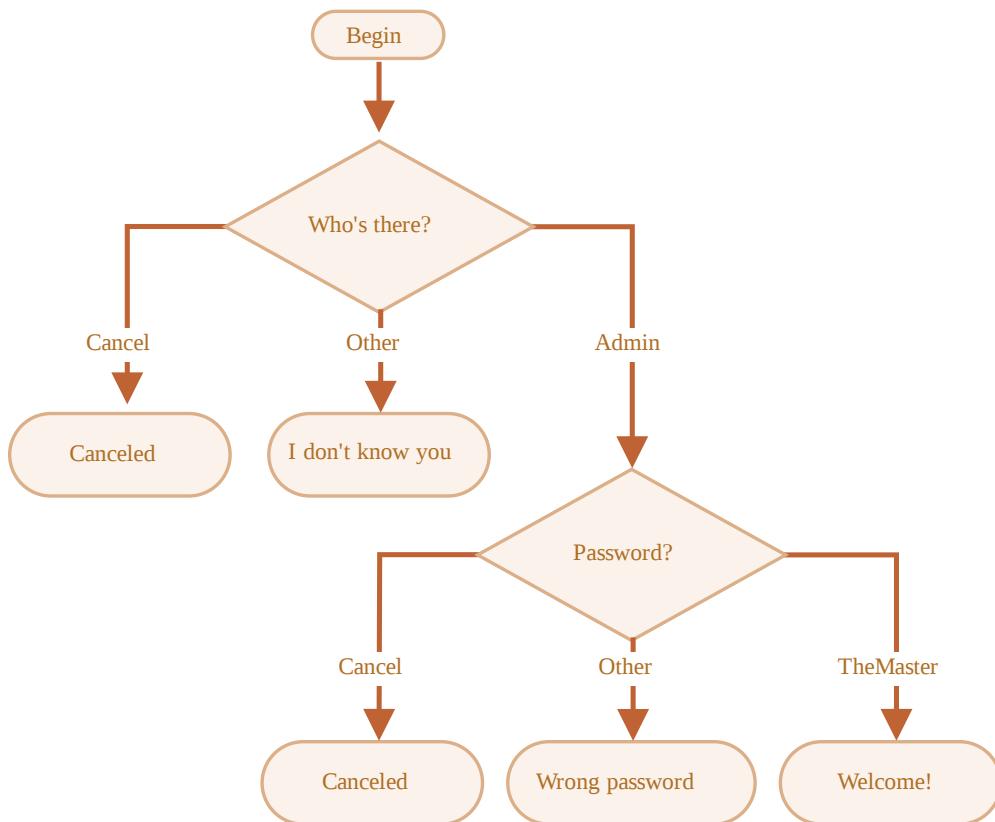
Scrivi il codice che richiede un login tramite `prompt`.

Se l'utente digita "Admin", si richiede una password tramite `prompt`; se l'input è una stringa vuota o `Esc` – mostra "Canceled."; se è diversa da "Admin", mostra "I don't know you".

La password viene controllata secondo i seguenti criteri:

- Se è uguale a "TheMaster", mostra "Welcome!",
- Un stringa diversa da "TheMaster" – mostra "Wrong password",
- Una stringa vuota o `Esc` – mostra "Canceled."

Lo schema:



Utilizza blocchi `if` annidati e tieni a mente la leggibilità del codice.

Suggerimento: passare un input vuoto tramite `prompt` ritorna una stringa vuota `''`. Premere `ESC` mentre il prompt è aperto ritorna `null`.

[Esegui la demo](#)

[Alla soluzione](#)

## Nullish coalescing operator '??'

### Aggiunta di recente

Questa funzionalità è stata aggiunta di recente al linguaggio. Vecchi browsers potrebbero richiedere polyfills.

Il *nullish coalescing operator* è rappresentato da due punti di domanda `??`.

Siccome trattiamo `null` e `undefined` in modo simile, avremo bisogno di una definizione particolare. In questo articolo, diremo che un'espressione è “definita” quando non è né `null` né `undefined`.

Il risultato di `a ?? b` è:

- se `a` è definito, allora `a`,
- se `a` non è definito, allora `b`.

In altre parole, tra due operatori `??` ritorna il primo se questo non è `null/undefined`; altrimenti, ritorna il secondo.

Il nullish coalescing operator non è qualcosa di completamente nuovo. È solo un modo più elegante per recuperare il primo valore “definito” tra due operatori.

Possiamo riscrivere `result = a ?? b` usando gli operatori che già conosciamo, nel seguente modo:

```
result = (a !== null && a !== undefined) ? a : b;
```

Un caso d'uso comune per l'operatore `??` è quello di fornire un valore di default per una variabile potenzialmente “non definita”.

Per esempio, qui mostriamo `Anonymous` se `user` non è definito:

```
let user;

alert(user ?? "Anonymous"); // Anonymous (user not defined)
```

Ovviamente, se `user` ha un qualsiasi valore eccetto `null/undefined`, allora vedremo quel valore:

```
let user = "John";

alert(user ?? "Anonymous"); // John (user defined)
```

Possiamo anche usare una sequenza di `??` per selezionare, da una lista, il primo valore che non sia `null/undefined`.

Per esempio, supponiamo di avere i dati di un utente nelle variabili `firstName`, `lastName` o `nickName`. Tutte queste potrebbero essere non definite, se l'utente dovesse decidere di non inserirne i valori.

Vorremmo visualizzare il nome dell'utente usando una di queste variabili, oppure mostrare "Anonymous" se nessuna di esse è definita.

Usiamo l'operatore `??`:

```
let firstName = null;
let lastName = null;
let nickName = "Supercoder";

// mostra il primo valore valido:
alert(firstName ?? lastName ?? nickName ?? "Anonymous"); // Supercoder
```

## Confronti con `||`

L'operatore OR `||` può essere usato nello stesso modo dell'operatore `??`, come descritto nel [capitolo precedente](#).

Per esempio, nel codice precedente potremmo rimpiazzare `??` con `||` e ottenere comunque il medesimo risultato:

```
let firstName = null;
let lastName = null;
let nickName = "Supercoder";

// mostra il primo valore vero:
alert(firstName || lastName || nickName || "Anonymous"); // Supercoder
```

L'operatore OR `||` esiste sin dagli inizi di JavaScript e gli sviluppatori lo hanno usato a tale scopo per molto tempo.

Il nullish coalescing operator `??`, invece, è stato aggiunto recentemente. La ragione è che alcuni sviluppatori non erano del tutto contenti dell'operatore `||`.

L'importante differenza tra essi è la seguente:

- `||` ritorna il primo valore *truthy*.
- `??` ritorna il primo valore *definito*.

In altre parole, `||` non distingue tra `false`, `0`, una stringa vuota `""` e `null/undefined`. In contesto booleano sono tutti valori `false`. Se uno di questi è il primo argomento di `||`, verrà ritornato il secondo argomento.

In pratica, però, potremmo voler usare il valore di default solamente quando la variabile è `null/undefined`. Ovvero quando è veramente non definita: una stringa vuota `''` o `0`, ad esempio, potrebbero tornarci utili.

Per esempio, consideriamo il seguente codice:

```
let height = 0;

alert(height || 100); // 100
alert(height ?? 100); // 0
```

- `height || 100` controlla se `height` ha un valore falso: così è.
  - il risultato è dunque il secondo argomento, `100`.
- `height ?? 100` controlla se `height` è `null/undefined`: non lo è.
  - quindi il risultato è `height`, ovvero `0`.

Se un'altezza pari a zero è un valore accettabile, questo non dovrebbe essere rimpiazzato con il valore di default (il secondo operatore, nel'esempio sopra `100`); in questo caso il *nullish coalescing operator* `??` è la scelta giusta.

## Precedenza

La precedenza dell'operatore `??` è piuttosto bassa: [5](#) nella [MDN table](#). Quindi `??` è valutato prima di `=` e `?`, ma dopo la maggior parte degli altri operatori, come `+` o `*`.

Quindi, se volessimo scegliere un valore tramite l'operatore `??` in un'espressione contenente altri operatori, dovremmo considerare l'utilizzo delle parentesi:

```
let height = null;
let width = null;

// importante: utilizzare le parentesi
let area = (height ?? 100) * (width ?? 50);

alert(area); // 5000
```

Altrimenti, se omettessimo le parentesi, siccome `*` ha una precedenza maggiore rispetto a `??`, sarebbe eseguito prima, portando a risultati scorretti.

```
// senza parentesi
let area = height ?? 100 * width ?? 50;

// ...funziona allo stesso modo del seguente codice (probabilmente non ciò che vogliamo)
let area = height ?? (100 * width) ?? 50;
```

## Usare `??` con `&&` o `||`

Per motivi di sicurezza, JavaScript proibisce l'utilizzo di `??` insieme agli operatori `&&` e `||`, a meno che la precedenza non sia esplicitamente specificata tramite l'utilizzo delle parentesi.

Il codice sotto causa un errore di sintassi:

```
let x = 1 && 2 ?? 3; // Syntax error
```

La limitazione è sicuramente discutibile, ma fu aggiunta alle specifiche del linguaggio; quando le persone hanno iniziato ad utilizzare `??` al posto di `||` - con lo scopo di evitare errori di programmazione.

Per aggirare il problema si possono utilizzare delle parentesi esplicite:

```
let x = (1 && 2) ?? 3; // Works
```

```
alert(x); // 2
```

## Riepilogo

- Il nullish coalescing operator `??` fornisce una scorciatoia per la scelta del primo valore "definito" da una lista di valori.

È usato per assegnare valori di default alle variabili:

```
// imposta height = 100 se 'height' è *null* o *undefined*
height = height ?? 100;
```

- L'operatore `??` ha una precedenza molto bassa, solo un po' più alta di `?` e `=`, quindi va considerata l'aggiunta di parentesi quando lo si utilizza all'interno di un'espressione.
- È proibito usarlo con `||` o `&&` senza l'utilizzo di parentesi esplicite.

## Cicli: while e for

Abbiamo spesso bisogno di eseguire la stessa azione più volte di fila.

Ad esempio, quando abbiamo bisogno di ritornare, una dopo l'altra, della merce da una lista; o anche solo eseguire lo stesso codice per ogni numero da 1 a 10.

I *cicli* sono un modo di ripetere una stessa parte di codice più volte.

### Il ciclo “while”

Il ciclo `while` ha la seguente sintassi:

```
while (condition) {
 // codice
 // "corpo del ciclo"
}
```

Fino a che la `condition` è `true`, il `code` nel corpo del ciclo viene eseguito.

Ad esempio, il ciclo qui sotto mostra `i` fino a che `i < 3`:

```
let i = 0;
while (i < 3) { // mostra 0, poi 1, poi 2
 alert(i);
 i++;
}
```

Un'esecuzione del codice nel corpo del ciclo viene chiamata *un'iterazione*. Il ciclo nell'esempio sopra fa tre iterazioni.

Se nell'esempio sopra non ci fosse `i++`, il ciclo si ripeterebbe per sempre; in teoria: in pratica, il browser ha dei metodi per bloccare questi cicli. In JavaScript server-side è necessario arrestare il processo.

Qualsiasi espressione o variabile può essere utilizzata come condizione di un ciclo, non solo un confronto come `i < 3`. Il ciclo `while` converte le espressioni al tipo booleano, che vengono poi valutate.

Ad esempio, un modo più breve di scrivere `while (i != 0)` potrebbe essere `while (i)`:

```
let i = 3;
while (i) { // quando i diventa 0, la condizione diventa falsa e il ciclo si conclude
 alert(i);
 i--;
}
```

### **💡 Le parentesi non sono richieste per un corpo composto da una singola linea di codice**

Se il corpo del ciclo ha una singola istruzione, possiamo omettere le parentesi `{...}`:

```
let i = 3;
while (i) alert(i--);
```

## Il ciclo “do...while”

La condizione da controllare può essere messa *dopo* il corpo del ciclo. Lo si fa utilizzando la sintassi `do .. while`:

```
do {
 // corpo del ciclo
} while (condition);
```

Il ciclo esegue prima il corpo, poi controlla la condizione; se questa è vera, esegue nuovamente il corpo.

Ad esempio:

```
let i = 0;
do {
 alert(i);
 i++;
} while (i < 3);
```

Questa tipo di sintassi viene usata molto raramente, ad eccezione dei casi in cui si vuole che il corpo del ciclo venga eseguito **almeno una volta**. Questo avviene ancor *prima* del controllo della condizione. La forma più comune, comunque, è `while(...){...}`.

## Il ciclo “for”

Il ciclo `for` è forse il più utilizzato.

La sua sintassi è:

```
for (begin; condition; step) {
 // ... corpo del ciclo ...
}
```

Cerchiamo ora di capire, tramite esempi, il suo funzionamento. Il ciclo sotto esegue `alert(i)` fino a quando la variabile `i` è più piccola di 3. A ogni iterazione, il valore di `i` aumenta di 1.

```
for (let i = 0; i < 3; i++) { // mostra 0, poi 1, poi 2
 alert(i);
}
```

Esaminiamo l'istruzione `for` parte per parte:

#### Parte

|           |                       |                                                                                                    |
|-----------|-----------------------|----------------------------------------------------------------------------------------------------|
| begin     | <code>i = 0</code>    | Viene eseguito una volta sola, all'entrata nel ciclo.                                              |
| condition | <code>i &lt; 3</code> | Viene controllata prima di ogni iterazione; se falsa, il ciclo si interrompe.                      |
| body      | <code>alert(i)</code> | Viene eseguito fino a quando la condizione è vera.                                                 |
| step      | <code>i++</code>      | Viene eseguito ad ogni iterazione, dopo il corpo, fintanto che la condizione è <code>true</code> . |

L'iterazione, generalmente, funziona nel modo seguente:

```
Eseguiamo begin
→ (if condition → run body and run step)
→ (if condition → run body and run step)
→ (if condition → run body and run step)
→ ...
```

Ricapitolando: `begin` viene eseguito per primo, *una volta sola*; subito dopo, inizia l'iterazione. Se `condition`, dopo la conversione a booleano, è `true`, vengono eseguiti `body` e `step`; se è `false`, il ciclo si interrompe.

Se i cicli vi sono nuovi, forse vi sarà d'aiuto tornare indietro agli esempi e provare a riprodurli, passo passo, su un foglio di carta.

Ecco esattamente, in dettaglio, ciò che avviene nel nostro codice:

```
// for (let i = 0; i < 3; i++) alert(i)

// inizia l'esecuzione, viene dichiarata la variabile i
let i = 0
// if(condition == true) → esegue il corpo e avanza
if (i < 3) { alert(i); i++ }
// if(condition == true) → esegue il corpo e avanza
if (i < 3) { alert(i); i++ }
// if(condition == true) → esegue il corpo e avanza
```

```
if (i < 3) { alert(i); i++ }
// ...si conclude, perché ora i == 3 e la condizione i < 3 == false
```

### ❶ Dichiarazioni di variabili inline

Qui il “counter”, che utilizzeremo nella nostra *condition*, è una variabile: `i`. Viene dichiarata all’interno del corpo del ciclo ed è accessibile solo al suo interno. Questo tipo di espressione si chiama “dichiarazione di una variabile *inline*”.

```
for (let i = 0; i < 3; i++) { // 'i' è definito, e accessibile, solo dentro il corpo del ciclo
 alert(i); // 0, 1, 2
}
alert(i); // errore, nessuna variabile 'i' fuori dal corpo del ciclo
```

Invece di definire una nuova variabile, possiamo utilizzarne una già esistente:

```
let i = 0;

for (i = 0; i < 3; i++) { // utilizza una variabile esistente
 alert(i); // 0, 1, 2
}

alert(i); // 3; la variabile `i` è accessibile (è stata dichiarata fuori dal corpo del ciclo)
```

## Parti opzionali

Ogni parte del ciclo `for` è opzionale.

Ad esempio, possiamo omettere `begin` se non abbiamo bisogno di una variabile per la nostra `condition`.

Come in questo esempio:

```
let i = 0; // la variabile 'i' è stata già dichiarata e assegnata

for (; i < 3; i++) { // saltiamo "begin"
 alert(i); // 0, 1, 2
}
```

Possiamo anche rimuovere lo `step`:

```
let i = 0;

for (; i < 3;) {
 alert(i++); // la variabile 'i' viene incrementata nel corpo del ciclo
}
```

Il ciclo, nell’esempio sopra, diventa uguale ad un `while (i < 3)`.

Possiamo rimuovere tutto, ma questo risulterebbe in un ciclo infinito:

```
for (;;) {
 // esegue il corpo del ciclo senza mai terminare
}
```

Nota che le due `;` del ciclo `for` devono essere presenti, altrimenti vi sarebbe un errore di sintassi.

## Interrompere un ciclo

Normalmente un ciclo termina quando la condizione diventa falsa.

Ma è possibile forzare l'uscita in qualsiasi momento. C'è una speciale direttiva per fare questo: `break`.

Ad esempio, il ciclo sotto richiede all'utente una serie di numeri, e termina quando nessun numero viene inserito:

```
let sum = 0;

while (true) {
 let value = +prompt("Enter a number", '');

 if (!value) break; // (*)

 sum += value;

}
alert('Sum: ' + sum);
```

La direttiva `break` viene attivata alla linea `(*)`, quando l'utente inserisce una linea vuota o annulla la procedura di input. Questo ferma il ciclo immediatamente, passando il controllo alla prima linea successiva al ciclo. In questo caso, `alert`.

La combinazione “ciclo infinito + `break` quando necessario” è utile in situazioni in cui la condizione deve essere verificata in un punto differente dall'inizio/fine del ciclo (questo può avvenire in un qualsiasi altro punto del corpo).

## Vai alla prossima iterazione

La direttiva `continue` è una versione leggera del `break`. Non blocca l'intero ciclo: interrompe solo l'iterazione corrente e forza il ciclo a passare all'iterazione successiva.

Possiamo utilizzarla se abbiamo finito con le operazioni che ci interessano e vogliamo passare all'iterazione seguente.

Il ciclo sotto usa `continue` per ritornare solo i valori dispari:

```
for (let i = 0; i < 10; i++) {
 // se condizione == true (`i` è pari) salta la restante parte di codice e passa alla prossima
 if (i % 2 == 0) continue;
 // se `i` è dispari, esegui codice
```

```
 alert(i); // 1, poi 3, 5, 7, 9 (nota le due condizioni sopra: abbiamo solo i con un valore dis
}
```

Per i valori pari di `i` la direttiva `continue` interrompe l'esecuzione del corpo e passa il controllo alla successiva iterazione del `for` (`i` viene incrementato, poi si controlla che la condizione sia true). Di conseguenza, l'`alert` viene seguito solo con i valori dispari.

### La direttiva `continue` aiuta a diminuire i livelli di nidificazione

Un ciclo che mostra solo valori dispari potrebbe essere:

```
for (let i = 0; i < 10; i++) {
 if (i % 2) { //la condizione == 0 quando un numero è pari; 0, ricordiamo, convertito a bool
 alert(i);
 }
}
```

Ovviamente possiamo raccogliere il codice in un blocco `if` piuttosto di usare `continue`. Dal punto di vista tecnico l'esempio sopra è identico a quello che lo precede, che invece utilizza `continue`. Nell'esempio sopra il codice dentro il corpo di `if` è una semplice chiamata ad `alert`; ma se il codice fosse più lungo di un paio di righe si rischierebbe di perdere in leggibilità.

### Vietato `break/continue` alla destra di '?

Va notato che questo costrutto sintattico non è un'espressione e non può quindi essere utilizzato con l'operatore ternario `?`. In particolare, direttive come `break/continue` non sono permesse.

Ad esempio, se prendiamo questo codice:

```
if (i > 5) {
 alert(i);
} else {
 continue;
}
```

...E lo riscriviamo utilizzando l'operatore ternario:

```
(i > 5) ? alert(i) : continue; // continue non è consentito qui
```

...Questo smetterà di funzionare. Un codice come quello sopra risulterà in un errore di sintassi:

Questa è solo un'altra ragione per cui non utilizzare l'operatore ternario `?` piuttosto di `if`.

## Etichette `break/continue`

Qualche volta abbiamo bisogno di uscire da una serie di cicli annidati in un *colpo solo*.

Ad esempio, nel codice sotto abbiamo due cicli `for` che usano `(i, j)` nelle proprie condizioni; fino a quando le variabili sono minori di 3, il ciclo viene eseguito; dentro al ciclo più interno un `prompt` mostra le due variabili `(i, j)` in una stringa di testo:

```
for (let i = 0; i < 3; i++) {

 for (let j = 0; j < 3; j++) {

 let input = prompt(`Value at coords ${i},${j}`);

 // come potremmo fare per uscire di qui e proseguire verso Done (sotto)?

 }
}

alert('Done!');
```

Abbiamo bisogno di un modo per bloccare il processo se l'utente annulla l'input.

Un semplice `break` dopo la variabile `input` interromperebbe solo il ciclo più interno. Questo non è sufficiente. I *label* ci vengono in soccorso.

Un *label* (“etichetta”) è un identificatore seguito da “`:`” e da un ciclo:

```
labelName: for (...) {
 ...
}
```

L'istruzione `break <labelName>` interrompe il ciclo e passa il controllo a *label*.

Come nell'esempio:

```
outer: for (let i = 0; i < 3; i++) {

 for (let j = 0; j < 3; j++) {

 let input = prompt(`Value at coords ${i},${j}`);

 // se si ha una stringa vuota, allora si esce da entrambi i cicli
 if (!input) break outer; // (*)

 // fa qualcosa con i valori...
 }
}
alert('Done!');
```

Nel codice sopra `break outer` interrompe il ciclo e va all'etichetta chiamata `outer`.

Quindi il controllo va da `(*)` a `alert('Done!')`.

Possiamo anche spostare l'etichetta in un'altra linea:

```
outer:
for (let i = 0; i < 3; i++) { ... }
```

Anche la direttiva `continue` può essere utilizzata con un'etichetta. In questo caso l'esecuzione salta alla prossima iterazione del ciclo con quell'etichetta.

### ⚠️ I `label` non equivalgono a “`goto`”

I `label` non permettono di saltare in un punto arbitrario del codice.

Ad esempio, non è possibile:

```
break label; // non salta all'etichetta sotto

label: for (...)
```

La direttiva `break` deve essere all'interno del blocco di codice. Tecnicamente l'etichettatura funzionerà con qualsiasi blocco di codice, ad esempio:

```
label: {
 // ...
 break label; // funziona
 // ...
}
```

...Comunque, nel 99.9% dei casi, `break` viene usato nei cicli, come abbiamo visto negli esempi precedenti.

La chiamata a `continue` è possibile solo dall'interno di un ciclo

## Riepilogo

Abbiamo visto tre tipi di cicli:

- `while` – La condizione viene controllata prima di ogni iterazione.
- `do..while` – La condizione viene controllata dopo una prima iterazione.
- `for (;;)` – La condizione viene controllata prima di ogni iterazione; sono possibili altre condizioni all'interno del ciclo.

Per creare un ciclo infinito, si usa il costrutto `while(true)`. Questo tipo di cicli, come tutti gli altri, possono essere interrotti con la direttiva `break`.

Se non si ha più intenzione di fare nulla nell'iterazione corrente e si vuole quindi saltare alla successiva, possiamo usare la direttiva `continue`.

`break/continue` supportano le etichette prima del ciclo. Un etichetta è l'unico modo per `break/continue` di uscire da cicli annidati ed arrivare ciclo esterno.

## ✓ Esercizi

## Valore all'ultimo ciclo

importanza: 3

Qual è l'ultimo valore mostrato da `alert` in questo codice? Perché?

```
let i = 3;

while (i) {
 alert(i--);
}
```

[Alla soluzione](#)

## Quali valori mostrerà il ciclo while?

importanza: 4

Per ogni iterazione del ciclo, scrivi quali valori verranno mostrati e poi confrontali con la soluzione.

Entrambi i cicli mostreranno gli stessi valori in `alert`, o no?

1.

La forma prefissa `++i`:

```
let i = 0;
while (++i < 5) alert(i);
```

2.

La forma postfissa `i++`

```
let i = 0;
while (i++ < 5) alert(i);
```

[Alla soluzione](#)

## Quali valori verranno mostrati dal ciclo "for"?

importanza: 4

Per ogni ciclo scrivete quali valori verranno mostrati. Poi confrontateli con la soluzione.

I due `alert` mostreranno gli stessi valori?

1.

Forma postfissa:

```
for (let i = 0; i < 5; i++) alert(i);
```

2.

Forma prefissa:

```
for (let i = 0; i < 5; ++i) alert(i);
```

[Alla soluzione](#)

---

## Mostra i numeri pari con un ciclo 'for'

importanza: 5

Utilizzate il ciclo `for` per mostrare i numeri pari da `2` a `10`.

[Esegui la demo](#)

[Alla soluzione](#)

---

## Sostituisci "for" con "while"

importanza: 5

Riscrivi il ciclo `for` utilizzando la sintassi `while`, ma senza alterarne la funzionalità (l'output deve rimanere lo stesso).

```
for (let i = 0; i < 3; i++) {
 alert(`number ${i}!`);
```

[Alla soluzione](#)

---

## Ripeti fino a quando l'input è corretto

importanza: 5

Scrivi un ciclo che richieda (tramite *prompt*) un numero maggiore di `100`. Se l'utente inserisce un numero non valido – chiedete di inserirlo nuovamente.

Il ciclo deve continuare a richiedere un numero fintanto che l'utente non inserisce un numero maggiore di `100`, oppure annulla l'input (sia premendo *cancel* che inserendo una riga vuota).

Possiamo assumere che l'utente inserisca solo numeri. Non c'è quindi bisogno di implementare alcun tipo di logica per un input di tipo non numerico.

[Esegui la demo](#)

[Alla soluzione](#)

---

## Mostra i numeri primi

importanza: 3

Un valore intero maggiore di 1 si definisce **primo** se non può essere diviso, senza resto , da nessun numero ad eccezione di 1 e se stesso.

In altre parole, n > 1 è primo se non può essere diviso da nessun altro numero ad eccezione di 1 e n .

Ad esempio, 5 è primo perchè non può essere diviso senza resto da 2, 3 o 4 .

**scrivi un codice che mostri i numeri primi nell'intervallo da 2 a n .**

Per n = 10 il risultato sarà 2, 3, 5, 7 .

P.S. Il codice dovrebbe funzionare per qualsiasi numero n , non solo per un valore numerico prefissato.

[Alla soluzione](#)

## L'istruzione "switch"

L'istruzione switch può essere utile per rimpiazzare multipli if .

E' infatti un metodo molto più descrittivo per lavorare con un elemento che può avere svariati valori.

## La sintassi

Un istruzione switch possiede uno o più case ed opzionalmente un blocco default.

Un esempio:

```
switch(x) {
 case 'value1': // if (x == 'value1')
 ...
 [break]

 case 'value2': // if (x == 'value2')
 ...
 [break]

 default:
 ...
 [break]
}
```

- Il valore di x viene controllato utilizzando l'uguaglianza stretta con i valori dei blocchi case ( value1 e value2 nell'esempio sopra).
- Se l'uguaglianza viene trovata, switch inizia ad eseguire il codice partendo dal corrispondente blocco case , fino al break più vicino (oppure fino alla fine dello switch ).

- Se non viene trovata nessuna uguaglianza allora viene eseguito il codice del blocco `default` (se presente).

## Un esempio

Un esempio di `switch`:

```
let a = 2 + 2;

switch (a) {
 case 3:
 alert('Too small');
 break;
 case 4:
 alert('Exactly!');
 break;
 case 5:
 alert('Too big');
 break;
 default:
 alert("I don't know such values");
}
```

Qui lo `switch` inizia confrontando `a` con il primo `case`, il cui valore è `3`. Non vi è corrispondenza.

Poi valuta `4`. C'è una corrispondenza, quindi l'esecuzione inizia da `case 4` fino al `break` più vicino.

**Se non c'è nessun `break` l'esecuzione procede al prossimo `case`.**

Un esempio senza `break`:

```
let a = 2 + 2;

switch (a) {
 case 3:
 alert('Too small');
 case 4:
 alert('Exactly!');
 case 5:
 alert('Too big');
 default:
 alert("I don't know such values");
}
```

Nell'esempio sopra, non essendoci un `break`, abbiamo l'esecuzione sequenziale dei tre `alert`:

```
alert('Exactly!');
alert('Too big');
alert("I don't know such values");
```

**i Qualsiasi espressione può essere un argomento `switch/case`.**

Sia `switch` che `case` accettano espressioni arbitrarie.

Ad esempio:

```
let a = "1";
let b = 0;

switch (+a) {
 case b + 1:
 alert("this runs, because +a is 1, exactly equals b+1");
 break;

 default:
 alert("this doesn't run");
}
```

Qui `+a` viene convertito in `1`, che nei `case` viene confrontato con `b + 1`, ed il codice

## Raggruppare i “case”

Possiamo raggruppare diverse varianti di `case` e far loro eseguire lo stesso codice.

Ad esempio, se vogliamo eseguire lo stesso codice per `case 3` e `case 5`:

```
let a = 3;

switch (a) {
 case 4:
 alert('Right!');
 break;

 case 3: // (*) raggruppiamo due casi
 case 5:
 alert('Wrong!');
 alert("Why don't you take a math class?");
 break;

 default:
 alert('The result is strange. Really.');
}
```

Ora sia `3` che `5` mostreranno lo stesso messaggio.

L’abilità di “raggruppare” più `case` è un effetto collaterale di come `switch/case` funziona senza `break`. Qui l’esecuzione del `case 3` inizia dalla linea `(*)` e prosegue fino a `case 5`, perché non c’è alcun `break`.

## Il tipo conta

Mettiamo in risalto che il confronto di uguaglianza è sempre stretto. I valori devono essere dello stesso tipo perché si possa avere una corrispondenza.

Ad esempio, consideriamo il codice:

```
let arg = prompt("Enter a value?");
switch (arg) {
 case '0':
 case '1':
 alert('One or zero');
 break;

 case '2':
 alert('Two');
 break;

 case 3:
 alert('Never executes!');
 break;
 default:
 alert('An unknown value');
}
```

1. Per `0` e `1`, viene eseguito il primo `alert`.
2. Per `2` viene eseguito il secondo `alert`.
3. Per `3`, il risultato del `prompt` è una stringa, `"3"`, che non è strettamente uguale `==` al numero `3`. Quindi abbiamo del codice ‘morto’ nel `case 3!` Verrà quindi eseguito il codice dentro `default`.

## ✓ Esercizi

### Riscrivi il costrutto "switch" come un "if"

importanza: 5

Riscrivi il codice utilizzando `if..else` in modo tale che corrisponda al seguente `switch`:

```
switch (browser) {
 case 'Edge':
 alert("You've got the Edge!");
 break;

 case 'Chrome':
 case 'Firefox':
 case 'Safari':
 case 'Opera':
 alert('Okay we support these browsers too');
 break;

 default:
 alert('We hope that this page looks ok!');
}
```

[Alla soluzione](#)

## Riscrivi "if" utilizzando "switch"

importanza: 4

Riscrivi il codice sotto utilizzando un singolo `switch`:

```
let a = +prompt('a?', '0');

if (a == 0) {
 alert(0);
}
if (a == 1) {
 alert(1);
}

if (a == 2 || a == 3) {
 alert('2,3');
}
```

[Alla soluzione](#)

## Funzioni

Molto spesso abbiamo la necessità di eseguire azioni simili in diverse parti dello script.

Ad esempio, vogliamo mostrare un messaggio quando un utente effettua il login/logout o per qualsiasi altro motivo.

Le funzioni sono le “fondamenta” di un programma. Consentono a un codice di essere utilizzato più volte, evitando ripetizioni.

Abbiamo già visto esempi di funzioni integrate nel linguaggio, come `alert(message)`, `prompt(message, default)` e `confirm(question)`. Ma possiamo benissimo anche creare delle nostre funzioni personali.

### Dichiarazione di Funzione

Per creare una funzione dobbiamo utilizzare una *dichiarazione di funzione*.

Che appare così:

```
function showMessage() {
 alert('Hello everyone!');
}
```

La parola chiave `function` va posta all'inizio; viene seguita dal *nome della funzione*, poi c'è una lista di *parametri*, racchiusi tra parentesi (separati da virgola, in questo esempio la lista è vuota, vedremo un esempio più avanti) e infine il codice della funzione, chiamato anche “corpo della funzione”, racchiuso tra parentesi graffe.

```
function name(parameter1, parameter2, ... parameterN) {
 ...body...
}
```

La nostra nuova funzione può essere chiamata tramite il suo nome: `showMessage()`.

Ad esempio:

```
function showMessage() {
 alert('Hello everyone!');
}

showMessage();
showMessage();
```

La chiamata `showMessage()` esegue il codice dentro la funzione. Qui vedremo il messaggio due volte.

Questo esempio mostra chiaramente uno dei principali scopi delle funzioni: evitare le ripetizioni di un codice.

Se avremo la necessità di cambiare il messaggio o il modo in cui viene mostrato, sarà sufficiente modificare il codice in un solo punto: la funzione che lo implementa.

## Variabili locali

Una variabile dichiarata all'interno di una funzione è visibile solamente all'interno di quella funzione.

Ad esempio:

```
function showMessage() {
 let message = "Hello, I'm JavaScript!"; // variabile locale

 alert(message);
}

showMessage(); // Hello, I'm JavaScript!

alert(message); // <-- Errore! La variabile è locale alla funzione
```

## Variabili esterne

Una funzione può accedere ad una variabile esterna, ad esempio:

```
let userName = 'John';

function showMessage() {
 let message = 'Hello, ' + userName;
 alert(message);
}
```

```
showMessage(); // Hello, John
```

La funzione ha pieno accesso alla variabile esterna. Può anche modificarla.

Un esempio:

```
let userName = 'John';

function showMessage() {
 userName = "Bob"; // (1) cambiata la variabile esterna

 let message = 'Hello, ' + userName;
 alert(message);
}

alert(userName); // John prima della chiamata di funzione

showMessage();

alert(userName); // Bob, il valore è stato modificato dalla funzione
```

La variabile esterna viene utilizzata solo se non ce n'è una locale.

Se una variabile con lo stesso nome viene dichiarata all'interno di una funzione, questa *oscurerà* quella esterna. Ad esempio, nel codice sotto la funzione usa la variabile locale `userName`. Quella esterna viene ignorata:

```
let userName = 'John';

function showMessage() {
 let userName = "Bob"; // dichiara una variabile locale

 let message = 'Hello, ' + userName; // Bob
 alert(message);
}

// la funzione creerà un suo 'personale' userName
showMessage();

alert(userName); // John, intoccato, la funzione non può accedere alla variabile esterna
```

### Variabili globali

Le variabili dichiarate all'esterno di qualsiasi funzione, come `userName` nel codice sopra, vengono chiamate *globali*.

Le variabili globali sono visibili a qualsiasi funzione (se non sono oscurate da quelle locali).

Soltanamente, una funzione dichiara internamente tutte le variabili necessarie per svolgere il suo compito. Le variabili locali vengono utilizzate per memorizzare dati relativi alla funzione stessa, quando è importante che queste non siano accessibili in qualsiasi punto del codice. I codici moderni cercano di evitare le variabili globali, sebbene qualche volta possano essere utili per dati

## Parametri

Possiamo passare dei dati arbitrari ad una funzione usando i parametri.

Nell'esempio sotto, la funzione ha due parametri: `from` e `text`.

```
function showMessage(from, text) { // parametri: from, text
 alert(from + ': ' + text);
}

showMessage('Ann', 'Hello!'); // Ann: Hello! (*)
showMessage('Ann', "What's up?"); // Ann: What's up? (**)
```

Quando la funzione viene chiamata nelle righe (\*) e (\*\*), il valore passato viene copiato nelle variabili locali `from` e `text`, che verranno utilizzate nella chiamata ad `alert`.

Guardiamo un altro esempio: abbiamo una variabile `from` e la passiamo a una funzione. Da notare: la funzione cambia `from`, ma il cambiamento non è visibile all'esterno perché la funzione usa sempre una copia del valore passato:

```
function showMessage(from, text) {

 from = '*' + from + '*'; // rende "from" più carino

 alert(from + ': ' + text);
}

let from = "Ann";

showMessage(from, "Hello"); // *Ann*: Hello

// il valore di "from" è lo stesso, la funzione ne ha modificato una copia locale
alert(from); // Ann
```

Quando un valore viene passato come parametro di funzione, viene anche chiamato *argomento*.

In altre parole In other words, per chiarire questi termini:

- Un parametro è la variabile elencata tra parentesi nella dichiarazione della funzione (fa parte della dichiarazione).
- Un argomento è il valore passato alla funzione quando viene chiamata (fa parte della chiamata).

Dichiariamo le funzioni elencando i loro parametri, quindi le chiamiamo passando gli argomenti.

Nell'esempio sopra, si potrebbe dire: "la funzione `showMessage` è dichiarata con due parametri, quindi viene chiamata con due argomenti: `from` and `"Hello"`".

## Valori di default

Se non viene fornito alcun parametro, questa assume il valore `undefined`.

Ad esempio, la funzione menzionata sopra `showMessage(from, text)` può essere chiamata con un solo argomento:

```
showMessage("Ann");
```

Questo non è un errore. Una chiamata simile mostrerà `"*Ann*: undefined"`. Siccome non viene passato nessun valore per `text`, questo è `undefined`.

Possiamo specificare un cosiddetto valore “default” (da usare se l’argomento è omesso) per i parametri nella dichiarazione di funzione, usando `=`:

```
function showMessage(from, text = "no text given") {
 alert(from + ": " + text);
}

showMessage("Ann"); // Ann: nessun text fornito
```

Adesso, se il parametro `text` non viene passato, assumerà il valore `"no text given"`

In questo caso `"no text given"` è una stringa, ma potrebbe essere un’espressione più complessa, che viene valutata e assegnata solamente se il parametro non viene fornito. Quindi, è possibile anche:

```
function showMessage(from, text = anotherFunction()) {
 // anotherFunction() viene eseguita solamente se non viene fornito text
 // il risultato diventa il valore di text
}
```

### Valutazione dei parametri di default

In JavaScript, un parametro di default viene valutato ogni volta che viene chiamata una funzione senza i rispettivo argomento.

Nell’esempio sopra, `anotherFunction()` non viene chiamata se viene passato il parametro `text`.

Viene invece chiamata ogni volta che il parametro manca.

A volte ha senso assegnare valori default ai parametri, non nella dichiarazione della funzione, ma in una fase successiva.

Possiamo verificare se il parametro viene passato durante l’esecuzione della funzione, confrontandolo con `undefined`:

```
function showMessage(text) {
 // ...

 if (text === undefined) { // if the parameter is missing
 text = 'empty message';
 }
}
```

```
 alert(text);
}

showMessage(); // empty message
```

...Oppure utilizzare l'operatore `||`:

```
function showMessage(from, text) {
 // se text è falso allora text assume il valore di default
 text = text || 'no text given';
 ...
}
```

I moderni motori JavaScript supportano il **nullish coalescing operator** `??`, più efficiente quando valori falsi come `0` vengono considerati regolari:

```
//se non c'è un parametro "count", mostra "unknown"
function showCount(count) {
 // if count is undefined or null, show "unknown"
 alert(count ?? "unknown");
}

showCount(0); // 0
showCount(null); // unknown
showCount(); // unknown
```

## Ritornare un valore

Una funzione può anche ritornare, come risultato, un valore al codice che ha effettuato la sua chiamata.

Un semplicissimo esempio è una funzione che somma due valori:

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

let result = sum(1, 2);
alert(result); // 3
```

La direttiva `return` può trovarsi in qualsiasi punto della funzione. Quando l'esecuzione incontra questa direttiva, si ferma, e il valore viene ritornato al codice che ha chiamato la funzione (nel codice sopra viene assegnato a `result`).

Ci possono essere più occorrenze di `return` in una singola funzione. Ad esempio:

```
function checkAge(age) {
 if (age >= 18) {
 return true;
 } else {
 return confirm('Do you have permission from your parents?');
```

```

 }
 }

let age = prompt('How old are you?', 18);

if (checkAge(age)) {
 alert('Access granted');
} else {
 alert('Access denied');
}

```

E' anche possibile utilizzare `return` senza alcun valore. Questo causerà l'uscita immediata dalla funzione.

Ad esempio:

```

function showMovie(age) {
 if (!checkAge(age)) {
 return;
 }

 alert("Showing you the movie"); // (*)
 // ...
}

```

Nel codice sopra, se `checkAge(age)` ritorna `false`, allora `showMovie` non procederà fino ad `alert`.

### **i Una funzione con un `return` vuoto ritorna `undefined`**

Se una funzione non ritorna alcun valore, è come se ritornasse `undefined`:

```

function doNothing() { /* vuoto */ }

alert(doNothing() === undefined); // true

```

Un `return` vuoto è dunque la stessa cosa di `return undefined`:

```

function doNothing() {
 return;
}

alert(doNothing() === undefined); // true

```

## Non aggiungere mai una nuova riga tra `return` e il valore

Per espressioni lunghe dopo la direttiva `return`, si potrebbe essere tentati dal metterle in una nuova riga, come:

```
return
(some + long + expression + or + whatever * f(a) + f(b))
```

Questo non funziona, perché JavaScript interpreta un punto e virgola dopo `return`. E' come se dopo `return` ci fosse scritto:

```
return;
(some + long + expression + or + whatever * f(a) + f(b))
```

Quindi, diventerebbe a tutti gli effetti un `return` vuoto. Dobbiamo quindi posizionare il valore da ritornare nella stessa riga.

## Denominare una funzione

Le funzioni sono azioni. Solitamente, per il loro nome vengono utilizzati verbi. Dovrebbero essere brevi, il più accurati possibili e descrittivi di ciò che la funzione fa. Un estraneo che legge il codice deve essere in grado di capire ciò che fa la funzione.

Una pratica molto diffusa è quella di iniziare il nome con un verbo, come prefisso, che descriva vagamente l'azione. Deve esserci un accordo con il team riguardo il significato dei prefissi.

Ad esempio, le funzioni che iniziano con `"show"` solitamente mostrano qualcosa.

Funzioni che iniziano per...

- `"get..."` – ritornano un valore,
- `"calc..."` – calcolano qualcosa,
- `"create..."` – creano qualcosa,
- `"check..."` – controllano qualcosa e ritornano un booleano, etc.

Esempi di nomi:

```
showMessage(..) // mostra un messaggio
getAge(..) // ritorna l'età (prendendola da qualche parte)
calcSum(..) // calcola la somma e ritorna il risultato
createForm(..) // crea un form (e solitamente lo ritorna)
checkPermission(..) // controlla i permessi, ritorna true/false
```

Tramite il prefisso, una semplice occhiata al nome della funzione dovrebbe far capire che tipo di lavoro eseguirà e quale sarà il tipo di valore ritornato.

## **i Una funzione – un’azione**

Una funzione dovrebbe fare esattamente ciò che il suo nome descrive, niente di più.

Due azioni separate solitamente meritano due funzioni diverse, anche se molto spesso vengono chiamate insieme (in questo caso potrebbe essere utile creare una terza funzione che chiama entrambe).

Un paio di esempi che non rispettano queste regole:

- `getAge` – sarebbe un pessimo nome se mostrasse un `alert` con l’età (dovrebbe solo restituirlo).
- `createForm` – sarebbe un pessimo nome se modificasse il documento, aggiungendo il `form` (dovrebbe solo crearlo e restituirlo).
- `checkPermission` – sarebbe un pessimo nome se mostrasse il messaggio `access granted/denied` (dovrebbe solo eseguire il controllo e ritornare il risultato).

Questi esempi assumono i significati comuni dei prefissi. Il loro significato dipende da voi e dal vostro team. E’ normale che il tuo codice abbia caratteristiche diverse, ma è fondamentale avere una ‘firma’ il cui prefisso sia sensato, che faccia capire cosa un determinato tipo di funzione può o non può fare. Tutte le funzioni che iniziano con lo stesso prefisso dovrebbero seguire determinate regole. E’ fondamentale che il team condivida queste informazioni.

## **i Nomi di funzioni ultra-corti**

Funzioni che vengono utilizzate *molto* spesso potrebbero avere nomi molto corti.

Ad esempio il framework [jQuery ↗](#) definisce una funzione con `$`. La libreria [Lodash ↗](#) ha nel core una funzione denominata `_`.

Queste sono eccezioni. Generalmente i nomi delle funzioni dovrebbero essere concisi e descrittivi.

## **Funzioni == Commenti**

Le funzioni dovrebbero essere brevi ed eseguire un solo compito. Se invece risultano lunghe, forse varrebbe la pena spezzarle in funzioni più piccole. Qualche volta può non essere semplice seguire questa regola, anche se sarebbe la cosa migliore.

Una funzione separata non è solo semplice da testare e correggere – la sua stessa esistenza è un commento!

Ad esempio, osserviamo le due funzioni `showPrimes(n)` sotto . Entrambe ritornano i numeri primi ↗ fino a `n` .

La prima versione utilizza le etichette:

```
function showPrimes(n) {
 nextPrime: for (let i = 2; i < n; i++) {

 for (let j = 2; j < i; j++) {
 if (i % j == 0) continue nextPrime;
 }
 }
}
```

```
 alert(i); // un primo
}
}
```

La seconda variante utilizza un funzione addizionale `isPrime(n)` per testare se un numero rispetta la condizione di essere primo:

```
function showPrimes(n) {

 for (let i = 2; i < n; i++) {
 if (!isPrime(i)) continue;

 alert(i); // un primo
 }
}

function isPrime(n) {
 for (let i = 2; i < n; i++) {
 if (n % i == 0) return false;
 }
 return true;
}
```

La seconda variante è più semplice da capire, non vi pare? Invece di un pezzo di codice vediamo il nome dell'azione (`isPrime`). Talvolta le persone si riferiscono a questo tipo di codice come *auto descrittivo*.

Quindi, le funzioni possono essere create anche se non abbiamo intenzione di riutilizzarle. Infatti forniscono una struttura migliore al codice e lo rendono più leggibile.

## Riepilogo

La dichiarazione di una funzione si scrive così:

```
function name(parameters, delimited, by, comma) {
 /* codice */
}
```

- Valori passati ad una funzione come parametri vengono copiati in variabili locali.
- Una funzione può avere accesso alle variabili esterne. Questo meccanismo funziona solo dall'interno verso l'esterno. Il codice esterno non può vedere le variabili locali ad una funzione.
- Una funzione può ritornare un valore. Se non lo fa esplicitamente, questo sarà `undefined`.

Per rendere il codice pulito e più facile da leggere, è consigliabile utilizzare principalmente variabili locali e parametri di funzione, non variabili esterne.

E' sempre più facile capire una funzione che accetta parametri, li lavora e ritorna un valore piuttosto di una funzione che non richiede parametri ma, come effetto collaterale, modifica variabili esterne.

Denominare le funzioni:

- Un nome dovrebbe descrivere chiaramente ciò che una funzione fa. Quando vediamo la chiamata di una funzione nel codice, se questa ha un buon nome ci farà immediatamente capire il suo comportamento e cosa ritornerà.
- Una funzione è un'azione, quindi i nomi delle funzioni utilizzano molto spesso dei verbi.
- Esistono molti prefissi comuni come `create...`, `show...`, `get...`, `check...` e molti altri. Utilizzateli per descrivere il comportamento di una funzione.

Le funzioni sono il principale blocco di un codice. Ora che abbiamo studiato le basi possiamo iniziare a crearle e utilizzarle. Ma questo è solo l'inizio. Ci ritorneremo molto spesso, andando molto in profondità, per capirne bene le caratteristiche.

## ✓ Esercizi

---

### E' richiesto "else"?

importanza: 4

La seguente funzione ritorna `true` se il parametro `age` è maggiore di `18`.

Altrimenti richiede una conferma e ritorna il risultato:

```
function checkAge(age) {
 if (age > 18) {
 return true;
 } else {
 // ...
 return confirm('Did parents allow you?');
 }
}
```

La funzione lavorerebbe diversamente se rimuovessimo `else` ?

```
function checkAge(age) {
 if (age > 18) {
 return true;
 }
 // ...
 return confirm('Did parents allow you?');
}
```

Ci sono differenze nel comportamento di queste due varianti?

[Alla soluzione](#)

---

### Riscrivi la funzione utilizzando '?' o '||'

importanza: 4

La seguente funzione ritorna `true` se il parametro `age` è maggiore di `18`.

Altrimenti richiede la conferma e ritorna il risultato.

```
function checkAge(age) {
 if (age > 18) {
 return true;
 } else {
 return confirm('Did parents allow you?');
 }
}
```

Riscrivila in modo che il comportamento sia uguale, ma senza utilizzare `if`. In una sola riga.

Fai due varianti di `checkAge`:

1. Utilizzando l'operatore `?`
2. Utilizzando OR `||`

[Alla soluzione](#)

---

## Funzione `min(a, b)`

importanza: 1

Scrivi una funzione `min(a, b)` che ritorna il valore più piccolo tra due numeri `a` e `b`.

Ad esempio:

```
min(2, 5) == 2
min(3, -1) == -1
min(1, 1) == 1
```

[Alla soluzione](#)

---

## Funzione `pow(x,n)`

importanza: 4

Scrivi una funzione `pow(x, n)` che ritorna la potenza `n` di `x`. In altre parole, moltiplica `x` per se stesso `n` volte e ritorna il risultato.

```
pow(3, 2) = 3 * 3 = 9
pow(3, 3) = 3 * 3 * 3 = 27
pow(1, 100) = 1 * 1 * ... * 1 = 1
```

Crea una pagina web che con un `prompt` richiede `x` e `n`, e in seguito mostra il risultato di `pow(x, n)`.

[Esegui la demo](#)

P.S. In questo esercizio è sufficiente che per `n` la funzione supporti solo i numeri naturali interi, a partire da `1`.

[Alla soluzione](#)

## Function expression

In JavaScript, una funzione non è una “struttura magica del linguaggio”, ma un valore speciale.

La sintassi che abbiamo utilizzato finora viene chiamata *Dichiarazione di Funzione*:

```
function sayHi() {
 alert("Hello");
}
```

E' disponibile un'altra sintassi per creare una funzione, chiamata *function expression*.

La sintassi:

```
let sayHi = function() {
 alert("Hello");
};
```

Qui la funzione viene esplicitamente creata ed assegnata ad una variabile, proprio come un qualsiasi altro valore. Non ha importanza come la funzione viene definita, è solo un valore salvato nella variabile `sayHi`.

Il significato di questo esempio è lo stesso di: "crea una funzione e mettila dentro la variabile `sayHi`".

Possiamo anche mostrarne il valore usando `alert`:

```
function sayHi() {
 alert("Hello");
}

alert(sayHi); // mostra il codice della funzione
```

Da notare che l'ultima riga di codice non esegue la funzione, perché non ci sono parentesi dopo `sayHi`. Ci sono linguaggi di programmazione in cui la semplice menzione del nome di una funzione ne causa l'esecuzione; JavaScript non funziona così.

In JavaScript, una funzione è un valore, quindi possiamo lavorarci come con un qualsiasi altro valore. Il codice sopra ne mostra la sua rappresentazione in stringa, cioè il codice contenuto dentro la funzione.

E' chiaramente un valore speciale, poiché possiamo richiamarla con `sayHi()`.

Ma rimane comunque un valore, e come tale possiamo trattarlo.

Possiamo copiare la funzione in un'altra variabile:

```
function sayHi() { // (1) creazione
 alert("Hello");
}

let func = sayHi; // (2) copia
```

```
func(); // Hello // (3) esegue la copia (funziona)!
sayHi(); // Hello // anche questa continua a funzionare (perché non dovrebbe?)
```

Quello che succede, nel dettaglio, è:

1. La dichiarazione di funzione (1) crea la funzione e la inserisce nella variabile denominata sayHi.
2. La linea (2) la copia nella variabile func. Ancora una volta: non ci sono parentesi dopo sayHi. Se ci fossero state, allora func = sayHi() avrebbe inserito *il risultato della chiamata* sayHi(), non *la funzione* sayHi.
3. Adesso la funzione può essere richiamata sia con sayHi() che con func().

Avremmo anche potuto utilizzare, nella prima riga, una function expression per dichiarare sayHi:

```
let sayHi = function() { // (1) create
 alert("Hello");
};

let func = sayHi;
// ...
```

Tutto funzionerebbe ugualmente. Risulta anche più chiaro cosa sta succedendo, giusto?

### ❶ Perché c'è un punto e virgola alla fine?

Vi starete chiedendo perché con la function expression bisogna mettere ; alla fine, mentre con la dichiarazione di funzione non serve:

```
function sayHi() {
 // ...
}

let sayHi = function() {
 // ...
};
```

La risposta è semplice:

- Non c'è bisogno di ; alla fine dei blocchi di codice che utilizzano una sintassi del tipo if { ... }, for { }, function f { } etc.
- Una function expression viene utilizzata all'interno di un'istruzione: let sayHi = ... ; , come valore. Non è un blocco di codice. Quindi il ; è consigliato al termine dell'istruzione, indipendentemente dal valore. Il punto e virgola non è collegato alla function expression; più semplicemente, termina un'istruzione.

## Callback functions (funzioni richiamate)

Diamo un'occhiata ad ulteriori esempi di passaggio di funzione come valore e utilizzo della sintassi function expression.

Scriveremo una funzione `ask(question, yes, no)` con tre parametri:

`question`

Il testo della domanda

`yes`

Funzione da eseguire se la risposta è “Yes”

`no`

Funzione da eseguire se la risposta è “No”

La funzione dovrebbe richiedere la `question` e, in base alla risposta dell’utente, chiamare `yes()` o `no()`:

```
function ask(question, yes, no) {
 if (confirm(question)) yes()
 else no();
}

function showOk() {
 alert("You agreed.");
}

function showCancel() {
 alert("You canceled the execution.");
}

// utilizzo: funzioni showOk, showCancel vengono passate come argomenti ad ask
ask("Do you agree?", showOk, showCancel);
```

Queste funzioni possono essere molto utili. La principale differenza tra un’implementazione realistica e gli esempi sopra è che le funzioni “reali” utilizzano modalità più complesse per interagire con l’utente, non un semplice `confirm`. In ambiente browser, queste funzioni mostrano spesso delle finestre molto carine per gli input dell’utente. Ma questo è un altro discorso.

Gli argomenti `showOk` e `showCancel` della `ask` sono chiamati *funzioni di richiamo* o semplicemente *callbacks*.

L’idea è di passare una funzione e di “richiamarla” più tardi se necessario. Nel nostro caso `showOk` diventa la callback per la risposta “yes”, e `showCancel` per la risposta “no”.

Possiamo utilizzare una function expression per scrivere la stessa funzione più concisamente:

```
function ask(question, yes, no) {
 if (confirm(question)) yes()
 else no();
}
```

```

ask(
 "Do you agree?",
 function() { alert("You agreed."); },
 function() { alert("You canceled the execution."); }
);

```

Qui la funzione viene dichiarata dentro alla chiamata di `ask( . . . )`. Queste non hanno nome, e perciò sono denominate *anonyme*. Queste funzioni non sono accessibili dall'esterno di `ask` (perché non sono assegnate a nessuna variabile), ma è proprio quel che vogliamo in questo caso.

Questo tipo di codice comparirà nei nostri script molto naturalmente: è nello spirito di JavaScript.

### **i Una funzione è un valore che rappresenta un “azione”**

I valori regolari come le stringhe o i numeri rappresentano *dati*.

Una funzione può anche essere vista come un’azione.

Possiamo passarla tra le variabili ed eseguirla quando vogliamo.

## Function expression vs Dichiarazione di Funzioni

Cerchiamo di elencare le differenze chiave tra Dichiarazioni ed Espressioni di Funzione.

Primo, la sintassi: come distinguerle nel codice.

- *Dichiarazione di funzione*: una funzione, dichiarata come un’istruzione separata, nel flusso principale del programma.

```

// Dichiarazione di funzione
function sum(a, b) {
 return a + b;
}

```

- *Function expression*: una funzione, creata all’interno di un’espressione o all’interno di un altro costrutto. Qui, la funzione è creata alla destra dell’ “espressione di assegnazione” `=` :

```

// function expression
let sum = function(a, b) {
 return a + b;
};

```

La differenza più subdola è quando una funzione viene creata dal motore JavaScript.

**Una function expression viene creata quando l’esecuzione la raggiunge ed è utilizzabile solo da quel momento in poi.**

Quando il flusso di esecuzione passa alla destra dell’operatore di assegnamento `let sum = function...`, la funzione viene creata e, a partire da questo momento, può essere utilizzata (assegnata, chiamata, etc...).

Una dichiarazione di funzione funziona diversamente.

**Una dichiarazione di funzione è utilizzabile nell'intero script/blocco di codice.**

In altre parole, quando JavaScript si *prepara* ad eseguire lo script o un blocco di codice, come prima cosa cerca le dichiarazioni di funzione e le crea. Possiamo pensare a questo processo come a uno “stage di inizializzazione”.

E’ solo dopo che tutte le dichiarazioni di funzione sono state processate che l’esecuzione potrà procedere.

Come risultato, una funzione creata tramite una dichiarazione di funzione può essere richiamata anche prima della sua definizione.

Ad esempio, il seguente codice funziona:

```
sayHi("John"); // Hello, John

function sayHi(name) {
 alert(`Hello, ${name}`);
}
```

La dichiarazione di funzione `sayHi` viene creata quando JavaScript si sta preparando ad eseguire lo script ed è visibile in ogni suo punto.

...Se fosse stata una function expression, non avrebbe funzionato:

```
sayHi("John"); // errore!

let sayHi = function(name) { // (*) nessuna magia
 alert(`Hello, ${name}`);
};
```

Le espressioni di funzione sono create quando l’esecuzione le incontra. In questo esempio avverrà solo alla riga `(*)`. Troppo tardi.

**In strict mode, quando una dichiarazione di funzione viene fatta all’interno di un blocco di codice sarà visibile ovunque all’interno del blocco, ma non al suo esterno.**

Qualche volta è comodo dichiarare funzioni locali, utili in un singolo blocco. Ma questa caratteristica potrebbe causare problemi.

Ad esempio, immaginiamo di aver bisogno di dichiarare una funzione `welcome()` (la utilizzeremo più avanti) in base ad un parametro `age` che valuteremo durante il *runtime*.

Il codice sotto non funziona:

```
let age = prompt("What is your age?", 18);

// dichiarazione di funzione condizionale
if (age < 18) {

 function welcome() {
 alert("Hello!");
 }
}
```

```

} else {

 function welcome() {
 alert("Greetings!");
 }

}

// ...utilizzo successivo
welcome(); // Errore: welcome non è definita

```

Questo accade perché una dichiarazione di funzione è visibile solamente all'interno del blocco di codice in cui è stata definita.

Un altro esempio:

```

let age = 16; // prendiamo 16 come esempio

if (age < 18) {
 welcome(); // \ (esegue)
 // |
 function welcome() { // |
 alert("Hello!"); // | Dichiaraione di funzione disponibile
 } // | ovunque nel blocco in cui è stata dichiarata
 // |
 welcome(); // / (esegue)

} else {

 function welcome() {
 alert("Greetings!");
 }
}

// Qui siamo all'esterno delle parentesi graffe,
// quindi non possiamo vedere le dichiarazioni di funzione fatte al suo interno.

welcome(); // Errore: welcome non è definita

```

Cosa possiamo fare per rendere visibile `welcome` all'esterno del blocco `if`?

Il giusto approccio è quello di utilizzare una function expression ed assegnarla ad una variabile `welcome`, che viene dichiarata all'esterno di `if` ed ha quindi la corretta visibilità.

Ora funziona:

```

let age = prompt("What is your age?", 18);

let welcome;

if (age < 18) {

 welcome = function() {
 alert("Hello!");
 };
}

```

```

} else {

 welcome = function() {
 alert("Greetings!");
 };

}

welcome(); // ora funziona

```

Oppure possiamo semplificarla con l'utilizzo dell'operatore `?`:

```

let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?
 function() { alert("Hello!"); } :
 function() { alert("Greetings!"); };

welcome(); // ora funziona

```

### **i Quando conviene scegliere una dichiarazione di funzione piuttosto di una function expression?**

Come regola fondamentale, quando abbiamo la necessità di dichiarare una funzione, la prima opzione da considerare è la dichiarazione di funzione. Fornisce maggiore libertà per quanto riguarda l'organizzazione del codice, poiché possiamo utilizzare la funzione anche prima della sua dichiarazione.

Risulta anche più facile vedere `function f(...){...}`, nel codice, piuttosto di `let f = function(...){...}`. La dichiarazione di funzione è più facile da individuare.

...Ma se per qualche ragione la dichiarazione di funzione non si applica bene al caso in questione (abbiamo visto degli esempi, sopra), allora la function expression può essere un'alternativa.

## Riepilogo

- Le funzioni sono valori. Possono essere assegnate, copiate o dichiarate in qualsiasi punto del codice.
- Se la funzione viene dichiarata come un blocco “separato” dal flusso d'esecuzione principale del codice, tale definizione viene chiamata “dichiarazione di funzione”.
- Se la funzione viene definita tramite un'espressione, viene chiamata “function expression”.
- Le dichiarazioni di funzione vengono processate prima che il blocco di codice dove sono state definite sia raggiunto.
- Le function expression vengono processate quando il flusso d'esecuzione del codice principale le raggiunge.

Nella maggior parte dei casi, quando abbiamo bisogno di dichiarare una funzione una dichiarazione di funzione è preferibile poiché è visibile anche prima della sua dichiarazione. Questo ci fornisce più flessibilità nell'organizzazione del codice, e solitamente risulta più leggibile.

Dovremmo, quindi, utilizzare una function expression solo quando una dichiarazione di funzione non è adatta a un caso specifico. Abbiamo visto un paio di esempi in questo capitolo, e ne vederemo altri in futuro.

## Arrow functions, le basi

Esiste un'altra sintassi molto semplice e concisa per creare funzioni e che spesso è migliore delle Function Expressions.

E' chiamata "arrow functions", perché si presenta in questo modo:

```
let func = (arg1, arg2, ..., argN) => expression;
```

Questo codice crea una funzione `func` che accetta gli argomenti `arg1..argN` e li utilizza per valutare `expression` e restituirne il risultato.

In altre parole è una versione abbreviata di:

```
let func = function(arg1, arg2, ..., argN) {
 return expression;
};
```

Vediamo un esempio concreto:

```
let sum = (a, b) => a + b;

/* Questa arrow function è una versione abbreviata di:

let sum = function(a, b) {
 return a + b;
};

alert(sum(1, 2)); // 3
```

Come puoi vedere `(a, b) => a + b` rappresenta una funzione che accetta due argomenti `a` e `b`. Al momento dell'esecuzione, questa valuta l'espressione `a + b` e restituisce il risultato.

- Se abbiamo un solo argomento, le parentesi che racchiudono gli argomenti possono essere omesse, abbreviando ulteriormente il codice.

Ad esempio:

```
let double = n => n * 2;
// più o meno lo stesso di: let double = function(n) { return n * 2 }

alert(double(3)); // 6
```

- Se non ci sono argomenti, le parentesi saranno vuote (ma devono essere presenti):

```
let sayHi = () => alert("Hello!");
sayHi();
```

Le arrow functions possono essere usate allo stesso modo delle Function Expressions.

Ad esempio, per creare dinamicamente una funzione:

```
let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?
 () => alert('Hello') :
 () => alert("Greetings!");

welcome();
```

Le arrow functions possono apparire poco familiari e leggibili all'inizio, ma ciò cambia rapidamente man mano che gli occhi si abitueranno alla struttura.

Esse sono molto comode per semplici azioni su una riga, se siamo troppo pigri per scrivere più parole.

## Arrow functions su più linee

Gli esempi precedenti hanno preso argomenti alla sinistra di “=>” e con essi hanno valutato l'espressione a destra.

A volte abbiamo bisogno di qualcosa di un po' più complesso, come espressioni o dichiarazioni multiple. Anche questo è possibile, ma dovremo racchiuderle tra parentesi graffe ed usare un normale return.

In questo modo:

```
let sum = (a, b) => { // le parentesi graffe aprono una funzione multilinea
 let result = a + b;
 return result; // se usiamo le parentesi graffe abbiamo bisogno di un esplicito "return"
};

alert(sum(1, 2)); // 3
```

### Molto di più...

Qui abbiamo presentato le arrow functions in breve, ma questo non è tutto!

Le arrow functions possiedono altre interessanti caratteristiche.

Per studiarle approfonditamente dobbiamo prima conoscere qualche altro aspetto di JavaScript, quindi torneremo alle arrow functions più avanti, nel capitolo [Arrow functions rivisitate](#).

Per ora possiamo già utilizzarle per azioni su una riga sola e per callbacks.

## Summary

Le arrow functions sono utili per azioni su una riga sola. Possono essere scritte in due modi:

1. Senza parentesi graffe: `(...args) => expression` – la parte destra è un'espressione: la funzione la valuta e restituisce il risultato.
2. Con parentesi graffe: `(...args) => { body }` – le parentesi ci permettono di scrivere comandi multipli all'interno della funzione, ma abbiamo bisogno di dichiarare esplicitamente `return` affinché sia ritornato qualcosa.

## ✓ Esercizi

### Riscrivi usando le arrow functions

Sostituisci le function expressions con arrow functions:

```
function ask(question, yes, no) {
 if (confirm(question)) yes();
 else no();
}

ask(
 "Do you agree?",
 function() { alert("You agreed."); },
 function() { alert("You canceled the execution."); }
);
```

[Alla soluzione](#)

## Specialità di JavaScript

Questo capitolo ricapitola brevemente le caratteristiche di JavaScript apprese fino ad ora, prestando particolare attenzioni ai punti più sottili.

### Struttura del codice

Le istruzioni sono delimitate da un punto e virgola:

```
alert('Hello'); alert('World');
```

Solitamente, un “a capo” viene considerato come nuova riga, quindi questo codice funzionerebbe ugualmente:

```
alert('Hello')
alert('World')
```

Questo viene definito “inserimento automatico del punto e virgola”. In qualche caso non funziona, ad esempio:

```
alert("There will be an error after this message")
[1, 2].forEach(alert)
```

Molte linee guida che descrivono lo stile del codice consigliano di mettere un punto e virgola alla fine di ogni istruzione.

Il punto e virgola non è richiesto dopo un blocco di codice `{ . . . }` e i costrutti sintattici che le utilizzano, come i cicli:

```
function f() {
 // non è richiesto il punto e virgola dopo la dichiarazione di funzione
}

for(;;) {
 // non è richiesto il punto e virgola dopo il ciclo
}
```

...Anche se mettessimo un punto e virgola “extra” da qualche parte, non sarebbe un errore. Sarà semplicemente ignorato.

Più dettagli: [Struttura del codice](#).

## Strict mode

Per abilitare completamente tutte le caratteristiche del moderno JavaScript, dovremmo iniziare lo script con `"use strict"`.

```
'use strict';
```

```
...
```

La direttiva deve essere posta all'inizio di ogni script o all'inizio di una funzione.

Senza `"use strict"`, tutto continuerebbe a funzionare, ma alcune caratteristiche si comporterebbero in vecchio-stile, per retrocompatibilità. Generalmente si preferisce la modalità con i comportamenti moderni.

Alcune caratteristiche moderne del linguaggio (come le classi che studieremo più avanti) attivano automaticamente la modalità script.

Di più: [Le tecniche moderne, "use strict"](#).

## Variabili

Possono essere dichiarate utilizzando:

- `let`
- `const` (costante, non può essere modificata)
- `var` (vecchio stile, lo vedremo più avanti)

Il nome di una variabile può includere:

- Lettere e numeri, il primo carattere non può però essere un numero.
- I caratteri come `$` e `_` vengono considerati normalmente, come se fossero lettere.
- Alfabeti non-latini e geroglifici sono comunque consentiti, ma non vengono comunemente utilizzati.

Le variabili vengono tipizzate dinamicamente. Possono memorizzare qualsiasi valore:

```
let x = 5;
x = "John";
```

Ci sono 7 tipi di dato:

- `number` si per i numeri in virgola mobile, che per quelli interi,
- `bignum` per valori numerici di lunghezza arbitraria,
- `string` per le stringhe,
- `boolean` per i valori logici: `true/false`,
- `null` – un tipo con un singolo valore `null`, che ha il significato di “vuoto” o “non esistente”,
- `undefined` – un tipo con un singolo valore `undefined`, che significa “non assegnato”,
- `object` e `symbol` – per strutture dati più complesse e identificatori unici, non li abbiamo ancora studiati.

L'operatore `typeof` ritorna il tipo di un valore, con due eccezioni:

```
typeof null == "object" // errore nel linguaggio
typeof function(){} == "function" // le funzioni vengono trattate diversamente
```

Di più in: [Variabili](#) e [Tipi di dato](#).

## Interazioni

Abbiamo utilizzato solo il browser come ambiente di sviluppo, quindi le interfacce di base saranno:

`prompt(question[, default])`

Pone una domanda `question`, e ritorna quello che l'utente ha inserito oppure `null` se ha premuto “cancel”.

`confirm(question)`

Pone una domanda `question` e fornisce la possibilità di scegliere tra Ok e Cancel. La scelta viene ritornata come `true/false`.

`alert(message)`

Stampa un messaggio `message`.

Tutte queste funzioni sono dei *modal*, interrompono l'esecuzione e impediscono all'utente di interagire con una pagina fino a che il visitatore non risponde.

Ad esempio:

```
let userName = prompt("Your name?", "Alice");
let isTeaWanted = confirm("Do you want some tea?");

alert("Visitor: " + userName); // Alice
alert("Tea wanted: " + isTeaWanted); // true
```

Di più: [Interazioni: alert, prompt, confirm.](#)

## Operatori

JavaScript supporta i seguenti operatori:

### Aritmetici

Regolari: `*` `+` `-` `/`, anche `%` per la divisione con resto e `**` per le potenze.

La somma binaria `+` che concatena stringhe. E se almeno uno degli operandi è una stringa, anche gli altri vengono convertiti a stringa:

```
alert('1' + 2); // '12', string
alert(1 + '2'); // '12', string
```

### Assegnazione

Ci sono le assegnazioni semplici: `a = b` e quelle combinate `a *= 2`.

### Bit a Bit

Gli operatori bit a bit funzionano con gli interi a livello di bit: guarda la [documentazione ↗](#) se ne avrai bisogno.

### Ternari

C'è un solo operatore con tre parametri: `cond ? resultA : resultB`. Se `cond` è vera, ritorna `resultA`, altrimenti `resultB`.

### Operatori logici

AND logico `&&` e OR `||` eseguono delle valutazioni locali e ritornano un valore quando si fermano. La negazione logica NOT `!` converte il valore a tipo booleano e ne ritorna l'inverso.

### Operatore di nullità

L'operatore `??` fornisce un modo per scegliere un valore definito tra una lista di valori. Il risultato dell'espressione `a ?? b` sarà sempre `a`, salvo che non sia `null/undefined`, in questo caso il risultato sarà `b`.

### Confronto

Confronto di uguaglianza `==` valori di tipi diversi vengono convertiti in numeri (ad eccezione di `null` e `undefined` che si egualano tra di loro e con nient'altro), quindi questi sono uguali:

```
alert(0 == false); // true
alert(0 == '0'); // true
```

Anche gli altri confronti convertono i valori in numeri.

L'operatore di uguaglianza stretta `==` non esegue la conversione: tipi differenti vengono interpretati com valori differenti.

I valori `null` e `undefined` sono speciali: sono uguali `==` tra di loro ma non con nessun altro.

Maggiore/minore confrontano le stringhe carattere per carattere, gli altri valori vengono convertiti a numeri.

## Operatori logici

Ce ne sono altri, come l'operatore virgola.

Di più in: [Operatori di base](#), [Confronti](#), [Operatori logici](#), [Nullish coalescing operator '??'](#).

## Cicli

- Abbiamo studiato tre tipi di ciclo:

```
// 1
while (condition) {
 ...
}

// 2
do {
 ...
} while (condition);

// 3
for(let i = 0; i < 10; i++) {
 ...
}
```

- La variabile dichiarata nel ciclo `for(let ...)` è visibile solo internamente al ciclo. Possiamo anche omettere `let` e riutilizzare una variabile esistente.
- Le direttive `break/continue` permettono di uscire dall'intero ciclo/iterazione. Con l'utilizzo di etichette si possono saltare cicli annidati.

Maggiori dettagli in: [Cicli: while e for](#).

Più avanti studieremo più tipi di ciclo per lavorare con gli oggetti.

## Il costrutto “switch”

Il costrutto “switch” può rimpiazzare controlli multipli con `if`. Utilizza `==` (uguaglianza stretta) per i confronti.

Ad esempio:

```

let age = prompt('Your age?', 18);

switch (age) {
 case 18:
 alert("Won't work"); // il risultato di prompt è una stringa, non un numero
 break;

 case "18":
 alert("This works!");
 break;

 default:
 alert("Any value not equal to one above");
}

```

Più dettagli in: L'istruzione "switch".

## Funzioni

Abbiamo studiato tre modi per creare funzioni in JavaScript:

1. Dichiarazione di funzione: la funzione nel flusso principale

```

function sum(a, b) {
 let result = a + b;

 return result;
}

```

2. Espressione di funzione: la funzione nel contesto di un'espressione

```

let sum = function(a, b) {
 let result = a + b;

 return result;
};

```

Le espressioni di funzione possono avere un nome, come `sum = function name(a, b)`, questo `name` è visibile solamente all'interno della funzione.

3. Funzione freccia:

```

// espressione dalla parte destra
let sum = (a, b) => a + b;

// oppure la sintassi multi-riga con { ... }, è necessario esplicitare return
let sum = (a, b) => {
 // ...
 return a + b;
}

// senza argomenti
let sayHi = () => alert("Hello");

```

```
// con un solo argomento
let double = n => n * 2;
```

- Le funzioni possono avere variabili locali: queste vengono dichiarate all'interno della funzione stessa o nella lista parametri. Queste variabili sono visibili solamente all'interno della funzione.
- I parametri possono avere valori di default: `function sum(a = 1, b = 2) { . . . }`.
- Le funzioni ritornano sempre qualcosa. Se non c'è nessuna istruzione `return`, allora viene restituito `undefined`.

Per più informazioni: vedi [Funzioni, Arrow functions, le basi](#).

## C'è di più

Questa era solo una breve lista delle caratteristiche di JavaScript. Per ora abbiamo studiato solo le basi. Più avanti nel tutorial troverai caratteristiche più avanzate di JavaScript.

## Qualità del codice

Impareremo le buone pratiche di programmazione da utilizzare durante lo sviluppo.

## Debugging in the browser

Prima di scrivere codice più complesso, dovremmo parlare di debugging.

[Debugging ↗](#) è il processo che prevede la ricerca e la risoluzione degli errori all'interno di uno script. Tutti i browser moderni e molti altri ambienti forniscono strumenti per il debugging – degli speciali strumenti che rendono il debugging un'operazione più semplice. Consentono anche di seguire l'esecuzione del codice passo per passo, per capire esattamente cosa sta accadendo...

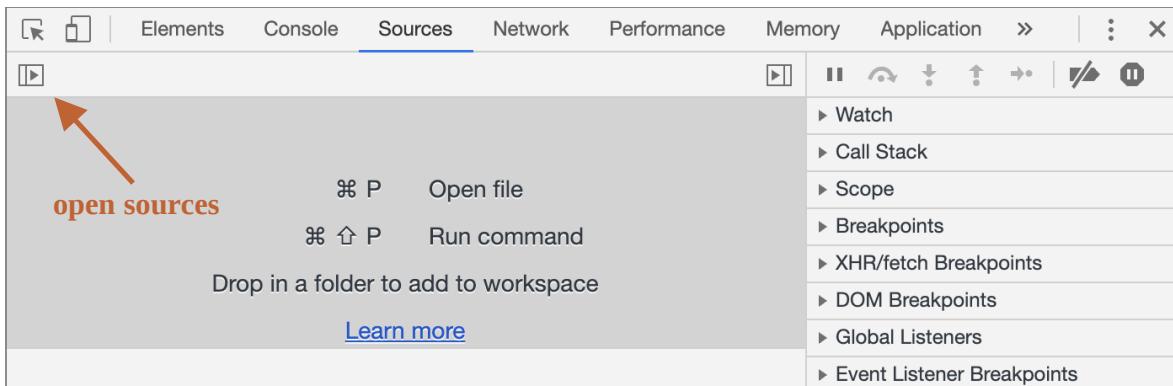
Noi useremo Chrome, poiché è probabilmente il più ricco di caratteristiche sotto questo aspetto.

## Il pannello “sources”

La tua versione di Chrome potrebbe essere differente, ma le funzioni principali dovrebbero essere molto simili.

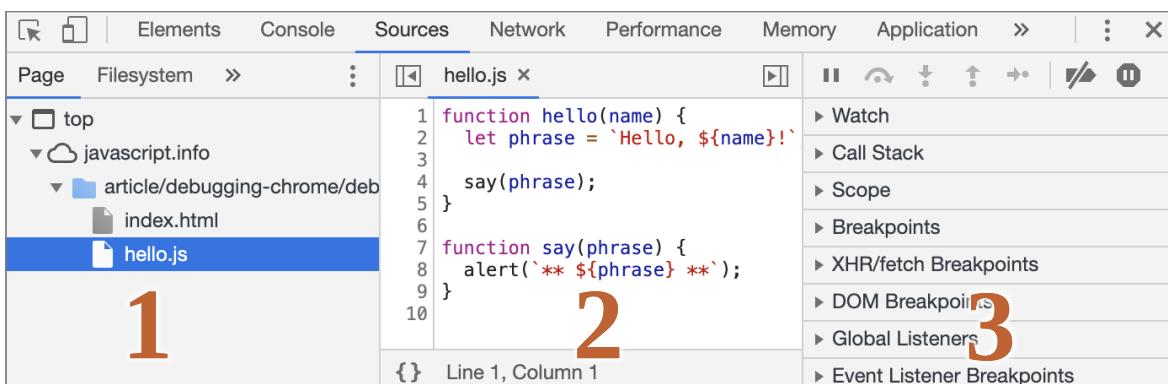
- Apri la [pagina di esempio](#) in Chrome.
- Attiva gli strumenti da sviluppatore con `F12` (Mac: `Cmd+Opt+I`).
- Seleziona il pannello `sources`.

Questo è quello che dovreste vedere se è la prima volta che lo aprirete:



Il bottone apre la barra laterale con i file.

Clicchiamoci sopra e selezioniamo `hello.js` dalla vista ad albero. Questo è quello che dovrebbe apparire:



Possiamo vedere tre zone:

1. La **zona Risorse** con la lista degli HTML, JavaScript, CSS e altri file, incluse le immagini che sono collegate alla pagina. Potrebbero apparire anche le estensioni di Chrome.
2. La **zona Sorgente** mostra il codice.
3. La **zona Informazione e controllo** utile per il debugging, la esploreremo meglio.

Ora puoi cliccare nuovamente lo stesso bottone per nascondere la lista risorse e dare più spazio al codice.

## Console

Se premiamo `Esc`, si apre una console in basso. Possiamo digitare comandi e premere `Enter` per eseguirli.

Dopo l'esecuzione dell'istruzione, il risultato viene mostrato sotto.

Ad esempio, `1+2` con risultato `3`, ed `hello("debugger")` non ritorna nulla, quindi il risultato è `undefined`:



## Breakpoint

Esaminiamo cosa sta succedendo nel codice della [pagina di esempio](#). In `hello.js`, cliccate nel numero della riga 4. Si, cliccate proprio sopra il numero 4, non dentro il codice.

Congratulazioni! Avete settato un breakpoint. Ora premete anche nella riga numero 8.

Dovrebbe apparire qualcosa di simile (in blu dove avreste dovuto cliccare):

```
function hello(name) {
 let phrase = `Hello, ${name}!`;
 say(phrase);
}
function say(phrase) {
 alert(`** ${phrase} **`);
}
```

here's the list

- ▶ Watch
- ▶ Call Stack
- ▶ Scope
- ▼ Breakpoints
  - hello.js:4  
say(phrase);
  - hello.js:8  
alert(`\*\* \${phrase} \*\*`);
- ▶ XHR/fetch Breakpoints

Un *breakpoint* è un punto del codice in cui il debugger si metterà in pausa automaticamente durante l'esecuzione del codice JavaScript.

Mentre il codice è in pausa, è possibile esaminare le variabili, eseguire comandi tramite la console etc. In altre parole, stiamo facendo debugging.

Possiamo anche visualizzare la lista dei breakpoint nel pannello di destra. Questo pannello può risultare utile quando abbiamo più breakpoint in file diversi. Infatti ci consente di:

- Saltare rapidamente ad un breakpoint (cliccando sopra al nome del breakpoint che ci interessa).
- Disabilitare temporaneamente un breakpoint semplicemente togliendo la spunta.
- Rimuovere breakpoint cliccando con il tasto destro e selezionando Rimuovi.
- ...E molto altro.

### **i** Breakpoint condizionali

Tasto destro sul numero della riga ci consente di creare un breakpoint *condizionale*. Che viene attivato solo quando l'espressione fornita risulta vera.

Questa caratteristica risulta molto utile quando abbiamo bisogno di fermare il flusso di esecuzione per determinati valori di una variabile.

## Comando debugger

Possiamo mettere in pausa il codice anche utilizzando il comando `debugger`, come nell'esempio:

```
function hello(name) {
 let phrase = `Hello, ${name}!`;

 debugger; // <-- il codice si metterà in pausa qui
```

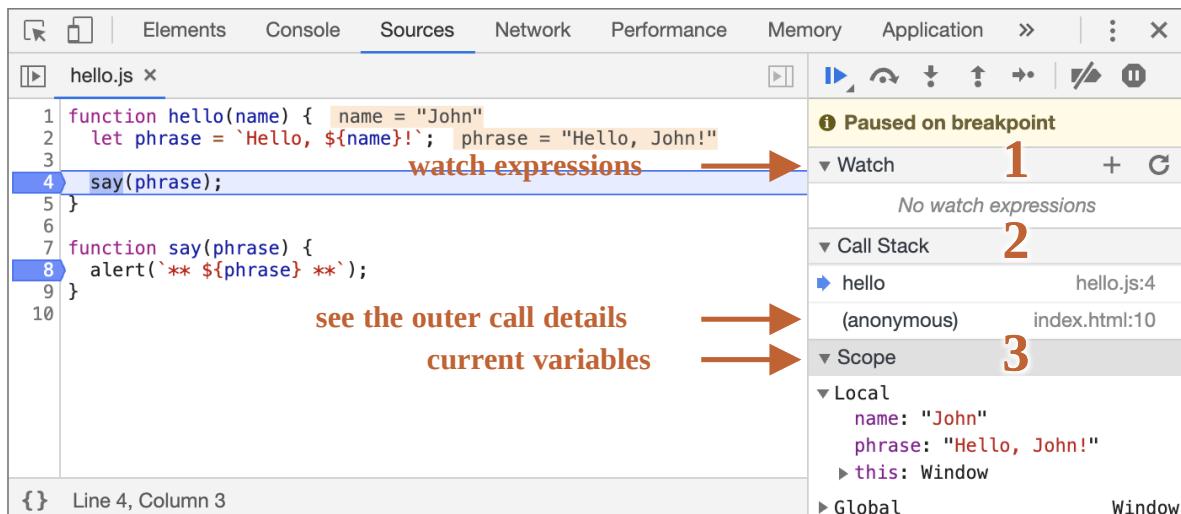
```
 say(phrase);
}
```

Questo risulta molto utile quando stiamo lavorando in un editor e non vogliamo passare alla finestra del browser, cercare il punto corretto nello script interessato e impostare il breakpoint.

## Interrompere l'esecuzione e guardarsi intorno

Nel nostro esempio, `hello()` viene richiamato durante il caricamento della pagina, quindi il metodo più facile per attivare il debugger è ricaricare la pagina. Quindi premete `F5` (Windows, Linux) o `Cmd+R` (Mac).

Con il breakpoint impostato, l'esecuzione si fermerà alla quarta linea:



Ora aprete il menu a cascata (quello con la freccetta accanto al nome). Ti consentirà di esaminare lo stato corrente del codice:

### 1. `Watch` – mostra il valore corrente per ogni espressione.

Puoi cliccare su `+` e inserire un'espressione. Il debugger ti mostrerà il suo valore ad ogni istante, che verrà automaticamente ricalcolato durante l'esecuzione.

### 2. `Call Stack` – mostra la catena delle chiamate annidate.

Attualmente il debugger si trova all'interno della chiamata `hello()`, chiamata da uno script interno a `index.html` (non ci sono funzioni qui, quindi viene definita "anonima").

Se premi su un elemento della pila, il debugger salterà al codice corrispondente, e potrai esaminare tutte le variabili.

### 3. `Scope` – variabili correnti.

`Local` mostra le variabili locali alla funzione. Potrete anche vedere i valori evidenziati nel codice.

`Global` mostra le variabili globali (fuori da tutte le funzioni).

C'è anche la keyword `this`, che studieremo più avanti.

## Tracciamento dell'esecuzione

Ora è il momento di tracciare lo script.

Ci sono dei bottoni appositi nella parte superiore del pannello di destra. Proviamo ad attivarli.

► – “Resume”: continua l’esecuzione, tasto **F8**.

Riprende l’esecuzione. Se non ci sono ulteriori breakpoint l’esecuzione continua e il debugger non avrà più il controllo.

Questo è quello che vedremo dopo aver cliccato:

The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. A file named 'hello.js' is open, showing the following code:

```
1 function hello(name) {
2 let phrase = `Hello, ${name}!`;
3
4 say(phrase);
5 }
6
7 function say(phrase) { phrase = "Hello, John!"
8 alert(`** ${phrase} **`);
9 }
10
```

A blue line highlights the 8th line of code: `alert(`\*\* \${phrase} \*\*`);`. The word 'nested calls' is written in orange above the code, with a red arrow pointing towards the call stack. The call stack on the right shows:

- Paused on breakpoint
- Watch: No watch expressions
- Call Stack:
  - say (hello.js:8)
  - hello (hello.js:4)
  - (anonymous) index.html:10
- Scope
- Local:
  - phrase: "Hello, John!"
  - this: Window
- Global: Window

Line 8, Column 3 is indicated at the bottom left.

L’esecuzione è ripartita, ha incontrato un altro breakpoint dentro `say()` e si è fermata nuovamente. Diamo un’occhiata al “Call stack” sulla destra. E’ stato incrementato con un’ulteriore chiamata. Ora siamo all’interno di `say()`.

→ – “Step”: esegue il prossimo comando, hotkey **F9**.

Esegue la prossima istruzione. Se lo clicchiamo, verrà mostrato l’`alert`.

Continuando a cliccare eseguiremo lo script un passo per volta.

⇨ – “Step over”: esegue il prossimo comando, ma *non entra nella funzione*, hotkey **F10**.

Molto simile al comando “Step”, ma si comporta diversamente nel caso in cui l’istruzione successiva sia una chiamata a funzione. O meglio: non una funzione built-in come `alert`, ma una funzione definita da noi.

Il comando “Step” entra nella funzione e mette in pausa l’esecuzione, mentre “Step over” esegue la chiamata a funzione, saltandone il contenuto.

L’esecuzione viene interrotta al termine della chiamata.

Questo è molto utile se non siamo interessati nel vedere cosa accade dentro la funzione.

↳ – fai uno step, tasto **F11**.

Lo stesso di quello precedente, ma fa “un passo all’interno” della funzione. Cliccando qui è possibile avanzare uno step alla volta tutte le azioni dello script.

↓ – “Step into”, hotkey **F11**.

Molto simile a “Step”, ma si comporta in maniera differente nel caso di chiamate a funzioni asincrone. Se state ancora imparando JavaScript, allora potete ignorare la differenza, visto che

non andremo ad utilizzare funzioni asincrone per ora.

Per il futuro, ricordatevi che “Step” ignora le funzioni asincrone, come ad esempio `setTimeout` (chiamata programmata di funzione). Invece “Step into” accede a questo tipo di funzioni, ed attende quanto necessario. Fate riferimento al [manuale DevTools ↗](#) per maggiori dettagli.

**↑ – “Step out”: continua l’esecuzione fino alla fine della funzione corrente, hotkey `Shift+F11`.**

Continua l’esecuzione e la interrompe all’ultima linea della funzione corrente. Questo comando risulta essere utile nel caso in cui entrassimo accidentalmente in una chiamata annidata utilizzando `→`, ma non siamo interessati alla sua esecuzione, e vogliamo arrivare al termine della sua esecuzione il prima possibile.

**+ – attiva/disattiva tutti i breakpoint.**

Questo bottone non influenza l’esecuzione. E’ semplicemente un on/off per i breakpoint.

**► – attiva/disattiva la pausa automatica in caso di errori.**

Quando questa opzione è attiva, e il pannello degli strumenti sviluppatore è aperto, un errore nello script metterà automaticamente in pausa l’esecuzione. Così potremmo analizzare le variabili per capire cosa è andato storto. Quindi se il nostro script si blocca con un errore, possiamo aprire il debugger, attivare questa opzione e ricaricare la pagina per vedere dove si blocca lo script e capirne il motivo.

**i Continua fino a qui**

Premendo tasto destro su una riga di codice si aprirà un menu con una bellissima opzione denominata “Continua fino a qui”.

Questa è molto utile quando vogliamo muoverci di più passi, ma siamo troppo pigri per impostare un breakpoint.

## Logging

Per stampare qualcosa sulla console, possiamo utilizzare la funzione `console.log`.

Ad esempio, questo stamperà i valori da `0` a `4` sulla console:

```
// apri la console per vedere il messaggio
for (let i = 0; i < 5; i++) {
 console.log("value,", i);
}
```

Gli utenti normali non vedranno questo output poiché viene mostrato in console. Per vederlo dovrebbero aprire il menu `Console` degli strumenti sviluppatore, oppure premere `Esc` mentre si trovano in un altro tab: questo tasto aprirà la console nella parte inferiore della schermata.

Utilizzando la funzione `console.log` nel nostro codice, possiamo vedere cosa sta accadendo anche senza utilizzare il debugger.

## Riepilogo

Come abbiamo visto, ci sono tre diversi modi di mettere in pausa uno script:

1. Un breakpoint.
2. L'istruzione `debugger`.
3. Un errore (solo se gli strumenti sviluppatore sono aperti ed è attivo il bottone ▶ )

Così possiamo esaminare le variabili e capire cosa è andato male durante l'esecuzione.

Ci sono veramente troppe opzioni negli strumenti da sviluppatore per coprirle qui. Il manuale completo è disponibile all'indirizzo <https://developers.google.com/web/tools/chrome-devtools>.

Le informazioni fornite da questo capitolo sono sufficienti per iniziare a fare un po' di debug, anche se più avanti, specialmente quando farete cose più avanzate, il link sopra potrà tornarvi utile.

Ah, inoltre potete anche cliccare i vari button negli strumenti da sviluppatore e vedere cosa succede. È probabilmente il metodo più rapido per imparare. Non dimenticate che molte opzioni sono disponibili anche con il click del tasto destro!

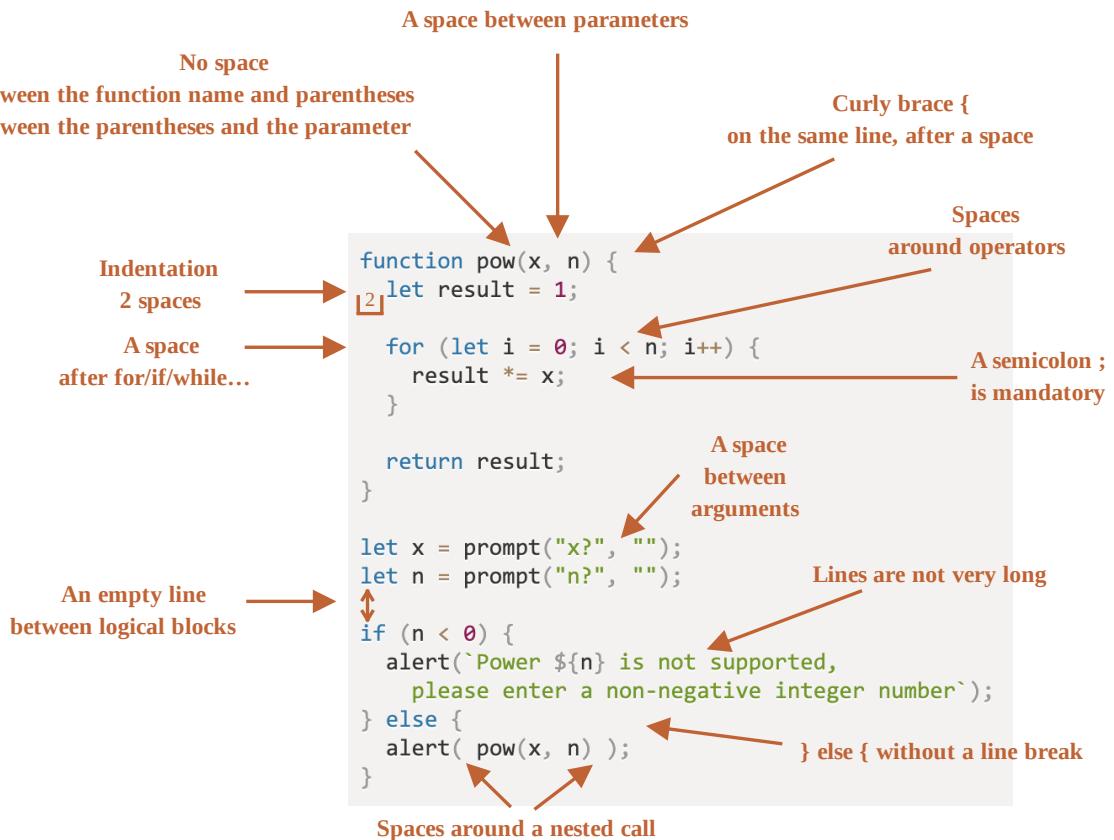
## Stile di programmazione

Il codice dovrebbe essere il più pulito e leggibile possibile.

La programmazione è come un arte – dobbiamo essere in grado di programmare funzionalità complesse che siano allo stesso tempo corrette e leggibili. Un corretto stile di programmazione aiuta molto in questo senso.

## Sintassi

Qui una lista di alcune regole da seguire (guardare sotto per maggiori dettagli):



Adesso discutiamo le regole e le rispettive motivazioni nel dettaglio.

### ⚠ Non ci sono delle regole obbligatorie

Niente di quello che stiamo dicendo è scolpito sulla pietra. Sono solo delle preferenze, non dei dogmi religiosi.

## Parentesi graffe

In molti progetti JavaScript le parentesi graffe sono scritte seguendo lo stile “Egiziano” con la parentesi aperta nella stessa linea della keyword – non in una nuova linea. Dovrebbe comunque esserci spazio prima della parentesi aperta, come in questo esempio:

```
if (condition) {
 // fai questo
 // ...e questo
 // ...e questo
}
```

Un caso limite è un costrutto con una singola linea. Dovremmo comunque usare le parentesi? Se si come?

Qui ci sono un paio di varianti, così potete giudicare voi stessi:

1. 😞 I principianti spesso fanno questo. Sbagliato! Le parentesi graffe non sono richieste:

```
if (n < 0) {alert(`Power ${n} is not supported`);}
```

2. ☹ Inserire l'istruzione su nuova riga senza parentesi. Non dovreste mai farlo, poiché rende molto semplice commettere errori nel caso si volesse aggiungere un'altra istruzione:

```
if (n < 0)
 alert(`Power ${n} is not supported`);
```

3. ☺ In una singola riga senza parentesi – accettabile, se l'istruzione è semplice:

```
if (n < 0) alert(`Power ${n} is not supported`);
```

4. ☺ La variante migliore:

```
if (n < 0) {
 alert(`Power ${n} is not supported`);
}
```

Per istruzioni molto brevi, è consentito scrivere su una sola riga, ad esempio `if (cond) return null`. Ma un blocco di codice risulta essere molto più leggibile.

## Lunghezza della riga

Nessuno ama leggere lunghe righe orizzontali di codice. Una buona norma è quella di dividere le righe più lunghe in righe più brevi.

Ad esempio:

```
// i backtick consentono di dividere la stringa in più righe
let str = `
 ECMA International's TC39 is a group of JavaScript developers,
 implementers, academics, and more, collaborating with the community
 to maintain and evolve the definition of JavaScript.
`;
```

Invece, per le istruzioni `if`:

```
if (
 id === 123 &&
 moonPhase === 'Waning Gibbous' &&
 zodiacSign === 'Libra'
) {
 letTheSorceryBegin();
}
```

La lunghezza massima dovrebbe essere accordata a livello di team. Solitamente tra gli 80-120 caratteri.

## Indentazione

Ci sono due tipi di indentazione:

- **Indentazione orizzontale: 2 o 4 spazi.**

Un indentazione orizzontale è realizzata usando 2 o 4 spazi oppure il tasto “Tab”. Quale scegliere è una guerra che dura da anni. Ad oggi gli spazi sono i più comuni.

Un vantaggio degli spazi contro i tabs è che gli spazi permettono configurazioni più flessibili.

Ad esempio, possiamo allineare gli argomenti con l'apertura della parentesi, come nell'esempio:

```
show(parameters,
 aligned, // padding di 5 spazi a sinistra
 one,
 after,
 another
) {
 // ...
}
```

- **Indentazione verticale: righe vuote per dividere il codice in blocchi logici.**

Anche una singola funzione può essere divisa in più blocchi logici. Nell'esempio sotto, l'inizializzazione delle variabili, il corpo del ciclo e il ritorno del risultato sono divisi verticalmente:

```
function pow(x, n) {
 let result = 1;
 // <--
 for (let i = 0; i < n; i++) {
 result *= x;
 }
 // <--
 return result;
}
```

Inserire una nuova riga vuota aiuta a rendere il codice più leggibile. Non dovrebbero esserci più di nove righe di codice senza un indentazione verticale.

## Punto e virgola

Il punto e virgola dovrebbe essere presente alla fine di ogni istruzione, anche se non è obbligatorio.

Esistono linguaggi in cui il punto e virgola è realmente opzionale e può essere omesso. In JavaScript, esistono casi in cui un “a capo” non viene interpretato come un punto e virgola, lasciando quindi il codice vulnerabile agli errori.

Quando diventerete più maturi come programmatore, potrete scegliere lo stile senza punto e virgola [StandardJS ↗](#). Fino a quel momento la scelta migliore è quella di inserirlo alla fine di ogni istruzione per evitare errori.

## Livelli di annidamento

Nel codice vanno evitati elevati livelli di annidamento.

Qualche volta torna utile la direttiva “[continue](#)” per evitare annidamenti extra.

Ad esempio, invece che aggiungere un `if`:

```

for (let i = 0; i < 10; i++) {
 if (cond) {
 ... // <- un ulteriore livello di annidamento
 }
}

```

Possiamo scriverlo come:

```

for (let i = 0; i < 10; i++) {
 if (!cond) continue;
 ... // <- nessun ulteriore livello di annidamento
}

```

Una cosa simile può essere risolta con `if/else` e `return`.

Ad esempio, i due costrutti sotto sono identici.

Opzione 1:

```

function pow(x, n) {
 if (n < 0) {
 alert("Negative 'n' not supported");
 } else {
 let result = 1;

 for (let i = 0; i < n; i++) {
 result *= x;
 }

 return result;
 }
}

```

Opzione 2:

```

function pow(x, n) {
 if (n < 0) {
 alert("Negative 'n' not supported");
 return;
 }

 let result = 1;

 for (let i = 0; i < n; i++) {
 result *= x;
 }

 return result;
}

```

La seconda opzione è molto più leggibile perché il “caso limite” `n < 0` viene gestito in anticipo. Dopo aver eseguito il controllo possiamo proseguire con il flusso principale senza doverci preoccupare di ulteriori casi di annidamento.

## Posizionamento delle funzioni

Se state scrivendo molte funzioni “ausiliarie”, ci sono tre modi per organizzarle nel codice.

1. Dichiarare le funzioni sopra il codice che le utilizza:

```
// dichiarazione di funzioni
function createElement() {
 ...
}

function setHandler(elem) {
 ...
}

function walkAround() {
 ...
}

// codice che le utilizza
let elem = createElement();
setHandler(elem);
walkAround();
```

2. Prima il codice, poi le funzioni:

```
// il codice che utilizza le funzioni
let elem = createElement();
setHandler(elem);
walkAround();

// --- funzioni di supporto ---
function createElement() {
 ...
}

function setHandler(elem) {
 ...
}

function walkAround() {
 ...
}
```

3. Mix: una funzione viene dichiarata nel punto in cui viene utilizzata la prima volta.

Nella maggior parte dei casi si preferisce la seconda opzione.

Questo perché quando leggiamo il codice vogliamo prima di tutto sapere cosa fa. Mettendo prima il codice possiamo fornire queste informazioni. Successivamente, potrebbe non essere necessario leggere le funzioni, soprattutto se i loro nomi sono autodescrittivi.

## Style guide

Una style guide contiene regole generali riguardo a “come scrivere” il codice, ad esempio quali api utilizzare, di quanti spazi indentare, quando andare a capo, e molti altri dettagli.

Quando tutti i membri del team utilizzano la stessa guida, il codice tende ad essere più uniforme.

Certamente un team può utilizzare la propria style guide, ma spesso non è necessario definirne una propria. Ce ne sono molte già pronte, scegliere una tra queste è generalmente la scelta migliore.

Alcune delle scelte più popolari:

- [Google JavaScript Style Guide ↗](#)
- [Airbnb JavaScript Style Guide ↗](#)
- [Idiomatic.JS ↗](#)
- [StandardJS ↗](#)
- (e molte altre)

Se sei un nuovo sviluppatore, inizia con i consigli di questo capitolo. Quando avrai appreso bene lo stile potrai cercare quello che più ti appartiene.

## Linter

I linters sono strumenti che controllano automaticamente lo stile del codice e vi danno consigli su come sistemarlo.

La miglior cosa di questi strumenti è che il controllo dello stile in qualche occasione può rilevare dei bug, ad esempio degli errori di battitura nei nomi delle funzioni. Proprio per queste sue caratteristiche, installare un linter è fortemente consigliato anche se non avete intenzioni di rimanere fedeli ad uno particolare “stile di programmazione”.

Alcuni fra i linter più conosciuti:

- [JSLint ↗](#) – uno dei primi linter.
- [JSHint ↗](#) – molte più opzioni di JSLint.
- [ESLint ↗](#) – il più recente.

Tutti quelli elencati svolgono molto bene il lavoro. L'autore della guida utilizza [ESLint ↗](#).

Molti linter sono integrati negli editor più popolari: è sufficiente attivare il plugin e configurare lo stile desiderato.

Ad esempio, per poter utilizzare ESLint è sufficiente:

1. Installa [Node.js ↗](#).
2. Installa ESLint con il comando `npm install -g eslint` (`npm` è un package manager di JavaScript).
3. Crea un file di configurazione e chiamalo `.eslintrc` nella root del tuo progetto JavaScript (la cartella che contiene tutti i file).
4. Installa/abilita il plugin per il tuo editor per integrare ESLint. La maggior parte degli editor ne possiede uno.

Qui un esempio di di un file `.eslintrc`:

```
{
 "extends": "eslint:recommended",
 "env": {
 "browser": true,
 "node": true,
 "es6": true
 },
 "rules": {
 "no-console": 0,
 "indent": 2
 }
}
```

La direttiva `"extends"` indica che la configurazione è basata sulla lista dei setting `"eslint:recommended"`. Dopodiché potremo specificare il nostro stile personale.

E' anche possibile scaricare un set di regole dal web ed estenderle a nostro piacimento. Vedi <http://eslint.org/docs/user-guide/getting-started> per maggiori dettagli riguardo l'installazione.

Molti IDE hanno dei linter integrati, che sono comodi ma non sono editabili come ESLint.

## Riepilogo

Tutte le regole sintattiche descritte in questo capitolo (e nei riferimenti delle style guides) aiutano ad incrementare la leggibilità del codice, ma sono tutte contestabili.

Quando stiamo pensando a come scrivere codice “migliore”, la domanda dovrebbe essere “Cosa rende il codice più leggibile e facile da capire?” e “Cosa può aiutarmi ad evitare gli errori?”. Queste sono le principali cose da tenere a mente quando stiamo cercando di scegliere una style guide.

Conoscere gli stili più popolari vi consentirà di tenervi aggiornati con le ultime idee riguardo gli stili di programmazione e le best practices.

## ✓ Esercizi

### Pessimo stile

importanza: 4

Cosa c'è di sbagliato con lo stile di programmazione qui sotto?

```
function pow(x,n)
{
 let result=1;
 for(let i=0;i<n;i++) {result*=x;}
 return result;
}

let x=prompt("x?", ''),
n=prompt("n?", '')
if (n<=0)
{
 alert(`Power ${n} is not supported, please enter an integer number greater than zero`);
}
else
```

```
{
 alert(pow(x,n))
}
```

Sistematelo.

[Alla soluzione](#)

## Commenti

Come abbiamo già appreso dal capitolo [Struttura del codice](#), i commenti possono essere di una singola riga: ed iniziano con `//` oppure multilinea: `/* ... */`.

Normalmente li utilizziamo per descrivere come e perché funziona il codice.

Inizialmente, l'inserimento di commenti può sembrare ovvio, ma i novizi solitamente non lo fanno nella maniera corretta.

## Commenti sbagliati

I novizi tendono ad utilizzare i commenti per spiegare “cosa sta succedendo nel codice”. Come in questo esempio:

```
// Questo codice farà questa cosa (...) e questa (...)
// ...e altre cose...
very;
complex;
code;
```

Nel buon codice la quantità di commenti “esplicativi” dovrebbe essere minima. In realtà il codice dovrebbe essere facile da comprendere anche senza.

C’è una bellissima citazione a riguardo: “Se il codice è così poco chiaro da richiedere un commento, probabilmente dovrebbe essere riscritto”.

## Ricetta: raccogliere in una funzione

Qualche volta può essere un beneficio rimpiazzare un pezzo di codice con una funzione, come in questo esempio:

```
function showPrimes(n) {
 nextPrime:
 for (let i = 2; i < n; i++) {

 // controlla se i è un numero primo
 for (let j = 2; j < i; j++) {
 if (i % j == 0) continue nextPrime;
 }

 alert(i);
 }
}
```

La migliore variante, sarebbe raccogliere il tutto in una funzione `isPrime`:

```
function showPrimes(n) {
 for (let i = 2; i < n; i++) {
 if (!isPrime(i)) continue;

 alert(i);
 }
}

function isPrime(n) {
 for (let i = 2; i < n; i++) {
 if (n % i == 0) return false;
 }

 return true;
}
```

Ora possiamo capire il codice più facilmente. La funzione stessa diventa un commento. Questo tipo di codice viene definito *auto-descrittivo*.

## Ricetta: creare funzioni

Anche se avete un lungo “pezzo di codice” come questo:

```
// qui aggiungiamo whiskey
for(let i = 0; i < 10; i++) {
 let drop = getWhiskey();
 smell(drop);
 add(drop, glass);
}

// qui aggiungiamo della spremuta
for(let t = 0; t < 3; t++) {
 let tomato = getTomato();
 examine(tomato);
 let juice = press(tomato);
 add(juice, glass);
}

// ...
```

Potrebbe essere una buona idea raccogliere tutto in una funzione:

```
addWhiskey(glass);
addJuice(glass);

function addWhiskey(container) {
 for(let i = 0; i < 10; i++) {
 let drop = getWhiskey();
 //...
 }
}

function addJuice(container) {
```

```
for(let t = 0; t < 3; t++) {
 let tomato = getTomato();
 //...
}
}
```

Ripeto nuovamente, le funzioni stesse dovrebbero dire cosa sta succedendo. Non dovrebbe esserci alcun bisogno di commenti. Inoltre anche l'architettura del codice è migliore quando è spezzata. Rende più chiaro lo scopo di ogni funzione.

Nella pratica, non possiamo evitare del tutto i commenti “esplicativi”. Ci sono algoritmi molto complessi. E ci sono vari “trucchi” con lo scopo di ottimizzare questo tipo di commenti. In linea di massima dovremmo però cercare di tenere il codice semplice ed auto-descrittivo.

## Buoni commenti

Quindi, solitamente i commenti esplicativi sono sbagliati. Quali sono allora i commenti giusti?

### Descrivere l'architettura

Fornire un visuale di alto livello dei componenti, come interagiscono, come si comporta il flusso d'esecuzione in certe situazioni... In breve – gli “occhi d'aquila” del codice. Esiste uno speciale linguaggio di schematizzazione, [UML](#) per la descrizione dell'architettura ad alto livello. Da studiare assolutamente.

### Documentare l'utilizzo di una funzione

Esiste una particolare sintassi [JSDoc](#) per documentare le funzioni: utilizzo, parametri, valori di ritorno.

Ad esempio:

```
/**
 * Ritorna la potenza n di x.
 *
 * @param {number} x La base della potenza.
 * @param {number} n Esponente, deve essere un numero naturale.
 * @return {number} x elevato alla n.
 */
function pow(x, n) {
 ...
}
```

Questi commenti ci consentono di capire lo scopo della funzione e come utilizzarla correttamente senza guardarne il codice.

I molti casi, gli editor come [WebStorm](#) sono in grado di comprenderli e possono quindi utilizzarli per autocompletamenti e alcune verifiche automatiche del codice.

Ci sono anche tool come [JSDoc 3](#) che possono generare documentazione in HTML a partire dai commenti. Puoi scoprire di più riguardo JSDoc su <http://usejsdoc.org/>.

## Perché l'azione viene risolta in quel modo?

Quello che viene scritto è fondamentale. Ma quello che *non* viene scritto potrebbe esserlo anche di più per capire cosa sta succedendo. Perché l'azione viene portata a termine in quel modo? Il codice non fornisce risposte.

Se ci sono diverse modalità di risolvere una determinata azione, perché si è scelta questa? Specialmente quando non risulta la scelta più ovvia.

Senza dei commenti si potrebbero generare le seguenti situazioni:

1. Tu (o un tuo collega) apri il codice un po' di tempo dopo, lo guardi e pensi che il codice è "poco ottimizzato".
2. Tu stesso potresti pensare: "Quanto stupido sono stato qui, e quanto intelligente sono adesso", e riscriverla utilizzando la variante "più ovvia e corretta".
3. ...Lo stimolo di riscriverla sarebbe forte. Ma quando l'hai scritta ti eri reso conto che la soluzione "più ovvia" era effettivamente peggiore. Andando a rileggerla potresti non ricordarti neanche perché. Quindi dopo averla riscritta ti rendi conto che è meglio tornare indietro, hai sprecato tempo.

Commenti che spiegano la soluzione sono fondamentali. Vi aiutano a sviluppare mantenendo sempre la strada corretta.

### **Ci sono alcune piccolezze? Dove vengono utilizzate?**

Se il codice contiene sottigliezze contro intuitive, vale certamente la pena commentarle.

## **Riepilogo**

Un importante qualità che deve possedere un bravo sviluppatore, è quella di saper scrivere dei buoni commenti.

I buoni commenti ci consentono di mantenere il codice in uno stato ottimale, e di poterci tornare dopo un po' di tempo e capire le scelte prese.

### **Commenti utili:**

- Architettura complessiva, vista ad alto livello.
- Utilizzo delle funzioni.
- Soluzioni importanti, soprattutto quando poco ovvie.

### **Commenti da evitare:**

- Quelli che dicono "come il codice funziona" e "cosa fa".
- Inseriteli solo se risulta impossibile rendere il codice semplice ed auto-descrittivo.

I commenti vengono utilizzati anche da strumenti che generano documentazione, come JSDoc3: li leggono e generano documenti HTML (o in altri formati).

## **Codice ninja**

*Imparare senza pensare è lavoro sprecato;  
pensare senza imparare è pericoloso.*

“ Confucio

I programmatore ninja in passato hanno usato queste tecniche per formare le menti dei manutentori del codice.

I guru della revisione dei codici le utilizzano per le attività di test.

I programmatore meno esperti le utilizzano anche meglio dei programmatore ninja.

Leggetele attentamente e cercate di capire a quale categoria appartenete – ninja, novelli, o forse revisionisti di codice?

### Allarme ironia

Molti hanno provato a seguire il percorso dei ninja. Pochi ci sono riusciti.

## La brevità è l'anima dell'intelligenza

Rendete il codice più breve possibile. Mostrerà quanto intelligenti siete.

Lasciate che le sottigliezze del linguaggio vi guidino.

Ad esempio, date un'occhiata all'operatore ternario `'?' :`

```
// preso da una libreria javascript famosa
i = i ? i < 0 ? Math.max(0, len + i) : i : 0;
```

Bello vero? Se scrivete cose del genere, uno sviluppatore che prova a leggere queste righe perderà molto tempo nel capire cosa rappresenta il valore `i`. Alla fine verrebbe da voi per cercare una risposta.

Insegnategli che la brevità è sempre la cosa migliore. Iniziatelo al percorso dei ninja.

## Variabili ad una lettera

*Il Dao si nasconde nel silenzio. Solo il Dao è completo.*



Laozi (Tao Te Ching)

Un altro metodo per programmare velocemente è utilizzare nomi di variabili a singolo carattere ovunque. Come `a`, `b` o `c`.

Una variabile molto breve sparirà nel codice come un vero ninja scompare nella foresta. Nessuno sarà in grado di ritrovarla, nemmeno utilizzando il tasto “cerca”. E se qualcuno mai riuscirà a trovarla non “decifrerà” mai il vero significato di `a` o `b`.

...C'è però un'eccezione. Un vero ninja non utilizzerà mai `i` come contatore in un ciclo `"for"`. Ovunque, ma non qui. Si guarderà intorno, ci sono moltissime lettere esotiche. Ad esempio `x` o `y`.

Utilizzare una variabile esotica come contatore di un ciclo è ancora meglio se il corpo è grande 1-2 pagine (o anche di più). Quindi se qualcuno guarderà dentro, nella profondità del ciclo, non sarà in grado di capire che `x` è il contatore del ciclo.

## Variabili abbreviate

Se il regolamento del team vieta di utilizzare nomi vaghi o composti di una sola lettera – allora accorciateli, inventate abbreviazioni.

Come questi:

- `list` → `lst`.
- `userAgent` → `ua`.
- `browser` → `brsr`.
- ...etc

Solo una persona con un'ottima intuizione sarà in grado di capire questi nomi. Provate ad abbreviare tutto. Solo una persona degna sarà capace di sostenere il vostro codice.

## Libratevi in alto. Siate astratti.

*I quadrati migliori non hanno spigoli  
Le migliori navi sono le ultime complete,  
I grandi commenti raramente hanno un suono,  
Le migliori immagini non hanno forma.*

“ Laozi (Tao Te Ching)

Quando scegliete un nome cercate di utilizzare parole più astratte possibili. Come `obj`, `data`, `value`, `item`, `elem` e molte altre.

- **Il nome ideale per una variabile è `data`.** Usatela il più possibile. Infondo, tutte le variabili contengono *dati* giusto?

...Cosa fare se `data` è già stato utilizzato? Provate `value`, anche questo è un nome universale. Dopo tutto una variabile potrebbe avere un *valore*.

- **Denominate le variabili in base al loro tipo:** `str`, `num` ...

Dategli una possibilità. Un giovane iniziato potrebbe pensare – sono veramente utili ad un ninja questi nomi? Infatti lo sono!

Certamente il nome della variabile contiene comunque un significato. Infatti informa riguardo cosa è contenuto nella variabile: una stringa, un numero o qualcos'altro. Ma quando un estraneo cercherà di capire il codice, rimarrà sorpreso scoprendo che in realtà non forniscono alcuna informazione! Quindi fallirà nel suo intento di modificare il vostro codice.

Il tipo del valore è semplice da trovare con un debugger. Ma qual'è il significato della variabile? Quale numero/stringa contiene?

Non c'è alcun modo per scoprirla senza un pò di meditazione!

- **...Cosa fare se non sono più disponibili questi nomi?** Aggiungete semplicemente un numero: `data1`, `item2`, `elem5` ...

## Test di attenzione

Solo un vero programmatore dovrebbe essere in grado di capire il vostro codice. Ma come essere sicuri di questo?

**Una delle possibili strade – usare nomi di variabili simili, come `date` e `data`.**

Mixateli anche insieme quando possibile.

Una rapida lettura di un codice del genere diventerà impossibile. E quando ci sarà un errore di battitura... Ummm... Avremo un pò di tempo per berci un buon tea.

## Sinonimi intelligenti

*La cosa più difficile al mondo è trovare un gatto nero in una stanza buia, specialmente se il gatto non c'è.*

“ Confucio

Utilizzare nomi *simili* per la stessa cosa rende la vita più interessante e mostra al pubblico la tua creatività.

Ad esempio, consideriamo i prefissi delle funzioni. Se una funzione mostra un messaggio sullo schermo – potremmo iniziarla con `display...`, ad esempio `displayMessage`. Se poi un'altra funzione mostra sullo schermo qualcosa altro, come uno username, potremmo iniziare con `show...` (ad esempio `showName`).

Facendo credere che ci sia una piccola differenza tra le due funzioni, quando invece non c'è.

Mettetevi d'accordo con gli altri ninja del team: se John inizia le funzioni di "showing" con `display...`, allora Peter potrebbe usare `render...`, e Ann – `paint...`. Il codice diventerebbe così molto misto e interessante.

...E ora il colpo finale!

Per due funzioni con differenze fondamentali – utilizzate lo stesso prefisso!

Ad esempio la funzione `printPage(page)` che utilizzerà la stampante. E la funzione `printText(text)` che mostrerà un messaggio sullo schermo. Così un estraneo dovrà pensare bene ai nomi delle funzioni `printMessage`: "Dove verrà messo il messaggio? Nella stampante o sullo schermo? ". Per rendere il tutto ancora più bello `printMessage(message)` dovrebbe mostrare il messaggio in una nuova pagina!

## Riciclo dei nomi

*Quando l'intero è diviso, le parti necessitano di nomi.*

“ Laozi (Tao Te Ching)

*Ci sono già abbastanza nomi.*

*Uno dovrebbe sapere quando fermarsi.*

Aggiungere una nuova variabile solo quando è assolutamente necessaria.

Piuttosto utilizzate quelle già presenti. Semplicemente riscrivetele.

In una funzione conviene utilizzare solo variabili passate come parametri.

Questo renderà davvero difficile capire cosa c'è esattamente all'interno della variabile *in questo momento*. E anche da dove proviene. Una persona con una pessima intuizione sarà costretta ad analizzare il codice linea per linea e tracciare tutti i possibili flussi.

**Una variante avanzata è quella di sostituire di nascosto (!) i valori con qualcos'altro di simile nel bel mezzo di una ciclo o di una funzione.**

Ad esempio:

```
function ninjaFunction(elem) {
 // 20 righe di codice che lavorano su elem

 elem = clone(elem);

 // altre 20 righe, che lavorano sul clone di elem!
}
```

Un programmatore esterno che vorrebbe provare ad interagire con `elem` nella seconda parte della funzione, rimarrà sorpreso... Solamente in fase di debugging, dopo aver esaminato attentamente il codice si renderà conto che stava lavorando con un clone!

Se ripetuto regolarmente nel codice, diventa letale anche contro i ninja più esperti.

## Underscore per divertimento

Inserite underscore `_` e `__` prima dei nomi delle variabili. Come `_name` o `__value`. Sarebbe bello sapere che solo tu ne conosci il significato. O meglio, aggiungeteli solo per divertimento, senza scopo. Oppure con differenti significati in posti diversi.

Prendereste due piccioni con una fava. Primo, il codice diventerebbe più lungo e meno leggibile, e secondo, uno sviluppatore poco scaltro ci perderà molto tempo prima di riuscire a metter le mani sul vostro codice.

Un ninja intelligente mette gli underscore in un solo punto del codice e li evita in altri. Questo aumenta la fragilità del codice e aumenta la possibilità di commettere errori.

## Mostrate il vostro amore

Mostrate a tutti quando siano belli i vostri elementi! Nomi come `superElement`, `megaFrame` e `niceItem` sicuramente cattureranno l'animo del lettore.

Infatti, utilizzerete dei bei nomi come: `super..`, `mega..`, `nice..` Ma in ogni caso non fornirete dettagli. Un lettore esterno potrebbe ragionare sui possibili significato perdendo una o due ore.

## Sovrascrittura delle variabili esterne

*Quando sei nella luce, non puoi vedere niente nell'oscurità.*

“ Guan Yin Zi

*Quando sei nell'oscurità, puoi vedere tutto nella luce.*

Usare gli stessi nomi per le variabili interne ed esterne. E' semplice. Non sono richiesti ulteriori sforzi.

```
let user = authenticateUser();

function render() {
 let user = anotherValue();
 ...
 ...many lines...
 ...
 ... // <-- un programmatore vorrebbe utilizzare user...
 ...
}
```

Un programmatore che si trova dentro `render` probabilmente non si accorgerà che la variabile locale `user` sta nascondendo quella esterna.

Quindi potrebbe provare a lavorare con `user` pensando erroneamente che sia quella esterna, quella con il risultato di `authenticateUser()`... La trappola è servita! Addio debugger...

## Side-effect ovunque!

Ci sono funzioni che sembrano non avere effetti. Come `isReady()`, `checkPermission()`, `findTags()`... Si presume che eseguano semplici calcoli, trovino e ritornino il dato, senza cambiare nulla all'esterno. In altre parole senza "side-effect".

**Un bellissima sorpresa potrebbe essere quella di aggiungere un azione “utile”, oltre a quella principale.**

Un'espressione di stupore apparirà nel volto del vostro collega quando scoprirà che una funzione chiamata `is...`, `check...` o `find...` cambia qualcosa – allargarrete di molto i suoi confini di comprensione.

**Un altro modo per stupire è ritornare risultati non standard.**

Mostrate il vostro pensiero originale! Alla chiamata di `checkPermission`, invece che ritornare `true/false`, ritornate un oggetto complesso contenente il risultato del check.

Quindi degli sviluppatori che proveranno a scrivere `if (checkPermission(..))`, si chiederanno perché non funziona. Rispondete severamente: "Leggete la documentazione".

## Funzioni potenti!

*Il grande Tao fluisce ovunque,  
sia a sinistra che a destra.*

“ Laozi (Tao Te Ching)

Non limitate le funzioni a ciò che il loro nome esprime. Siate elastici.

Ad esempio, una funzione `validateEmail(email)` dovrebbe (dopo averne controllato la correttezza) mostrare un errore e chiedervi di reinserire la mail.

Azioni supplementari non dovrebbero risultare ovvie dal nome della funzione. Un vero programmatore ninja cerca di renderle il meno ovvie possibile.

### **Unire più azioni in una protegge il vostro codice dal riutilizzo.**

Immaginate, un altro sviluppatore vuole solo verificare le email, e non mostrare alcun messaggio. La vostra funzione `validateEmail(email)` che svolge entrambi i compiti non fa al caso suo. Quindi non dovrà infastidirvi facendo domande, visto che non utilizzerà il vostro codice.

## **Riepilogo**

Tutti i “consigli” forniti sopra provengono da codici reali... Qualche volta scritti da programmatori con esperienza. Forse con più esperienza di voi ;)

- Seguite alcuni di questi consigli, e il vostro codice sarà pieno di sorprese.
- Seguitene molti, e il vostro codice diventerà veramente vostro, nessuno vorrà utilizzarlo.
- Seguitele tutte, ed il vostro codice sarà una guida spirituale per i giovani sviluppatori in cerca della luce.

## **Test automatici con Mocha**

Il test automatico sarà utilizzato per molte attività.

Fa parte della “preparazione minima” di uno sviluppatore.

### **Perché sono necessari i test?**

Quando scriviamo una funzione, spesso possiamo immaginare quello che deve fare: i parametri necessari e i risultati restituiti.

Durante lo sviluppo, possiamo controllare le funzioni eseguendole e controllando se i risultati sono quelli aspettati. Ad esempio possiamo farlo tramite la console.

Se qualcosa non funziona – allora possiamo sistemare il codice, rieseguirlo e controllare nuovamente il risultato – e continuare a ripetere questa procedura fino a risolvere il bug.

Ma alcuni test manuali non sono sempre perfetti.

### **Quando testiamo il codice manualmente rieseguendolo, è facile dimenticare qualcosa.**

Ad esempio, stiamo creando una funzione `f`. Scriviamo del codice, lo testiamo con: `f(1)` e funziona, ma con `f(2)` non funziona. Sistemiamo il codice e ora `f(2)` funziona. Il test sembra completo? Invece ci siamo dimenticati di ri-testare `f(1)`. Questo infatti potrebbe contenere un errore.

Questo è un errore tipico. Quando sviluppiamo qualcosa, cerchiamo di tenere a mente molti possibili casi di utilizzo. Ma è difficile aspettarsi che un programmatore controlli a mano il risultato dopo ogni cambiamento. Diventa quindi facile sistemare una bug e crearne uno di nuovo.

**Test automatici significa che i test vengono scritti separati, e sono complementari al codice. Possono essere facilmente eseguiti ed utilizzati per controllare i principali casi di utilizzo.**

## Behavior Driven Development (BDD)

Utilizziamo una tecnica chiamata [Behavior Driven Development](#) o, in breve, BDD. Questo approccio viene utilizzato in moltissimi progetti. BDD non offre solo testing, ha molte altre funzionalità.

**BDD contiene tre cose in una: test, documentazione ed esempi.**

Abbiamo parlato abbastanza. Vediamo degli esempi.

### Sviluppo di “pow”: le specifiche

Vogliamo creare una funzione `pow(x, n)` che calcola la potenza di `x` per un intero `n`. Assumiamo che `n≥0`.

Questo è solo un esempio: infatti l'operatore `**` svolge quest'azione, ma concentriamoci sul flusso di sviluppo, che potremmo poi applicare a funzioni più complesse.

Prima di scrivere il codice di `pow`, possiamo immaginare cosa vogliamo che la funzione faccia e descriverlo.

Questa descrizione viene chiamata *specifica* o, in breve, *spec*, ed appare così:

```
describe("pow", function() {
 it("raises to n-th power", function() {
 assert.equal(pow(2, 3), 8);
 });
});
```

Una spec ha tre principali blocchi:

```
describe("title", function() { ... })
```

Viene descritta la funzionalità. Utilizzata per raggruppare le “attività” – i blocchi `it`. Nel nostro caso descriviamo la funzione `pow`.

```
it("title", function() { ... })
```

Nel titolo di `it` descriviamo il particolare caso d'uso *leggibile per gli umani*, come secondo argomento una funzione che lo testa.

```
assert.equal(value1, value2)
```

Il codice all'interno del blocco `it`, se l'implementazione è corretta, dovrebbe eseguire senza errori.

Le funzioni `assert.*` vengono utilizzate per controllare che `pow` funzioni come dovrebbe. Proprio qui ne utilizziamo una – `assert.equal`, che confronta gli argomenti e ritorna un errore se questi non sono uguali. Qui verifichiamo che il risultato di `pow(2, 3)` sia uguale `8`.

Ci sono molti altri tipi di confronto e controllo che vederemo più avanti.

### Il flusso di sviluppo

Il flusso di sviluppo solitamente segue i passi:

1. Viene scritta una spec iniziale, con dei test per le funzionalità di base.
2. Si crea un implementazione di base.
3. Per verificare che questa funzioni, utilizziamo un framework di testing come [Mocha ↗](#) (presto maggiori dettagli) che esegue le spec. Vengono mostrati gli errori. Facciamo le correzioni e riproviamo finché tutto funziona correttamente.
4. Ora abbiamo un'implementazione iniziale che funziona bene con i test.
5. Aggiungiamo più casi d'uso alla spec, magari ancora non supportate dall'implementazione. Così i test inizieranno a fallire.
6. Quindi tornate al passo 3, aggiornate l'implementazione e continuate finché tutto non funziona correttamente.
7. Ripetete gli step 3-6 fino ad ottenere la funzionalità desiderata.

Quindi la fase di sviluppo è *iterativa*. Scriviamo la specifica, la implementiamo, ci accertiamo che passi i test, ci assicuriamo che faccia ciò che deve. Al termine di questa procedura avremmo un'implementazione già testata e funzionante.

Nel nostro caso, il primo step è completo: abbiamo una specifica iniziale di `pow`. Quindi ora passiamo all'implementazione. Come prima cosa facciamo l'esecuzione “zero” con le specifiche scritte, per essere sicuri che tutto funzioni (ovviamente i test dovrebbero fallire tutti).

## La spec in azione

In questo guida utilizzeremo le seguenti librerie JavaScript per fare test:

- [Mocha ↗](#) – un core framework: fornisce le maggiori funzioni di test come `describe` e `it` e le principali funzioni che eseguono i test.
- [Chai ↗](#) – una libreria con molte asserzioni. Ci consente di utilizzare molte asserzioni differenti, per ora ci servirà solamente `assert.equal`.
- [Sinon ↗](#) – una libreria per il controllo oltre le funzioni, emula funzioni integrate e molto altro, la utilizzeremo più avanti.

Queste librerie sono utili sia per il test browser, sia per il test lato server. Qui considereremo la variante browser.

La pagina HTML con questi framework e le spec di `pow`:

```
<!DOCTYPE html>
<html>
<head>
 <!-- add mocha css, to show results -->
 <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.css">
 <!-- add mocha framework code -->
 <script src="https://cdnjs.cloudflare.com/ajax/libs/mocha/3.2.0/mocha.js"></script>
 <script>
 mocha.setup('bdd'); // minimal setup
 </script>
 <!-- add chai -->
 <script src="https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js"></script>
 <script>
 // chai has a lot of stuff, let's make assert global
```

```

let assert = chai.assert;
</script>
</head>

<body>

<script>
 function pow(x, n) {
 /* function code is to be written, empty now */
 }
</script>

<!-- the script with tests (describe, it...) -->
<script src="test.js"></script>

<!-- the element with id="mocha" will contain test results -->
<div id="mocha"></div>

<!-- run tests! -->
<script>
 mocha.run();
</script>
</body>

</html>

```

La pagina può essere suddivisa in cinque parti:

1. `<head>` – aggiunge librerie di terze parti e un po di stile utile per i test.
2. `<script>` con la funzione da testare, nel nostro caso – con il codice di `pow`.
3. i test – nel nostro caso uno script esterno `test.js` che contiene `describe("pow", ...)` visti sopra.
4. L'elemento HTML `<div id="mocha">` verrà utilizzato da Mocha per mostrare i risultati.
5. Il test viene iniziato dal comando `mocha.run()`.

Il risultato:

The screenshot shows a browser window displaying the results of a Mocha test. The title of the page is "pow". Below the title, there is a red error message: "AssertionError: expected undefined to equal 8 at Context.<anonymous> (test.js:4:12)". At the top right of the browser window, there is a status bar with the text "passes: 0 failures: 1 duration: 0.02s 100%".

Per ora, i test falliscono, ci sono quindi errori. Questo è ovvio: abbiamo una funzione `pow` vuota, quindi `pow(2, 3)` ritorna `undefined` invece di `8`.

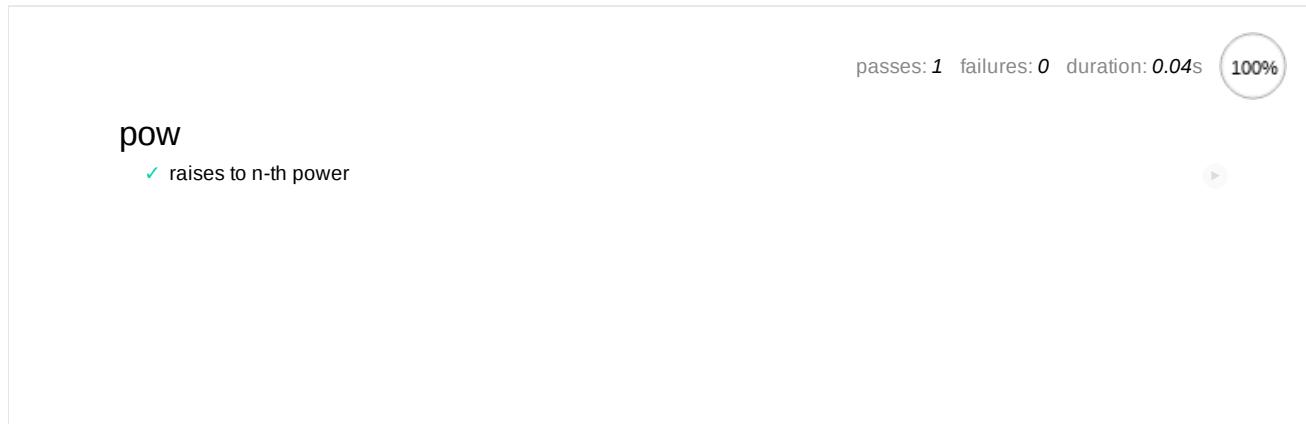
Per il futuro, vi faccio notare che ci sono dei test più avanzati, come [karma ↗](#) e altri. Quindi solitamente non ci saranno problemi a impostare differenti test.

## Implementazione iniziale

Proviamo a fornire una semplice implementazione di `pow`, per passare il test:

```
function pow(x, n) {
 return 8; // :) abbiamo barato!
}
```

Wow, funziona!



## Miglioriamo le spec

Quello che abbiamo fatto finora è barare. La funzione non “funziona”: un tentativo di calcolare `pow(3, 4)` fornirebbe un risultato scorretto, ma il test risulta comunque passato.

... Questa situazione è fra le più tipiche, nella pratica succede molto spesso. I test vengono passati ma le funzioni non lavorano correttamente. La nostra spec è imperfetta. Abbiamo bisogno di introdurre un numero maggiore di casi d’uso.

Aggiungiamo un altro test per verificare se `pow(3, 4) = 81`.

Possiamo selezionare uno dei due metodi per organizzare i test:

1. La prima variante – aggiunger un ulteriore `assert` all’interno dello stesso `it`:

```
describe("pow", function() {
 it("raises to n-th power", function() {
 assert.equal(pow(2, 3), 8);
 assert.equal(pow(3, 4), 81);
 });
});
```

2. La seconda – scrivere due test separati:

```
describe("pow", function() {
 it("2 raised to power 3 is 8", function() {
 assert.equal(pow(2, 3), 8);
 });
 it("3 raised to power 4 is 81", function() {
```

```
 assert.equal(pow(3, 4), 81);
 });

});
```

La principale differenza è che quando `assert` trova un errore, `it` si blocca e il test viene terminato. Quindi nella prima variante se il primo `assert` fallisce, allora non potremo vedere il risultato del secondo `assert`.

Scrivere test separati è utile per ottenere maggiori informazioni riguardo ciò che sta succedendo, quindi la seconda variante è la migliore.

Ed oltre a questo ci sono altre regole che sono utili da seguire.

### Un test controlla una sola cosa.

Se guardiamo al codice di un test e vediamo che controlla due cose differenti, è meglio dividerlo in due test più semplici.

Quindi continuiamo con l'idea che la seconda variante risulta essere la migliore.

Il risultato:

The screenshot shows a test runner interface. At the top right, it displays "passes: 1 failures: 1 duration: 0.04s 100%". Below this, there's a section titled "pow" with two entries: a green checkmark followed by "2 raised to power 3 is 8" and a red X followed by "3 raised to power 4 is 81". A red error message box is overlaid on the screen, containing the text "AssertionError: expected 8 to equal 81 at Context.<anonymous> (test.js:8:12)".

Proprio come ci aspettavamo, il secondo test è fallito. Ovvio, la nostra funzione ritorna sempre `8`, mentre l'`assert` si aspetta `81`.

## Migliorare l'implementazione

Proviamo a scrivere qualcosa di più sensato per passare i test:

```
function pow(x, n) {
 let result = 1;

 for (let i = 0; i < n; i++) {
 result *= x;
 }

 return result;
}
```

Per essere sicuri che la funzione svolga il suo lavoro correttamente, vanno testati più valori. Piuttosto che scrivere i blocchi `it` manualmente, possiamo generarli con un ciclo `for`:

```

describe("pow", function() {

 function makeTest(x) {
 let expected = x * x * x;
 it(`#${x} in the power 3 is ${expected}`, function() {
 assert.equal(pow(x, 3), expected);
 });
 }

 for (let x = 1; x <= 5; x++) {
 makeTest(x);
 }
});

```

Il risultato:

```

passes: 5 failures: 0 duration: 0.04s 100%
pow
✓ 1 in the power 3 is 1
✓ 2 in the power 3 is 8
✓ 3 in the power 3 is 27
✓ 4 in the power 3 is 64
✓ 5 in the power 3 is 125

```

## Describe annidati

Adesso aggiungeremo ulteriori test. Prima di tutto ci rendiamo conto che la funzione di supporto `makeTest` e il ciclo `for` dovrebbero essere raggruppati. Non abbiamo bisogno di una funzione `makeTest` negli altri test, è utile solo nel `for`: il loro scopo è solo di controllare come `pow` si comporta al crescere delle potenze.

Il raggruppamento viene effettuato con un `describe` annidato:

```

describe("pow", function() {

 describe("raises x to power 3", function() {

 function makeTest(x) {
 let expected = x * x * x;
 it(`#${x} in the power 3 is ${expected}`, function() {
 assert.equal(pow(x, 3), expected);
 });
 }

 for (let x = 1; x <= 5; x++) {
 makeTest(x);
 }
 });
});

```

```
// ... altri test
});
```

Il `describe` annidato definisce un nuovo “sotto-gruppo” di test. Nell’output potremmo vedere l’indentazione:

```
passes: 5 failures: 0 duration: 0.04s 100%
pow
 raises x to power 3
 ✓ 1 in the power 3 is 1
 ✓ 2 in the power 3 is 8
 ✓ 3 in the power 3 is 27
 ✓ 4 in the power 3 is 64
 ✓ 5 in the power 3 is 125
```

In futuro potremmo aggiungere più `it` e `describe` allo stesso livello, ognuno di questi avrà le proprie funzioni di supporto ma non potranno vedere `makeTest`.

### **i** before/after and beforeEach/afterEach

Possiamo impostare le funzione `before/after` (prima/dopo) che vengono eseguite prima/dopo i test, o addirittura le funzioni `beforeEach/afterEach` (prima di ogni/dopo di ogni) che verranno eseguite prima di ogni `it`.

Ad esempio:

```
describe("test", function() {

 before(() => alert("Testing started - before all tests"));
 after(() => alert("Testing finished - after all tests"));

 beforeEach(() => alert("Before a test - enter a test"));
 afterEach(() => alert("After a test - exit a test"));

 it('test 1', () => alert(1));
 it('test 2', () => alert(2));

});
```

La sequenza d'esecuzione sarà:

```
Testing started - before all tests (before)
Before a test - enter a test (beforeEach)
1
After a test - exit a test (afterEach)
Before a test - enter a test (beforeEach)
2
After a test - exit a test (afterEach)
Testing finished - after all tests (after)
```

[Open the example in the sandbox.](#) ↗

Solitamente, `beforeEach/afterEach` (`before/each`) vengono utilizzati per eseguire inizializzazioni, azzerare i contatori o fare qualcosa prima di iniziare il prossimo test.

## Estendere le spec

La funzionalità di base di `pow` è completa. La prima iterazione di sviluppo è fatta. Dopo aver festeggiato e bevuto champagne – andiamo avanti provando ad aggiungere funzionalità.

Come abbiamo detto, la funzione `pow(x, n)` è stata sviluppata per funzionare con interi positivi `n`.

Per indicare un errore matematico, JavaScript solitamente ritorna `Nan`. Facciamo lo stesso per valori non validi di `n`.

Come prima cosa aggiungiamo il nuovo comportamento alle `spec(!)`:

```
describe("pow", function() {

 // ...
```

```

it("for negative n the result is NaN", function() {
 assert.isNaN(pow(2, -1));
});

it("for non-integer n the result is NaN", function() {
 assert.isNaN(pow(2, 1.5));
});

});

```

Il risultato con il nuovo test sarà:

```

pow
✖ if n is negative, the result is NaN
 AssertionError: expected 1 to be NaN
 at Function.assert.isNaN (https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js:11:1)
 at Context.<anonymous> (test.js:19:12)

✖ if n is not integer, the result is NaN
 AssertionError: expected 4 to be NaN
 at Function.assert.isNaN (https://cdnjs.cloudflare.com/ajax/libs/chai/3.5.0/chai.js:11:1)
 at Context.<anonymous> (test.js:23:12)

raises x to power 3
✓ 1 in the power 3 is 1
✓ 2 in the power 3 is 8
✓ 3 in the power 3 is 27
✓ 4 in the power 3 is 64
✓ 5 in the power 3 is 125

```

I nuovi test aggiunti falliranno, perché la nostra implementazione non li supporta ancora. Così è come funziona BDD: prima si scrivono test in modo che falliscano, e successivamente si lavora sull'implementazione.

## Altre asserzioni

Metto in evidenza l'asserzione `assert.isNaN`: che effettua controlli di tipo `Nan`.

In Chai sono presenti molte altre asserzioni, ad esempio:

- `assert.equal(value1, value2)` – controlla l'uguaglianza `value1 == value2`.
- `assert.strictEqual(value1, value2)` – verifica l'uguaglianza stretta `value1 === value2`.
- `assert.notEqual`, `assert.notStrictEqual` – controllo inverso a quello dell'uguaglianza stretta.
- `assert.isTrue(value)` – esegue il controllo `value === true`
- `assert.isFalse(value)` – verifica che `value === false`
- ...l'intera lista è disponibile nella [documentazione ↗](#)

Dovremmo quindi aggiungere un paio di linee a `pow`:

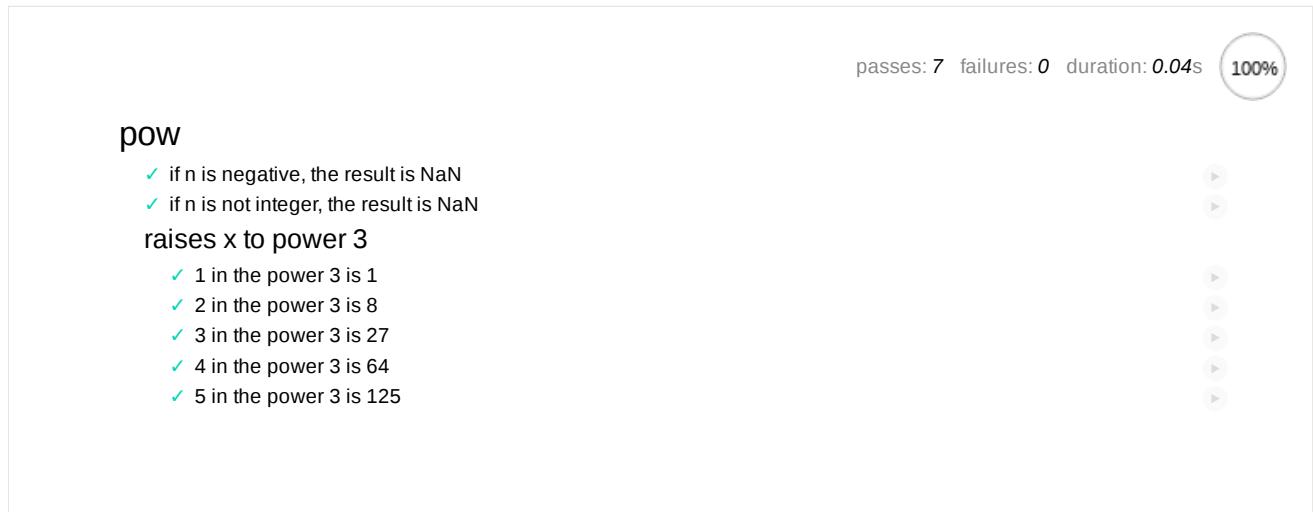
```
function pow(x, n) {
 if (n < 0) return NaN;
 if (Math.round(n) != n) return NaN;

 let result = 1;

 for (let i = 0; i < n; i++) {
 result *= x;
 }

 return result;
}
```

Ora funziona, tutti i test vengono passati:



The screenshot shows a test runner interface. At the top right, it displays "passes: 7 failures: 0 duration: 0.04s" and a circular progress bar labeled "100%". Below this, there's a summary section for the `pow` function. It lists two main assertions:

- ✓ if n is negative, the result is NaN
- ✓ if n is not integer, the result is NaN

Underneath these, it says "raises x to power 3" and lists five individual test cases:

- ✓ 1 in the power 3 is 1
- ✓ 2 in the power 3 is 8
- ✓ 3 in the power 3 is 27
- ✓ 4 in the power 3 is 64
- ✓ 5 in the power 3 is 125

Each test case has a small circular arrow icon to its right.

[Open the full final example in the sandbox. ↗](#)

## Riepilogo

In BDD, le specifiche (spec) vengono come primo passo, vengono seguite dall'implementazione. Alla fine avremmo sia le specifiche che il codice.

Le spec vengono utilizzate in tre modi:

1. **Tests** garantire che il codice funzioni correttamente.
2. **Docs** – il titolo di `describe` e `it` specificano cosa la funzione faccia.
3. **Examples** – i test sono dei veri e propri esempi su come la funzione si comporti e come può essere utilizzata.

Con le spec, possiamo migliorare, cambiare e anche riscrivere il codice da zero in totale sicurezza ed essere sicuri che tutto continui a funzionare come dovrebbe.

Questo è particolarmente importante specie nei grandi progetti, quando le funzioni vengono utilizzate più volte in posti diversi. Quando cambiamo una di queste funzioni, non c'è un modo pratico per controllare che queste continuino a funzionare ovunque.

Senza i test, le persone avrebbero due possibilità:

1. Cambiare qualcosa, non importa cosa. E successivamente gli utenti dovrebbero fare un rapporto quando incontrano un bug. Non sempre possiamo permetterci di farlo.
2. Essere impauriti dai cambiamenti, soprattutto se la punizione in caso di errori è severa. Un giorno queste funzioni diventeranno vecchie, e coperte di ragnatele, nessuno vorrà più utilizzarle, questa opzione non è quindi ottimale.

### **Il codice testato automaticamente evita questi problemi!**

Se il progetto viene coperto dai test, non ci saranno problemi. Infatti possiamo eseguire i test ed eseguire molte verifiche in un paio di secondi.

### **Inoltre, un codice ben testato ha un architettura più robusta.**

Ovvio, poiché è codice semplice da migliorare. Ma non è solo questo.

Per scrivere dei test, il codice dovrebbe essere organizzato in un modo tale che ogni funzione venga chiaramente descritta, con input e output ben definiti. Questo si ottiene progettando una buona architettura fin dal principio.

Nella vita reale qualche volta non è così semplice. Talvolta risulta difficile scrivere una spec prima del codice, perché non è ancora molto chiaro come dovrebbe comportarsi. Ma in generale scrivere i test rende lo sviluppo più rapido e stabile.

## **E ora?**

Più avanti nel tutorial incontrerai molte funzioni con i test integrati. Così imparerai con degli esempi pratici.

Scrivere dei test richiede delle buone conoscenze di JavaScript. Per ora possiamo imparare ad utilizzarli un passo per volta. Quindi, per ora non vi è richiesto di essere in grado di scrivere test, ma dovreste almeno essere in grado di leggerli, anche se risultano essere poco più complessi di quelli di questo capitolo.

## **Esercizi**

---

### **Cosa c'è di sbagliato in questo test?**

importanza: 5

Cosa c'è di sbagliato nel test di `pow`?

```
it("Raises x to the power n", function() {
 let x = 5;

 let result = x;
 assert.equal(pow(x, 1), result);

 result *= x;
 assert.equal(pow(x, 2), result);

 result *= x;
 assert.equal(pow(x, 3), result);
});
```

P.S. Sintatticamente il test corretto ed esegue senza errori.

[Alla soluzione](#)

## Polyfills e transpilers

Il linguaggio JavaScript si evolve costantemente. Nuove proposte per il linguaggio arrivano regolarmente, vengono analizzate, e successivamente se ritenute valide vengono aggiunte alla lista <https://tc39.github.io/ecma262/> fino a diventare delle [specifiche](#).

I team che stanno dietro il motore di JavaScript hanno le loro personali idee riguardo cosa implementare. Potrebbero quindi decidere di implementare delle proposte recenti e posticipare quelle più vecchie a causa di difficoltà nell'implementazione.

Quindi per un motore di script è naturale implementare solo le cose che si trovano nello standard.

Se si vuole rimanere aggiornati riguardo lo stato di supporto delle caratteristiche si può controllare la pagina <https://kangax.github.io/compat-table/es6/> (è molto grande, dovremmo studiare ancora molto).

Come programmati, amiamo utilizzare le più recenti caratteristiche del linguaggio!

Ma come si può fare per farle funzionare sui vecchi motori JavaScript che non le comprendono ed interpretano?

Esistono due strumenti per questo:

1. Transpilers.
2. Polyfills.

In questo capitolo cercheremo di capire il loro funzionamento ed il loro ruolo nello sviluppo web.

## Transpilers

Un [transpiler](#) è un particolare software che traduce il codice sorgente da un formato ad un altro. Può analizzare il codice moderno e riscriverlo utilizzando sintassi e costrutti meno recenti, rendendolo compatibile con i motori JavaScript meno recenti.

Es. JavaScript prima del 2020 non aveva “l’operatore di coalescenza nullo” `??`. Quindi, se un visitatore utilizza un vecchio browser, questo non potrebbe comprendere `height = height ?? 100`.

Un transpiler analizzerebbe il codice e riscriverebbe `height ?? 100` in `(height !== undefined && height !== null) ? height : 100`.

```
// prima dell'analisi del transpiler
height = height ?? 100;

// dopo l'analisi del transpiler
height = (height !== undefined && height !== null) ? height : 100;
```

Ora il codice riscritto è adatto anche ai vecchi motori JavaScript.

In genere lo sviluppatore fa girare il transpiler in locale sul proprio computer, quindi distribuisce sul server il codice riscritto.

Facendo qualche nome, [Babel](#) è uno dei più diffusi transpilers del momento.

I moderni ‘bundler’ utilizzati per ‘assemblare’ progetti, come [webpack](#), possono eseguire il transpiler automaticamente ad ogni modifica del codice, è quindi molto facile integrarlo nei processi di sviluppo.

## Polyfills

Nuove caratteristiche di un linguaggio possono riguardare, oltre alla sintassi, operatori e costrutti, anche funzioni integrate.

Ad esempio, `Math.trunc(n)` è una funzione che “tronca” la parte decimale di un numero, es. `Math.trunc(1.23)` ritorna `1`.

In alcuni (vecchissimi) motori JavaScript, non esiste `Math.trunc`, quindi il codice non funzionerebbe.

Poiché stiamo parlando di nuove funzioni, non di modifiche alla sintassi, in questo caso non ci serve un transpiler. Dobbiamo solo dichiarare la funzione mancante.

Uno script che aggiorna/aggiunge nuove funzioni è chiamato “polyfill”. Questo colma il gap ed aggiunge le implementazioni mancanti.

In questo particolare caso, il polyfill per `Math.trunc` è uno script che implementa la funzione in questo modo :

```
if (!Math.trunc) { // se la funzione non esiste
 // implementala
 Math.trunc = function(number) {
 // Math.ceil e Math.floor sono presenti anche in vecchi motori JavaScript
 // verranno trattati più avanti in questo tutorial
 return number < 0 ? Math.ceil(number) : Math.floor(number);
 };
}
```

JavaScript è un linguaggio altamente dinamico, gli script possono aggiungere/modificare qualsiasi funzione, anche quelle integrate.

Due interessanti librerie polyfills sono:

- [core js ↗](#) ha molte funzioni e consente di includere solo le funzionalità necessarie.
- [polyfill.io ↗](#) servizio che fornisce uno script con polyfill, a seconda delle funzionalità e del browser dell'utente.

## Riepilogo

In questo capitolo vorremmo motivarvi a studiare le funzionalità più moderne ed all'avanguardia" del linguaggio, anche se non sono ancora ben supportate dai motori JavaScript.

Basta non dimenticare di usare transpiler (se si utilizza la sintassi o gli operatori moderni) e i polyfill (per aggiungere funzioni che potrebbero mancare). Questi si assicureranno che il codice funzioni.

Ad esempio, in seguito, quando avrai familiarità con JavaScript, potrai configurare un sistema di compilazione del codice basato su [webpack ↗](#) con [babel-loader ↗](#) plugin.

Buone risorse che mostrano lo stato attuale del supporto per varie funzionalità:

- [https://kangax.github.io/compat-table/es6/ ↗](https://kangax.github.io/compat-table/es6/) – per puro JavaScript.
- [https://caniuse.com/ ↗](https://caniuse.com/) – per le funzioni integrate dei browsers.

P.S. Google Chrome è solitamente il browser più aggiornato con le funzionalità del linguaggio, provalo se una demo tutorial fallisce. Tuttavia, la maggior parte delle demo dei tutorial funziona con qualsiasi browser moderno.

## Oggetti: le basi

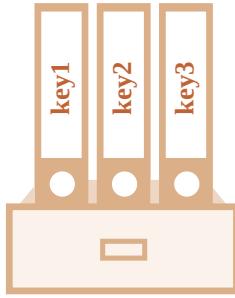
### Oggetti

Come sappiamo dal capitolo *Tipi di dati*, in Javascript ci sono otto tipi di dati. Sette di loro sono chiamati "primitivi", perché i loro valori contengono sempre un singolo elemento (una stringa, un numero, un booleano ecc).

Gli oggetti, invece, vengono utilizzati per catalogare vari tipi di dati ed altri elementi più complessi. In Javascript, essi permeano ogni aspetto del linguaggio. Dobbiamo perciò comprenderli bene prima di procedere nello studio approfondito di un qualsiasi altro argomento.

Un oggetto può essere creato tramite le parentesi graffe `{ . . . }`, con un'opzionale lista di proprietà. Una proprietà è una coppia "chiave: valore", dove "chiave" è una stringa (detta anche "nome di proprietà"), mentre "valore" può essere qualsiasi cosa.

Possiamo immaginare un oggetto come un archivio con dei documenti catalogati. Ogni dato viene archiviato utilizzando una specifica chiave. E' facile trovare un file quando se ne conosce il nome, oppure aggiungerne di nuovi o rimuovere quelli vecchi.



Un oggetto vuoto (“archivio vuoto”) può essere creato utilizzando una delle due sintassi:

```
let user = new Object(); // sintassi "costruttore oggetto"
let user = {}; // sintassi "oggetto letterale"
```



Solitamente vengono utilizzate le parentesi graffe `{ . . . }`. Questo tipo di dichiarazione viene chiamata *object literal* (“oggetto letterale”).

## Le proprietà dei literal

Possiamo inserire subito delle proprietà in `{ . . . }` come una coppia “key: value”:

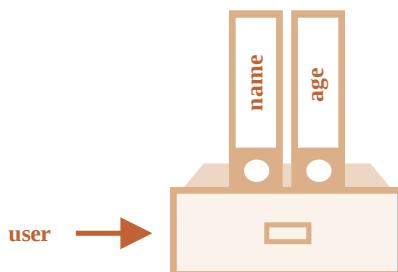
```
let user = { // un oggetto
 name: "John", // una chiave "name" memorizza il valore "John"
 age: 30 // una chiave "age" memorizza 30
};
```

Una proprietà ha una chiave (conosciuta anche come “nome” o “identificatore”) prima dei due punti `" : "`, ed un valore alla sua destra.

Nell’oggetto `user` ci sono due proprietà:

1. La prima proprietà ha come nome `"name"` e come valore `"John"`.
2. La seconda ha come nome `"age"` e come valore `30`.

L’oggetto `user` può essere visto come un archivio con due file etichettati come “name” ed “age”.



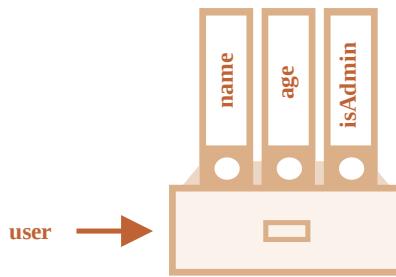
Possiamo aggiungere, rimuovere o leggere un file in qualsiasi momento.

I valori delle proprietà sono accessibili utilizzando la notazione puntata:

```
// ritorna i campi dell'oggetto:
alert(user.name); // John
alert(user.age); // 30
```

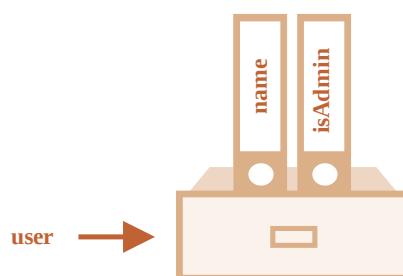
Il valore può essere di qualsiasi tipo. Aggiungiamo un booleano:

```
user.isAdmin = true;
```



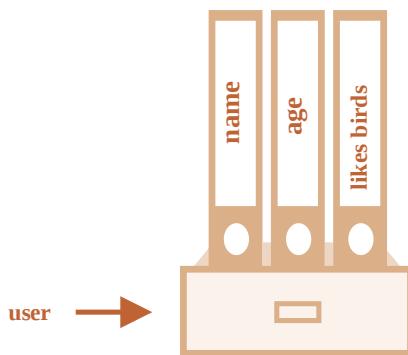
Per rimuovere una proprietà, possiamo utilizzare l'operatore `delete`:

```
delete user.age;
```



Possiamo anche utilizzare nomi di proprietà composti da più parole (“multi-parola”), ma devono essere racchiusi tra virgolette:

```
let user = {
 name: "John",
 age: 30,
 "likes birds": true // un nome di proprietà composto da più parole deve essere racchiuso tra ''
};
```



L'ultima proprietà in lista può terminare con una virgola:

```
let user = {
 name: "John",
 age: 30,
}
```

Rende più facile l'aggiunta/rimozione/spostamento delle proprietà, poiché tutte le righe hanno una virgola.

## Parentesi quadre

Per le proprietà con nomi “multi-parola” l’accesso con la notazione puntata non funziona:

```
// questo darebbe un errore di sintassi
user.likes birds = true
```

Questo perché il punto richiede che la chiave che segue sia un identificatore valido. Un identificatore non deve avere spazi (oltre a seguire le altre limitazioni già studiate).

Per aggirare questo vincolo esiste una “notazione con parentesi quadre”:

```
let user = {};
// set
user["likes birds"] = true;
// get
alert(user["likes birds"]); // true
// delete
delete user["likes birds"];
```

Ora funziona. Da notare che la stringa all’interno delle parentesi va comunque messa tra virgolette (singole o doppie).

Le parentesi quadre permettono di passare il nome della proprietà come risultato di un’espressione – a differenza delle stringhe letterali --, ad esempio una variabile:

```
let key = "likes birds";
```

```
// lo stesso di user["likes birds"] = true;
user[key] = true;
```

Qui la variabile `key` può essere calcolata durante il run-time o dipendere dall'input dell'utente. Successivamente possiamo utilizzarla per accedere alla proprietà. Questa caratteristica ci fornisce una grande flessibilità.

Ad esempio:

```
let user = {
 name: "John",
 age: 30
};

let key = prompt("What do you want to know about the user?", "name");

// accesso tramite variabile
alert(user[key]); // John (se si inserisce "name")
```

La notazione puntata non può essere utilizzata in questo modo:

```
let user = {
 name: "John",
 age: 30
};

let key = "name";
alert(user.key) // undefined
```

## Proprietà calcolate

Possiamo utilizzare le parentesi quadre al momento della creazione di un oggetto letterale. Questo metodo viene chiamato *calcolo delle proprietà*.

Ad esempio:

```
let fruit = prompt("Which fruit to buy?", "apple");

let bag = {
 [fruit]: 5, // il nome della proprietà viene preso dalla variabile fruit
};

alert(bag.apple); // 5 se fruit="apple"
```

La logica dietro le proprietà calcolate è semplice: `[fruit]` significa che il nome della proprietà deve essere preso da `fruit`.

Quindi, se un utente inserisce `"apple"`, `bag` diventerà `{apple: 5}`.

Essenzialmente, questo funziona allo stesso modo di:

```
let fruit = prompt("Which fruit to buy?", "apple");
```

```
let bag = {};
// prende il nome della proprietà dalla variabile fruit
bag[fruit] = 5;
```

...Ma è meno carino.

Possiamo utilizzare anche espressioni più complesse all'interno delle parentesi quadre:

```
let fruit = 'apple';
let bag = {
 [fruit + 'Computers']: 5 // bag.appleComputers = 5
};
```

Le parentesi quadre sono molto più potenti della notazione puntata. Ci permettono di assegnare qualsiasi nome, ma sono più “ingombranti”. La maggior parte delle volte, quando il nome della proprietà è conosciuto e semplice, la notazione puntata viene preferita. Se invece necessitiamo di qualcosa di più complesso, possiamo utilizzare le parentesi quadre.

## Abbreviazione per il valore di una proprietà

Spesso usiamo delle variabili esistenti come valori per i nomi delle proprietà.

Ad esempio:

```
function makeUser(name, age) {
 return {
 name: name,
 age: age
 // ...altre proprietà
 };
}

let user = makeUser("John", 30);
alert(user.name); // John
```

Nell'esempio sopra, le proprietà hanno lo stesso nome delle variabili. Il caso d'uso di creare una proprietà da una variabile è molto comune, tanto che, per comodità, esiste una speciale *abbreviazione*.

Invece di scrivere `name: name` possiamo semplicemente scrivere `name`, come in questo esempio:

```
function makeUser(name, age) {
 return {
 name, // equivalente a name: name
 age // equivalente a age: age
 // ...
 };
}
```

Possiamo usare entrambe le proprietà, normale e abbreviata, nello stesso oggetto:

```
let user = {
 name, // equivalente a name:name
 age: 30
};
```

## Limitazioni per i nomi di una proprietà

Come già sappiamo, una variabile non può avere il nome uguale ad una parola chiave riservata al linguaggio come “for”, “let”, “return” etc.

Ma per le proprietà degli oggetti, non ci sono restrizioni:

```
// queste variabili sono tutte corrette
let obj = {
 for: 1,
 let: 2,
 return: 3
};

alert(obj.for + obj.let + obj.return); // 6
```

In breve, non ci sono limitazioni per i nomi delle proprietà. Possono essere stringhe o simboli (un tipo di dato speciale che andremo ad analizzare più avanti).

Nomi di proprietà con altri tipi “primitivi” vengono automaticamente convertiti a stringhe.

Ad esempio, un numero `0` diventa una stringa `"0"` quando viene utilizzato come chiave di una proprietà:

```
let obj = {
 0: "test" // equivale a "0": "test"
};

// entrambi gli alert accedono alla stessa proprietà (il numero 0 viene convertito nella stringa
alert(obj["0"]); // test
alert(obj[0]); // test (stessa proprietà)
```

Esiste una piccola falla per la proprietà `__proto__`. Non possiamo impostarla ad un valore diverso dal tipo oggetto:

```
let obj = {};
obj.__proto__ = 5; // assegnamo un numero
alert(obj.__proto__); // [object Object] - il valore è un oggetto, non ha funzionato come ci si
```

Come possiamo osservare nel codice sopra, l’assegnazione del numero intero `5` è stata ignorata.

Studieremo più nel dettaglio `__proto__` nel [capitolo](#), e vedremo come [sistemare](#) questo comportamento.

## Controllo di esistenza, operatore “in”

Un'importante caratteristica degli oggetti, in Javascript, è che è possibile accedere a una qualsiasi proprietà. Non ci sarà alcun errore se la proprietà non esiste!

L'accesso ad una variabile non esistente ritornerà `undefined`. Possiamo quindi facilmente verificare se una proprietà esiste:

```
let user = {};

alert(user.noSuchProperty === undefined); // true significa "nessuna proprietà"
```

Esiste anche uno speciale operatore `"in"` per lo stesso scopo.

La sintassi è:

```
"key" in object
```

Ad esempio:

```
let user = { name: "John", age: 30 };

alert("age" in user); // true, significa che user.age esiste
alert("blabla" in user); // false, significa che user.blabla non esiste
```

Da notare che alla sinistra di `in` deve esserci il *nome di una proprietà*. Questa, solitamente, è una stringa.

Se omettiamo le virgolette attorno alla proprietà da cercare, verrà cercata una variabile con quel nome e verrà utilizzato il suo valore. Ad esempio:

```
let user = { age: 30 };

let key = "age";
alert(key in user); // true, prende il nome da key e controlla l'esistenza della proprietà
```

### **i Utilizzare "in" con le proprietà che contengono undefined**

Solitamente, il confronto stretto con "===" undefined funziona correttamente. Ma c'è un particolare caso in cui questo fallisce, mentre con "in" funziona correttamente.

Questo accade quando una proprietà esiste, ma contiene undefined :

```
let obj = {
 test: undefined
};

alert(obj.test); // è undefined, quindi -- non esiste la proprietà?

alert("test" in obj); // true, la proprietà esiste!
```

Nel codice sopra, tecnicamente, la proprietà obj.test esiste. Quindi l'operatore in funziona.

Situazioni come questa capitano raramente, perché solitamente non si assegna undefined. Si usa più comunemente null per valori "sconosciuti" o "vuoti". Quindi l'operatore in è più un ospite "esotico" nel codice.

## **Il ciclo "for...in"**

Per attraversare tutte le chiavi di un oggetto, esiste una speciale forma di ciclo: for .. in. Questo è completamente diverso da for(;;).

La sintassi:

```
for (key in object) {
 // esegue il corpo del ciclo per ogni proprietà dell'oggetto
}
```

Ad esempio, proviamo a mostrare tutte le proprietà di user :

```
let user = {
 name: "John",
 age: 30,
 isAdmin: true
};

for (let key in user) {
 // keys
 alert(key); // name, age, isAdmin
 // valori delle keys
 alert(user[key]); // John, 30, true
}
```

Da notare che tutti i costrutti "for" ci consentono di dichiarare delle variabili da utilizzare all'interno del ciclo stesso, come let key in questo esempio.

Inoltre possiamo utilizzare qualsiasi altra variabile al posto di `key`. Ad esempio `"for(let prop in obj)"` è molto utilizzato.

## Ordinato come un oggetto

Gli oggetti sono ordinati? In altre parole, se iteriamo un oggetto, otterremo le sue proprietà nello stesso ordine in cui le abbiamo aggiunte?

Una risposta breve è: "sono ordinati in modo speciale": le proprietà che hanno numeri interi come chiavi vengono ordinate, le altre appaiono seguendo l'ordine di creazione. Seguiranno maggiori dettagli.

Per fare un esempio, consideriamo un oggetto con dei prefissi telefonici:

```
let codes = {
 "49": "Germany",
 "41": "Switzerland",
 "44": "Great Britain",
 // ...
 "1": "USA"
};

for (let code in codes) {
 alert(code); // 1, 41, 44, 49
}
```

L'oggetto può essere utilizzato per suggerire una lista di opzioni all'utente. Se stiamo sviluppando un sito dedicato al pubblico tedesco probabilmente vorranno vedersi apparire come primo valore `49`.

Se proviamo ad eseguire il codice, vedremo un risultato totalmente inaspettato:

- USA (1) viene per primo
- poi Switzerland (41) e a seguire gli altri.

I prefissi telefonici seguono un ordine crescente; questo accade perché sono numeri interi. Quindi vedremo `1, 41, 44, 49`.

### **i Proprietà degli interi? Cos'è?**

La "proprietà degli interi" è un termine che indica una stringa che può essere convertita da e ad un intero senza subire modifiche.

Quindi "49" segue la proprietà degli interi, perché quando viene trasformato in un numero intero e riportato a stringa, rimane uguale. Ad esempio "+49" e "1.2" non lo sono:

```
// Math.trunc è una proprietà integrata che rimuove la parte decimale
alert(String(Math.trunc(Number("49")))); // "49", rimane uguale
alert(String(Math.trunc(Number("+49")))); // "49", è diverso da "+49" => non è un numero
alert(String(Math.trunc(Number("1.2")))); // "1", è diverso da "1.2" => non è un numero
```

...Differentemente, se le chiavi non sono numeri interi, vengono restituite nell'ordine di creazione, ad esempio:

```

let user = {
 name: "John",
 surname: "Smith"
};
user.age = 25; // aggiungiamo un'altra

// le proprietà non intere vengono elencate nell'ordine di creazione
for (let prop in user) {
 alert(prop); // name, surname, age
}

```

Quindi per sistemare il problema con i prefissi telefonici, possiamo “barare” rendendo i prefissi non interi. Questo lo otteniamo inserendo un `"+"` prima di ogni numero.

Come nel codice sotto:

```

let codes = {
 "+49": "Germany",
 "+41": "Switzerland",
 "+44": "Great Britain",
 // ...
 "+1": "USA"
};

for (let code in codes) {
 alert(+code); // 49, 41, 44, 1
}

```

Ora funziona come previsto.

## Riepilogo

Gli oggetti sono arrays associativi con diverse caratteristiche speciali:

Possono memorizzare proprietà (coppie di chiave-valore) in cui:

- Il nome della proprietà (chiave) deve essere composta da una o più stringhe o simboli (solitamente stringhe).
- I valori possono essere di qualsiasi tipo.

Per accedere ad una proprietà possiamo utilizzare:

- La notazione puntata: `obj.property`.
- La notazione con parentesi quadre `obj["property"]`. Questa notazione consente di accettare chiavi dalle variabili, come `obj[varWithKey]`.

Operatori specifici:

- Per cancellare una proprietà: `delete obj.prop`.
- Per controllare se un una proprietà con un certo nome esiste: `"key" in obj`.
- Per iterare un oggetto: `for(let key in obj)`.

Gli oggetti vengono assegnati e copiati per riferimento. In altre parole, la variabile non memorizza il “valore dell’oggetto”, ma piuttosto un “riferimento” (indirizzo di memoria). Quindi copiando questa variabile o passandola come argomento ad una funzione, fornirà un riferimento all’oggetto e non una copia. Tutte le operazioni effettuate su un oggetto copiato per riferimento (come aggiungere/rimuovere proprietà) vengono effettuate sullo stesso oggetto.

Quello che abbiamo studiato in questo capitolo viene chiamato “oggetto semplice”, o solo `Object`.

Ci sono altri tipi di oggetti in Javascript:

- `Array` per memorizzare dati ordinati,
- `Date` per memorizzare informazioni riguardo date e orari,
- `Error` per memorizzare informazioni riguardo errori.
- ...e molti altri.

Ognuno di questi ha le sue caratteristiche speciali che studieremo più avanti. Qualche volta le persone dicono cose tipo “Array type” (“tipo Array”) o “Date type” (“tipo Data”), ma formalmente non sono dei tipi, appartengono al tipo di dato “object”. Sono semplicemente delle estensioni.

Gli oggetti in JavaScript sono molto potenti. Qui abbiamo grattato solamente la superficie, l’argomento è veramente ampio. Lavoreremo molto con gli oggetti per impararne ulteriori caratteristiche.

## ✓ Esercizi

---

### Hello, object

importanza: 5

Scrivi il seguente codice, una riga per ogni azione:

1. Crea un oggetto vuoto `user`.
2. Aggiungi la proprietà `name` con valore `John`.
3. Aggiungi la proprietà `surname` con valore `Smith`.
4. Cambia il valore di `name` con `Pete`.
5. Rimuovi la proprietà `name` dall’oggetto.

[Alla soluzione](#)

---

### Controlla se è vuoto

importanza: 5

Scrivi la funzione `isEmpty(obj)` che ritorna `true` se l’oggetto non ha proprietà, altrimenti ritorna `false`.

Dovrebbe funzionare con queste istruzioni:

```
let schedule = {};
```

```
alert(isEmpty(schedule)); // true
schedule["8:30"] = "get up";
alert(isEmpty(schedule)); // false
```

Apri una sandbox con i test. [↗](#)

[Alla soluzione](#)

---

## Oggetti costanti?

importanza: 5

E' possibile modificare un oggetto dichiarato con `const`? Cosa ne pensi?

```
const user = {
 name: "John"
};

// does it work?
user.name = "Pete";
```

[Alla soluzione](#)

---

## Somma le proprietà dell'oggetto

importanza: 5

Abbiamo un oggetto che memorizza il salario dei dipendenti del team:

```
let salaries = {
 John: 100,
 Ann: 160,
 Pete: 130
}
```

Scrivi il codice per sommare tutti i salari contenuti e memorizza il risultato in `sum`. Dovrebbe essere `390`.

Se `salaries` è vuoto il risultato dovrebbe essere `0`.

[Alla soluzione](#)

---

## Moltiplica le proprietà numeriche per 2

importanza: 3

Crea una funzione `multiplyNumeric(obj)` che moltiplica tutte le proprietà numeriche di `obj` per `2`.

Ad esempio:

```
// before the call
let menu = {
 width: 200,
 height: 300,
 title: "My menu"
};

multiplyNumeric(menu);

// after the call
menu = {
 width: 400,
 height: 600,
 title: "My menu"
};
```

Nota che `multiplyNumeric` non deve ritornare nulla. Deve solamente modificare l'oggetto.

P.S. Usa `typeof` per controllare il tipo.

[Apri una sandbox con i test.](#)

[Alla soluzione](#)

## Oggetti: riferimento e copia

Una delle maggiori differenze tra oggetti e primitivi è che gli oggetti vengono memorizzati e copiati “per riferimento”, mentre i primitivi (stringhe, numeri, booleani, ecc...) vengono sempre copiati “per valore”.

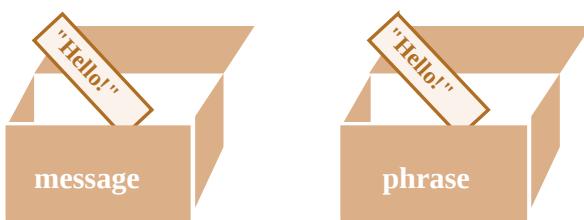
Questa differenza è facile da comprendere se andiamo a guardare il comportamento del linguaggio quando copiamo un valore.

Partiamo con un primitivo, ad esempio una stringa.

Qui facciamo una copia di `message` in `phrase`:

```
let message = "Hello!";
let phrase = message;
```

Come risultato otteniamo due variabili distinte, ognuna delle quali contiene la stringa `"Hello!"`.



E’ un risultato abbastanza ovvio, giusto?

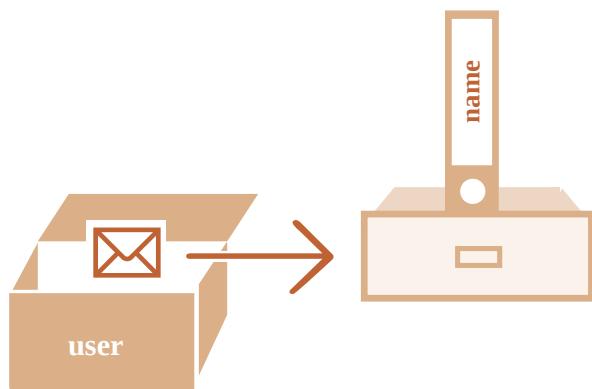
Gli oggetti non funzionano allo stesso modo.

**Una variabile assegnata ad un oggetto non contiene l'oggetto in sé, ma il suo “indirizzo in memoria” – in altre parole “un riferimento” all’oggetto.**

Diamo un’occhiata a un esempio di tale variabile:

```
let user = {
 name: "John"
};
```

Ed ecco come viene effettivamente archiviata in memoria:



L’oggetto è archiviato da qualche parte nella memoria (a destra nell’immagine), mentre la variabile `user` (a sinistra) contiene il “riferimento” ad esso.

Potremmo immaginare la “variabile oggetto” `user`, come un foglio di carta con scritto l’indirizzo dell’oggetto.

Quando eseguiamo azioni con l’oggetto, ad es. leggere una proprietà `user.name`, il motore JavaScript guarda cosa c’è a quell’indirizzo ed esegue l’operazione sull’oggetto reale.

Ecco perché è così importante.

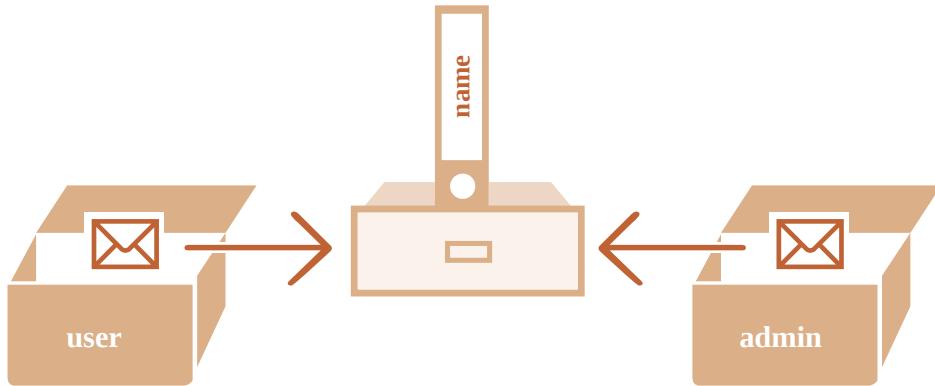
**Quando una “variabile oggetto” viene copiata, in realtà viene copiato il riferimento, ma l’oggetto in sé non viene duplicato.**

Esempio:

```
let user = { name: "John" };

let admin = user; // copia il riferimento
```

Ora abbiamo due variabili, entrambe contengono il riferimento allo stesso oggetto:



Come puoi vedere, l'oggetto è uno solo, ma ora con due variabili che si riferiscono ad esso.

Possiamo usare entrambe le variabili per accedere all'oggetto e modificarne il contenuto:

```
let user = { name: 'John' };

let admin = user;

admin.name = 'Pete'; // modificato dal riferimento in "admin"

alert(user.name); // 'Pete', le modifiche sono visibili dal riferimento in "user"
```

E' come se avessimo un armadietto con due chiavi e ne usassimo una (`admin`) per aprirlo ed apportare delle modiche al contenuto. Quindi, successivamente, potremmo aprire lo stesso armadietto con un'altra chiave (`user`) ed accedere al contenuto modificato.

## Confronto per riferimento

Due oggetti sono uguali solo se sono lo stesso oggetto. Suona un po' strano, ma ora chiariremo.

Qui `a` e `b` si riferiscono allo stesso oggetto, quindi sono uguali:

```
let a = {};
let b = a; // copia il riferimento

alert(a == b); // true, entrambe le variabili si riferiscono allo stesso oggetto
alert(a === b); // true
```

Qui, invece, due oggetti identici (entrambi vuoti), ma indipendenti, non soddisfano l'uguaglianza:

```
let a = {};
let b = {};// due oggetti indipendenti

alert(a == b); // false
```

Per confronti tra oggetti (Es. `obj1 > obj2`) o con primitivi (Es. `obj == 5`), gli oggetti vengono convertiti in primitivi. Vedremo molto presto come avviene questa conversione, anche

se, a dire il vero, questo tipo di confronto è molto raro e generalmente è il risultato di un errore di programmazione.

## Clonazione e unione, `Object.assign`

Come abbiamo detto, copiare una “variabile oggetto” crea un ulteriore riferimento allo stesso oggetto.

Quindi, come possiamo fare se abbiamo bisogno di duplicare un oggetto? Creare una copia indipendente, un clone?

Anche questo è fattibile, ma con un po' di difficoltà visto che JavaScript non ha alcun metodo integrato per farlo. In realtà non è un'operazione frequente, il più delle volte la copia per riferimento è adatta alla situazione.

Ma se proprio ne abbiamo bisogno, allora dobbiamo creare un nuovo oggetto e replicare la struttura di quello esistente iterando le sue proprietà e copiandole a livello primitivo.

Così:

```
let user = {
 name: "John",
 age: 30
};

let clone = {} // il nuovo oggetto vuoto

// copiamo nella variabile clone tutte le proprietà di user
for (let key in user) {
 clone[key] = user[key];
}

// ora clone è un oggetto completamente indipendente ma con lo stesso contenuto di user
clone.name = "Pete"; // cambiamo la proprietà name

alert(user.name); // nell'oggetto originale è rimasto "John"
```

Possiamo anche usare il metodo [Object.assign](#).

La sintassi è:

```
Object.assign(dest, [src1, src2, src3...])
```

- Il primo argomento `dest` è l'oggetto di destinazione.
- Gli argomenti successivi `src1, ..., srcN` (possono essere quanti vogliamo) sono gli oggetti da copiare.
- Il metodo copia tutte le proprietà degli oggetti `src1, ..., srcN` in quello di destinazione `dest`.
- Viene restituito l'oggetto `dest`.

Per fare un esempio, possiamo unire diversi oggetti in uno solo:

```

let user = { name: "John" };

let permissions1 = { canView: true };
let permissions2 = { canEdit: true };

// copia tutte le proprietà da permissions1 e permissions2 in user
Object.assign(user, permissions1, permissions2);

// ora user = { name: "John", canView: true, canEdit: true }

```

Se una delle proprietà copiate è già presente nell'oggetto di destinazione, verrà sovrascritta.

```

let user = { name: "John" };

Object.assign(user, { name: "Pete" });

alert(user.name); // ora user = { name: "Pete" }

```

Possiamo anche usare `Object.assign` per sostituire il ciclo `for..in` nella clonazione semplice:

```

let user = {
 name: "John",
 age: 30
};

let clone = Object.assign({}, user);

```

Vengono copiate tutte le proprietà di `user` nell'oggetto vuoto, il quale, poi, viene restituito.

There are also other methods of cloning an object, e.g. using the [spread syntax](#) `clone = {...user}`, covered later in the tutorial.

Finora abbiamo assunto che le proprietà di `user` fossero primitive. Ma le proprietà possono anche essere riferimenti ad altri oggetti. Come si fa in questo caso?

Così:

```

let user = {
 name: "John",
 sizes: {
 height: 182,
 width: 50
 }
};

alert(user.sizes.height); // 182

```

In questo caso non è sufficiente copiare `clone.sizes = user.sizes`. Siccome `user.sizes` è un oggetto, verrà copiato per riferimento. Quindi `clone` e `user` condivideranno lo stesso oggetto “sizes”.

Vediamo un esempio:

```

let user = {
 name: "John",
 sizes: {
 height: 182,
 width: 50
 }
};

let clone = Object.assign({}, user);

alert(user.sizes === clone.sizes); // true, è lo stesso oggetto

// user e clone condividono sizes
user.sizes.width++; // cambiamo una proprietà da una parte
alert(clone.sizes.width); // 51, e vediamo il risultato dall'altra

```

Per risolvere questo problema, dobbiamo usare un ciclo di clonazioni che esaminerà ogni valore di `user[key]` e, nel caso sia un oggetto, replichi anche la sua struttura. Questa operazione è chiamata “deep cloning” (copia profonda).

Per implementare questa funzione possiamo usare la ricorsione. Oppure, per non reinventare la ruota, possiamo usare qualcosa di già pronto, ad esempio `_.cloneDeep(obj)` ↗ dalla libreria JavaScript `lodash` ↗.

### i Gli oggetti dichiarati con `const` possono essere modificati

Un importante “side effect” della memorizzazione per riferimento è che un oggetto dichiarato con `const` può essere modificato.

Esempio:

```

const user = {
 name: "John"
};

user.name = "Pete"; // (*)

alert(user.name); // Pete

```

Saremmo portati a pensare che la linea `(*)` causi un errore, ma non è così. Il valore di `user` è costante, si riferisce sempre allo stesso oggetto, ma le proprietà dell'oggetto sono libere di cambiare.

In altre parole, `const user` restituisce un errore solo se proviamo a riassegnare in toto `user=...`.

Detto questo, se vogliamo veramente rendere invariabili le proprietà di un oggetto, possiamo farlo, ma con un metodo totalmente differente. Ne parleremo nel capitolo [Attributi e descrittori di proprietà](#).

## Riepilogo

Gli oggetti sono assegnati e copiati per riferimento. In altre parole una variabile non contiene il “valore oggetto” ma un “riferimento” (indirizzo in memoria) di quel valore. Quindi copiando tale variabile o passandola come argomento di una funzione si copia quel riferimento, non l’oggetto stesso.

Tutte le operazioni su un riferimento duplicato (come aggiungere o rimuovere proprietà) hanno effetto sul medesimo oggetto.

Per creare una “vera copia” (clonare) effettuare una cosiddetta “shallow copy” (copia superficiale) con `Object.assign` (gli oggetti nidificati vengono copiati per riferimento), oppure un “deep cloning” (copia profonda) con funzioni tipo `_.cloneDeep(obj)` ↗ .

## Garbage collection ("Spazzatura")

In JavaScript la gestione della memoria viene eseguita automaticamente ed è invisibile. Noi creiamo primitive, oggetti, funzioni... Tutte queste occupano memoria.

Cosa succede quando qualcosa non è più necessario? Come fa JavaScript a scoprirla e pulirla?

### Raggiungibilità

Il principale concetto della gestione della memoria in JavaScript è la *raggiungibilità*.

Semplicemente una valore “raggiungibile” deve essere accessibile o utilizzabile. Questa proprietà ne garantisce la permanenza in memoria.

1. C’è una gruppo di valori che sono intrinsecamente raggiungibili, che non possono essere cancellati per ovvie ragioni.

Ad esempio:

- Funzioni in esecuzione, i loro parametri e le loro variabili locali.
- Funzioni che fanno parte della catena delle chiamate annidate, i loro parametri e le loro variabili locali.
- Variabili globali.
- (ce ne sono altri, anche interni)

Questi valori sono detti *radici*.

2. Qualsiasi altro valore viene considerato raggiungibile se è possibile ottenerlo per riferimento o per catena di riferimenti.

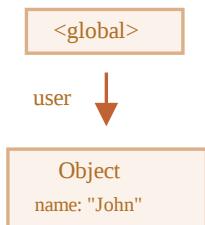
Ad esempio, se c’è un oggetto in una variabile locale, e l’oggetto ha una proprietà che si riferisce ad un altro oggetto, quest’ultimo viene considerato raggiungibile. E anche tutti i suoi riferimenti lo saranno. Seguiranno esempi più dettagliati.

C’è un processo che lavora in background nel motore JavaScript, chiamato [garbage collector](#) ↗ . Monitora gli oggetti e rimuove quelli che sono diventati irraggiungibili.

### Un semplice esempio

Qui un esempio molto semplice:

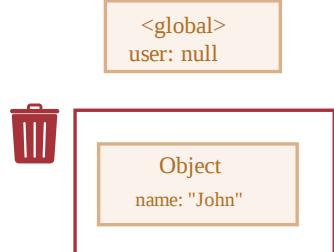
```
// user ha un riferimento all'oggetto
let user = {
 name: "John"
};
```



Qui la freccia indica un riferimento ad un oggetto. La variabile globale `"user"` fa riferimento all'oggetto `{name: "John"}` (lo chiameremo John per brevità). La proprietà `"name"` di John memorizza un tipo primitivo, quindi viene descritto all'interno dell'oggetto.

Se il valore di `user` viene sovrascritto, il riferimento viene perso.

```
user = null;
```



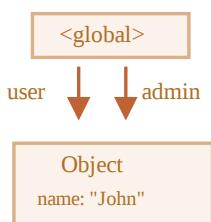
Ora John diventa irraggiungibile. Non c'è modo per accedervi, nessun riferimento. Il Garbage collector scarterà il dato per liberare la memoria.

## Due riferimenti

Ora proviamo a pensare di aver copiato il riferimento da `user` su `admin`:

```
// user ha un riferimento all'oggetto
let user = {
 name: "John"
};

let admin = user;
```



Ora se facciamo:

```
user = null;
```

...L'oggetto rimane raggiungibile tramite `admin`, quindi è in memoria. Se sovrascriviamo anche `admin`, allora verrà rimosso.

## Oggetti interconnessi

Ora vediamo un esempio più complesso. La famiglia:

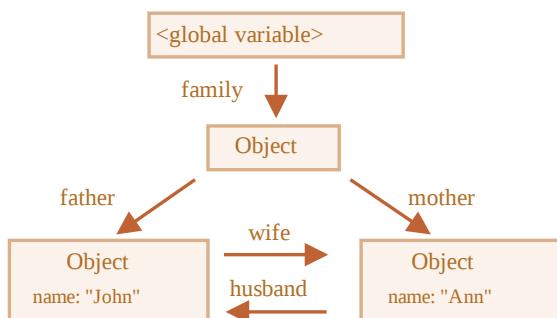
```
function marry(man, woman) {
 woman.husband = man;
 man.wife = woman;

 return {
 father: man,
 mother: woman
 }
}

let family = marry({
 name: "John"
, {
 name: "Ann"
});
```

La funzione `marry` "sposa" due oggetti facendo in modo di fornire un riferimento l'uno per l'altro, e ritorna un oggetto che li contiene entrambi.

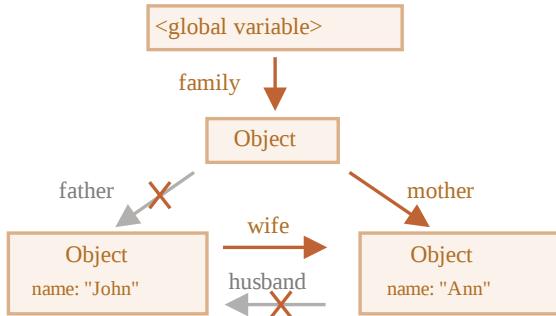
La struttura della memoria risultante:



Per ora tutti gli oggetti sono raggiungibili.

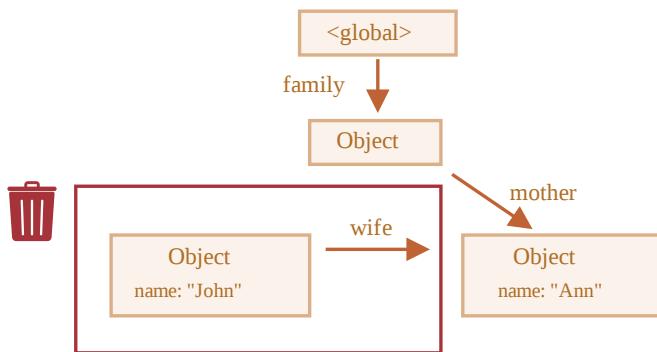
Ora proviamo a rimuovere due riferimenti:

```
delete family.father;
delete family.mother.husband;
```



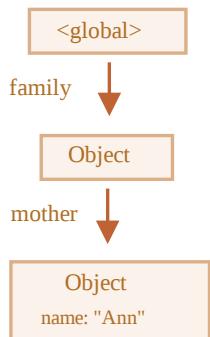
Non è sufficiente cancellare solo uno dei due riferimenti, perché l'oggetto rimarrebbe comunque raggiungibile.

Ma se li cancelliamo entrambi, allora John non ha più modo di essere raggiunto:



I riferimenti in uscita non contano. Solo quelli in entrata possono rendere l'oggetto raggiungibile. Quindi, John risulta ora irraggiungibile e verrà quindi rimosso dalla memoria, come tutti i suoi dati visto che sono inaccessibili.

Dopo la pulizia del Garbage collector:



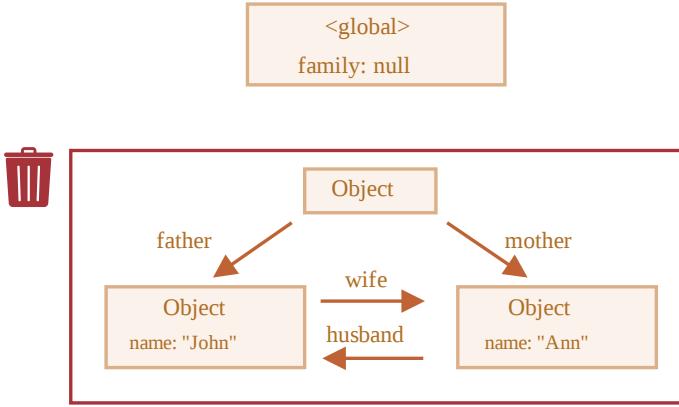
## Isola irraggiungibile

E' possibile che l'intera isola degli oggetti collegati a vicenda diventi inaccessibile e venga quindi rimossa dalla memoria.

L'oggetto sorgente è lo stesso di quello sopra. Poi:

```
family = null;
```

La memoria ora risulta così:



Questo esempio dimostra quanto sia importante il concetto della raggiungibilità.

E' ovvio che John e Ann siano ancora collegati, ma questo non è sufficiente.

L'oggetto che li conteneva "family" è stato rimosso dalla radice, non esistono quindi dei riferimenti, l'isola diventa irraggiungibile e verrà cancellata.

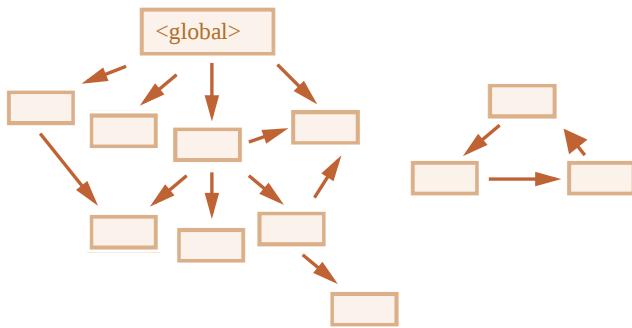
## Algoritmi interni

L'algoritmo basico utilizzato dal garbage collector viene chiamato "mark-and-sweep" (segnala e pulisci).

Vengono seguiti questi step per eseguire un processo di "garbage collection":

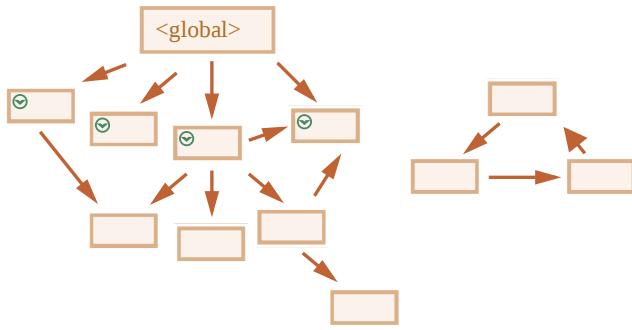
- Il garbage collector "marchia" (ricorda) le radici.
- Successivamente visita e "marchia" tutti i riferimenti contenuti.
- Successivamente visita gli oggetti marcati e marca i vari riferimenti. Tutti gli oggetti visitati vengono ricordati, così da non ricontrollarli nuovamente in futuro.
- ...E così via fino ad aver controllato tutti i riferimenti (raggiungibili dalle radici).
- Tutti gli oggetti tranne quelli marcati vengono rimossi.

Ad esempio, rende la struttura del nostro oggetto del tipo:

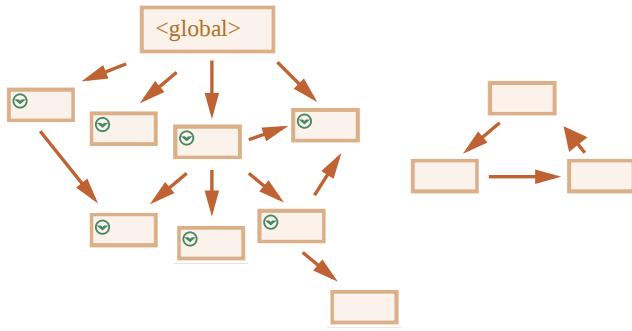


Possiamo chiaramente vedere un "isola irraggiungibile" nella parte destra. Ora vediamo come la gestisce l'algoritmo "mark-and-sweep".

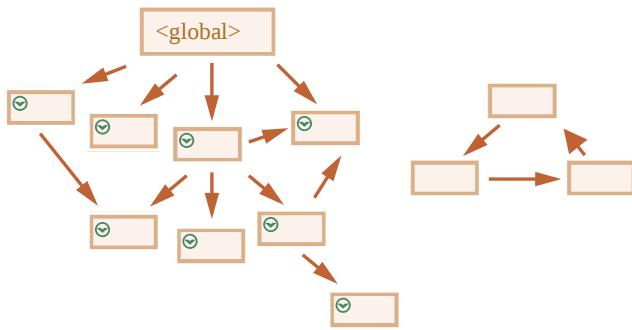
Il primo step sta nel marcare le radici:



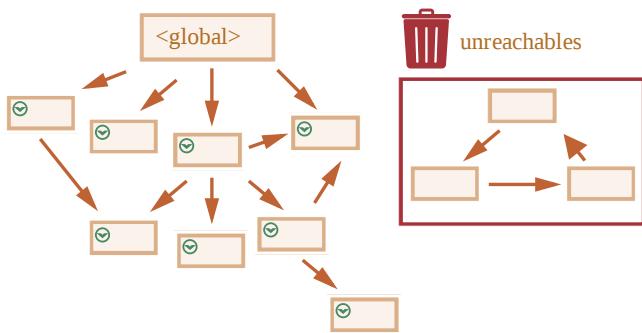
Poi vengono marcati i loro riferimenti:



...E i loro riferimenti, finché non si esauriscono:



Ora gli oggetti che non sono stati visitati vengono considerati irraggiungibili e verranno rimossi:



Questo è il concetto che sta dietro il funzionamento del Garbage collector.

JavaScript applica diverse ottimizzazioni per renderlo più rapido.

Alcune delle ottimizzazioni:

- **Raggruppamento generazionale** – gli oggetti vengono divisi in due gruppi: “nuovi” e “vecchi”. Molti oggetti vengono costruiti, eseguono il proprio lavoro e muoiono, quindi possono essere

rimossi rapidamente. Quelli che vivono abbastanza al lungo vengono considerati come "vecchi" e verranno controllati con minore intensità

- **Raggruppamento incrementale** – se ci sono molti oggetti, e ci mettessimo a controllare interi gruppi per marcarli, si perderebbe molto tempo, questo ritardo diventerebbe visibile durante l'esecuzione. Quindi i motori JavaScript tentano di dividere il processo in diverse parti. Questi pezzi vengono controllati uno per uno, separatamente. E' richiesto l'utilizzo di un registro per tenere traccia dei cambiamenti, in cambio avremmo tanti piccoli ritardi piuttosto che uno singolo ma enorme.
- **Raggruppamento per inattività** – il garbage collector cerca di eseguire i suoi processi solo nei momenti in cui la CPU è inattiva, per ridurre al minimo possibile i ritardi durante l'esecuzione.

Ci sono altre ottimizzazioni per ottimizzare i processi del Garbage collector. Anche se mi piacerebbe poterli spiegare in dettaglio, sono costretto a fermarmi, poiché le varie ottimizzazioni dipendono dai motori che vengono utilizzati. Inoltre, i motori cambiano, si aggiornano e diventano sempre più "avanzati". Quindi se siete realmente interessati, vi lascio qualche link sotto.

## Riepilogo

Le principali cose da conoscere:

- Il processo di Garbage collection viene eseguito automaticamente. Non possiamo forzarlo o bloccarlo.
- Gli oggetti vengono mantenuti in memoria solo finché risultano raggiungibili.
- Essere riferimento di qualunque altro oggetto non significa essere raggiungibili (dalla radice): un gruppo di oggetti possono diventare irraggiungibili in un solo colpo.

I motori moderni applicano algoritmi avanzati di garbage collection.

Un buon libro "The Garbage Collection Handbook: The Art of Automatic Memory Management" (R. Jones et al) che descrive alcuni degli algoritmi.

Se conoscete la programmazione a basso livello, informazioni più dettagliate riguardo il Garbage collector V8 sono disponibili nell'articolo [A tour of V8: Garbage Collection ↗](#).

[V8 blog ↗](#) pubblica articoli riguardo i cambiamenti nella gestione della memoria. Ovviamente per apprendere il processo di garbage collection, è fortemente consigliato imparare il funzionamento del garbage collector V8 leggendo il blog [Vyacheslav Egorov ↗](#) che ha lavorato come ingegnere per lo sviluppo del V8. Vi dico "V8" perché è quello più utilizzato e maggiormente spiegato su internet. Per gli altri motori, gli approcci sono simili, ci sono alcune sottili differenze.

Le conoscenze profonde dei motori sono importanti quando necessitate di ottimizzazioni a basso livello. Potrebbe essere un buon traguardo dopo essere diventati familiari con il linguaggio.

## Metodi degli oggetti,"this"

Gli oggetti solitamente vengono creati per rappresentare entità del mondo reale, come utenti, prodotti e molto altro:

```
let user = {
 name: "John",
```

```
 age: 30
};
```

Inoltre, nel mondo reale, un utente può *agire*: selezionare qualcosa dalla lista degli acquisti, effettuare login, logout etc.

In JavaScript le azioni vengono rappresentate tramite le funzioni.

## Esempio di un metodo

Per iniziare, insegniamo a `user` a salutare:

```
let user = {
 name: "John",
 age: 30
};

user.sayHi = function() {
 alert("Hello!");
};

user.sayHi(); // Hello!
```

Qui abbiamo appena utilizzato un'espressione di funzione per creare una funzione ed assegnarla alla proprietà `user.sayHi` dell'oggetto.

Successivamente possiamo chiamarla. Ora l'utente può parlare!

Una funzione che è una proprietà di un oggetto si chiama *metodo*.

Quindi, nell'esempio abbiamo un metodo `sayHi` dell'oggetto `user`.

Ovviamente possiamo utilizzare una funzione già dichiarata come metodo:

```
let user = {
 // ...
};

// prima la dichiariamo
function sayHi() {
 alert("Hello!");
};

// poi la aggiungiamo come metodo
user.sayHi = sayHi;

user.sayHi(); // Hello!
```

## Programmazione orientata agli oggetti

Quando scriviamo codice utilizzando gli oggetti per rappresentare le entità, questa viene definita [programmazione orientata agli oggetti](#), in breve: "OOP".

OOP è una grande cosa, un ambito di interesse con i propri studi. Come scegliere le giuste entità? Come organizzare le interazioni tra loro? Questa è l'architettura di un codice, e ci sono molti libri importanti che trattano questo argomento, come "Design Patterns: Elements of Reusable Object-Oriented Software" di E.Gamma, R.Helm, R.Johnson, J.Vissides oppure "Object-Oriented Analysis and Design with Applications" di G.Booch, e molti altri.

## La forma breve dei metodi

Esiste una sintassi più breve per i metodi in un oggetto letterale:

```
// questi oggetti fanno la stessa cosa

user = {
 sayHi: function() {
 alert("Hello");
 }
};

// la sintassi più breve risulta più carina
user = {
 sayHi() { // equivalente a "sayHi: function(){...}"
 alert("Hello");
 }
};
```

Come possiamo notare, si può omettere "function" e scrivere solamente `sayHi()`.

A dire la verità, la notazione non è proprio uguale. Ci sono delle sottili differenze legate all'ereditarietà degli oggetti (le studieremo più avanti), ma per ora non hanno importanza. Nella maggior parte dei casi la forma breve viene preferita.

## "this" nei metodi

E' molto comune che, per eseguire determinate azioni, un metodo abbia necessità di accedere alle informazioni memorizzate nell'oggetto.

Ad esempio, il codice dentro `user.sayHi()` potrebbe aver bisogno del nome dell'`user`.

Per accedere all'oggetto, un metodo può utilizzare la parola chiave `this`.

Il valore di `this` è l'oggetto "prima del punto", quello che ha eseguito la chiamata del metodo.

Ad esempio:

```
let user = {
 name: "John",
 age: 30,
 sayHi() {
```

```
// "this" is the "current object"
alert(this.name);
}

};

user.sayHi(); // John
```

In fase di esecuzione, quando viene chiamato il metodo `user.sayHi()`, il valore di `this` sarà `user`.

Tecnicamente, è possibile accedere all'oggetto anche senza `this`; lo si fa tramite riferimento alla variabile esterna:

```
let user = {
 name: "John",
 age: 30,

 sayHi() {
 alert(user.name); // "user" piuttosto di "this"
 }
};
```

...Questo codice è instabile. Se decidessimo di copiare `user` in un'altra variabile, ad esempio `admin = user` e sovrascrivere `user` con qualcos'altro, verrebbe allora effettuato l'accesso all'oggetto sbagliato.

Dimostriamolo:

```
let user = {
 name: "John",
 age: 30,

 sayHi() {
 alert(user.name); // porta ad un errore
 }
};

let admin = user;
user = null; // sovrascriviamo per rendere tutto più ovvio

admin.sayHi(); // Errore: non possiamo leggere la proprietà 'name' di null
```

Se scriviamo `this.name` piuttosto di `user.name` all'interno di `alert`, il codice funzionerà.

## “this” non ha limiti

In JavaScript, la parola chiave “this” si comporta diversamente da come fa in molti altri linguaggi di programmazione. Essa può essere usata in qualsiasi funzione, anche se non si tratta del metodo di un oggetto.

Non c'è alcun errore di sintassi in un codice come questo:

```
function sayHi() {
 alert(this.name);
}
```

Il valore di `this` viene valutato al momento dell'esecuzione. E può essere un valore qualsiasi.

Ad esempio, la stessa funzione potrebbe avere diversi "this" quando viene chiamata da oggetti diversi:

```
let user = { name: "John" };
let admin = { name: "Admin" };

function sayHi() {
 alert(this.name);
}

// utilizziamo la stessa funzione su due oggetti
user.f = sayHi;
admin.f = sayHi;

// queste chiamate hanno un this diverso
// "this" all'interno della funzione è riferito all'oggetto "prima del punto"
user.f(); // John (this == user)
admin.f(); // Admin (this == admin)

admin['f'](); // Admin (il punto o le parentesi quadre forniscono entrambi accesso ai metodi - n
```

La regola è semplice: se viene chiamato `obj.f()`, allora, durante la chiamata di `f`, `this` si riferisce a `obj`. Nell'esempio sopra assume il valore sia di `user` che di `admin`.

### Invocazione senza un oggetto: `this == undefined`

Possiamo anche chiamare la funzione senza un oggetto:

```
function sayHi() {
 alert(this);
}

sayHi(); // undefined
```

In questo caso `this` è `undefined` in modalità strict. Se tentiamo di accedere a `this.name`, ci sarà un errore.

Se non è attiva la modalità *strict* il valore di `this` in questo caso sarà *l'oggetto globale* (`window` in un browser, lo studieremo più avanti nel capitolo [Oggetto globale](#)). Questo strano comportamento ha delle motivazioni storiche, che `"use strict"` risolve.

Solitamente questo tipo di chiamate significano un errore di programmazione. Se c'è un `this` all'interno di una funzione, ci si aspetta che sia chiamato da un oggetto.

### **i** Le conseguenze della libertà di `this`

Se avete utilizzato altri linguaggi di programmazione, probabilmente sarete abituati all'idea di un "this limitato": quando viene definito un metodo in un oggetto, questo avrà sempre in `this` il riferimento all'oggetto.

In JavaScript `this` è "libero", il suo valore viene calcolato durante l'esecuzione e non dipende da dove il metodo è stato definito, ma piuttosto dall'oggetto "prima del punto".

Il concetto di valutare `this` durante l'esecuzione ha i suoi pregi e difetti. Da una parte una funzione può essere riutilizzata per oggetti diversi, dall'altra questa grande flessibilità può essere fonte di molti errori.

Il nostro scopo non è di giudicare se questa caratteristica del linguaggio sia buona o cattiva, ma di capire come lavorare con essa sfruttandone i benefici ed evitando i problemi.

## Le arrow functions non hanno "this"

Ad esempio, qui `arrow()` usa `this` preso dal metodo esterno `user.sayHi()`:

```
let user = {
 firstName: "Ilya",
 sayHi() {
 let arrow = () => alert(this.firstName);
 arrow();
 }
};

user.sayHi(); // Ilya
```

Questa è una speciale caratteristica delle arrow functions; è utile quando non vogliamo avere un ulteriore `this`, ma utilizzare quello del contesto esterno. Più avanti nel capitolo [Arrow functions](#) rivisitate studieremo più in dettaglio le arrow functions.

## Riepilogo

- Le funzioni che vengono memorizzate come proprietà di un oggetto vengono dette "metodi".
- I metodi consentono agli oggetti di "agire", come `object.doSomething()`.
- I metodi possono riferirsi all'oggetto tramite `this`.

Il valore `this` viene definito durante l'esecuzione (run-time).

- Quando una funzione viene dichiarata, può utilizzare `this`, ma questo `this` non avrà alcun valore fino a che la funzione non verrà chiamata.
- Una funzione può essere copiata in vari oggetti.
- Quando una funzione viene chiamata come "metodo": `object.method()`, il valore di `this` durante la chiamata si riferisce a `object`.

Da notare che le arrow functions sono speciali: non hanno `this`. Quando si prova ad accedere a `this` in una funzione freccia, questo verrà preso dal contesto esterno.

## ✓ Esercizi

### Utilizzare "this" in un oggetto letterale

importanza: 5

Qui la funzione `makeUser` ritorna un oggetto.

Qual è il risultato dell'accesso a `ref`? Perché?

```
function makeUser() {
 return {
 name: "John",
 ref: this
 };

let user = makeUser();

alert(user.ref.name); //Qual è il risultato?
```

[Alla soluzione](#)

### Create una calcolatrice

importanza: 5

Create un oggetto `calculator` con tre metodi:

- `read()` richiede tramite prompt due valori e li salva come proprietà dell'oggetto.
- `sum()` ritorna la somma dei valori salvati.
- `mul()` moltiplica i valori salvati e ritorna il risultato.

```
let calculator = {
 // ... your code ...
};

calculator.read();
alert(calculator.sum());
alert(calculator.mul());
```

[Esegui la demo](#)

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

### Concatenazione

importanza: 2

Qui abbiamo un oggetto `ladder` che ci consente di salire e scendere:

```
let ladder = {
 step: 0,
 up() {
 this.step++;
 },
 down() {
 this.step--;
 },
 showStep: function() { // shows the current step
 alert(this.step);
 }
};
```

Ora, se abbiamo bisogno di eseguire più chiamate in sequenza, possiamo:

```
ladder.up();
ladder.up();
ladder.down();
ladder.showStep(); // 1
ladder.down();
ladder.showStep(); // 0
```

Modicare il codice di `up`, `down` e `showStep` per rendere le chiamate concatenabili, come in questo esempio:

```
ladder.up().up().down().showStep().down().showStep(); // shows 1 then 0
```

Questo approccio è largamente utilizzato dalle librerie JavaScript.

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

## Costruttore, operatore "new"

La sintassi `{ ... }` ci consente di creare un oggetto. Ma spesso abbiamo bisogno di creare multipli oggetti simili, come ad esempio più utenti, oggetti del menu e molto altro.

Questo può essere fatto utilizzando un costruttore e l'operatore `"new"`.

## Costruttore

Tecnicamente un costruttore è una normale funzione. Ma ci sono due convenzioni:

1. Vengono denominati con la prima lettera maiuscola.
2. Questi dovrebbero essere eseguiti solo con l'operatore `"new"`.

Ad esempio:

```
function User(name) {
```

```
this.name = name;
this.isAdmin = false;
}

let user = new User("Jack");

alert(user.name); // Jack
alert(user.isAdmin); // false
```

Quando una funzione viene eseguita con `new`, esegue questi passaggi:

1. Un nuovo oggetto, vuoto, viene creato ed assegnato a `this`.
2. Viene eseguito il corpo della funzione. Solitamente questo modifica `this`, aggiungendo nuove proprietà.
3. Viene ritornato il valore assegnato a `this`.

In altre parole, `new User(...)` fa qualcosa del genere:

```
function User(name) {
 // this = {} (implicito)

 // aggiungiamo proprietà a this
 this.name = name;
 this.isAdmin = false;

 // return this; (implicito)
}
```

Quindi `let user = new User("Jack")` dà lo stesso risultato di:

```
let user = {
 name: "Jack",
 isAdmin: false
};
```

Ora, se vogliamo creare altri utenti, possiamo chiamare `new User("Ann")`, `new User("Alice")` e così via. Molto più rapido rispetto all'utilizzare ogni volta oggetti letterali; risulta anche più facile da leggere.

Questo è il principale scopo dei costruttori – implementare codice riutilizzabile per la creazione di oggetti.

Ribadiamo – tecnicamente, ogni funzione (eccetto le arrow functions, siccome non hanno `this`) può essere utilizzata come costruttore. Ovvero: ogni funzione può essere eseguita con `new`. La “prima lettera maiuscola” è semplicemente una convenzione, per rendere esplicito che la funzione deve essere eseguita con `new`.

### **i new function() { ... }**

Se abbiamo molte linee di codice utili alla creazione di un unico oggetto, possiamo raggrupparle in un costruttore richiamato contestualmente, come qui:

```
// create a function and immediately call it with new
let user = new function() {
 this.name = "John";
 this.isAdmin = false;

 // ...altro codice per la creazione di user
 // magari logiche complesse e istruzioni
 // variabili locali etc
};
```

Il costruttore non può essere chiamato nuovamente, perché non è salvato da nessuna parte; viene solo creato e chiamato. Questo trucco consente di incapsulare un codice che costruisce un singolo oggetto, senza necessità di riutilizzo futuro.

## Costruttori modalità test: new.target

### **i Tecniche avanzate**

La sintassi presentata nella seguente sezione viene utilizzata raramente; potete tranquillamente saltarla se non vi interessa sapere proprio tutto.

Dentro la funzione, possiamo controllare quando questa viene chiamata con `new` e quando senza, utilizzando una speciale proprietà `new.target`.

Questa risulta ‘`undefined`’ per le chiamate normali, mentre contiene la funzione se viene chiamata con `new`:

```
function User() {
 alert(new.target);
}

// senza "new":
User(); // undefined

// con "new":
new User(); // function User { ... }
```

Questo può essere utilizzato per consentire ad entrambe le chiamate di funzionare (con `new` e senza), quindi sia in “modalità costruttore” che in “modalità regolare”.

Possiamo anche fare in modo che le chiamate con `new` e quelle regolari facciano la stessa cosa, come in questo esempio:

```
function User(name) {
 if (!new.target) { // se mi esegui senza new
 return new User(name); // ...Aggiungo new al posto tuo
```

```

 }

 this.name = name;
}

let john = User("John"); // reindirizza la chiamata a new User
alert(john.name); // John

```

Questo approccio viene adottato in alcune librerie per rendere la sintassi più flessibile. Rende possibile la chiamata della funzione sia con `new` che senza.

Ma non è un'ottima cosa utilizzare la doppia sintassi ovunque, perché omettendo `new` il codice perde in leggibilità. Con la parola chiave `new` possiamo sapere con certezza che si sta creando un nuovo oggetto.

## Return nel costruttore

Solitamente, i costruttori non hanno l'istruzione `return`. Il loro compito è di eseguire tutto ciò che è necessario a creare l'oggetto lavorando su `this`; quest'ultimo sarà il risultato.

Se decidiamo di inserire un'istruzione di `return`, vanno seguite delle semplici regole:

- Se `return` viene invocato con un oggetto, questo verrà ritornato al posto di `this`.
- Se `return` viene invocato con un tipo primitivo, verrà ignorato.

In altre parole, `return` con un oggetto ritorna quell'oggetto; in tutti gli altri casi verrà ritornato `this`.

Ad esempio, qui `return` sovrascrive `this` ritornando un oggetto:

```

function BigUser() {

 this.name = "John";

 return { name: "Godzilla" }; // <-- ritorna questo oggetto
}

alert(new BigUser().name); // Godzilla

```

Qui invece abbiamo un esempio con un `return` vuoto (potremmo anche ritornare un qualsiasi valore di tipo primitivo):

```

function SmallUser() {

 this.name = "John";

 return; // <-- returns this
}

alert(new SmallUser().name); // John

```

Solitamente i costruttori non hanno l'istruzione `return`. Abbiamo comunque riportato, per completezza, quel che succede se si tenta di ritornare un oggetto.

### Omettere le parentesi

Possiamo anche omettere le parentesi dopo `new`, se non ci sono argomenti:

```
let user = new User; // <-- no parentheses
// same as
let user = new User();
```

L'omissione delle parentesi non viene considerata come "buona programmazione", la sintassi comunque lo permette.

## Metodi in un costruttore

Utilizzare costruttori per creare degli oggetti ci dà un grande vantaggio in termini di flessibilità. Il costruttore può avere dei parametri che definiscono come costruire l'oggetto, e cosa "metterci dentro".

Ovviamente, possiamo aggiungere a `this` non solo proprietà, ma anche metodi.

Ad esempio, `new User(name)` crea un oggetto con un nome (passato come `name`) e un metodo `sayHi`:

```
function User(name) {
 this.name = name;

 this.sayHi = function() {
 alert("My name is: " + this.name);
 };
}

let john = new User("John");

john.sayHi(); // My name is: John

/*
john = {
 name: "John",
 sayHi: function() { ... }
}
*/
```

Per creare oggetti più complessi, esiste una sintassi più avanzata, [classes](#), che copriremo più avanti.

## Riepilogo

- Le funzioni di costruzione, o meglio, i costruttori, sono solo delle normali funzioni; seguono però una convenzione comune che prevede di denominarle con la prima lettera maiuscola.

- Un costruttore dovrebbe essere chiamato solamente utilizzando `new`. Questo tipo di chiamata
- implica la creazione di un oggetto vuoto, `this`, che verrà popolato entro la fine della funzione.

Possiamo utilizzare i costruttori per costruire molti oggetti simili tra loro.

JavaScript fornisce costruttori per la maggior parte degli oggetti integrati nel linguaggio: come `Date` per le date, `Set` per gli insiemi e molti altri che studieremo più avanti.

### Oggetti, ci ritorneremo!

In questo capitolo abbiamo coperto solamente le basi degli oggetti e dei costruttori. Era necessario conoscerne le basi per capire meglio i data types e le funzioni che studieremo nel prossimo capitolo.

Dopo questo, ritorneremo sugli oggetti e li analizzeremo più in dettaglio nei capitoli [Prototypes](#), [inheritance](#) e [Classi](#).

## Esercizi

### Due funzioni – un oggetto

importanza: 2

E' possibile creare due funzioni `A` e `B` tali che `new A() == new B()` ?

```
function A() { ... }
function B() { ... }

let a = new A;
let b = new B;

alert(a == b); // true
```

Se pensi che sia possibile, prova a scrivere il codice.

[Alla soluzione](#)

### Create una nuova Calculator

importanza: 5

Scrivete un costruttore `Calculator` che crea oggetti con 3 metodi:

- `read()` richiede due valori utilizzando `prompt` e li memorizza nelle proprietà dell'oggetto.
- `sum()` ritorna la somma delle proprietà.
- `mul()` ritorna il prodotto delle proprietà.

Ad esempio:

```
let calculator = new Calculator();
calculator.read();
```

```
alert("Sum=" + calculator.sum());
alert("Mul=" + calculator.mul());
```

[Esegui la demo](#)

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

## Create un nuovo Accumulator

importanza: 5

Scrivete un costruttore `Accumulator(startingValue)`.

L'oggetto che viene creato dovrebbe:

- Salvare il “valore corrente” nella proprietà `value`. Il valore di partenza viene impostato prendendo il valore passato all’argomento del costruttore `startingValue`.
- Il metodo `read()` dovrebbe richiedere tramite `prompt` un numero e sommarlo a `value`.

In altre parole, la proprietà `value` è la somma di tutti i numeri inseriti dall’utente partendo dal valore iniziale `startingValue`.

Qui una demo del codice:

```
let accumulator = new Accumulator(1); // initial value 1

accumulator.read(); // adds the user-entered value
accumulator.read(); // adds the user-entered value

alert(accumulator.value); // shows the sum of these values
```

[Esegui la demo](#)

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

## Concatenamento opzionale '?.'

### ⚠ Aggiunta di recente

Questa funzionalità è stata aggiunta di recente al linguaggio. Vecchi browsers potrebbero richiedere polyfills.

Il concatenamento opzionale (optional chaining), `? .`, è un modo sicuro di accedere alle proprietà annidate di un oggetto, anche nel caso in cui una proprietà intermedia non dovesse esistere.

## Il problema della “proprietà inesistente”

Se avete appena cominciato a leggere questo tutorial e a imparare JavaScript, forse questo problema non lo avete ancora affrontato, ma è piuttosto comune.

Ad esempio, ipotizziamo di avere un oggetto `user`, in cui sono memorizzate le informazioni relative ai nostri utenti.

La maggior parte dei nostri utenti possiedono l'indirizzo nella proprietà `user.address`, la via in `user.address.street`, ma qualcuno potrebbe non averle fornite.

In questo caso, quando proviamo ad accedere a `user.address.street`, e l'utente non possiede un indirizzo, avremo un errore:

```
let user = {} // un utente senza la proprietà "address"

alert(user.address.street); // Errore!
```

Questo è il risultato che ci si aspetta. JavaScript funziona in questo modo. Se `user.address` è `undefined`, un tentativo di accesso a `user.address.street` fallirà con un errore.

Nella maggior parte dei casi, preferiremmo avere `undefined` piuttosto di un errore (in questo caso con il significato “nessuna via”).

... Un altro esempio. Il metodo `document.querySelector('.elem')` ritorna un oggetto che corrisponde ad un elemento della pagina web, che ritorna `null` quando l'elemento non esiste.

```
// document.querySelector('.elem') è null se non esiste l'elemento
let html = document.querySelector('.elem').innerHTML; // errore se è null
```

Di nuovo, se un elemento non esiste, otterremo un errore nel tentativo di accedere a `.innerHTML` di `null`. In alcuni casi, in cui l'assenza di un elemento è normale, vorremo evitare l'errore e accettare come risultato `html = null`.

Come possiamo farlo?

La soluzione più ovvia sarebbe di controllare il valore utilizzando `if` o l'operatore condizionale `?` prima di accedere alle proprietà, come nell'esempio:

```
let user = {};

alert(user.address ? user.address.street : undefined);
```

Funziona, nessun errore... Ma è poco elegante. Come potete vedere, `"user.address"` appare due volte nel codice. Per proprietà molto più annidate, potrebbe diventare un problema, in quanto saranno necessarie molte più ripetizioni.

Ad esempio, proviamo a recuperare il valore di `user.address.street.name`.

Dobbiamo verificare sia `user.address` che `user.address.street`:

```
let user = {}; // l'utente non ha address
alert(user.address ? user.address.street ? user.address.street.name : null : null);
```

Questo è semplicemente terribile, un codice del genere potrebbe essere difficile da comprendere. Ci sarebbe un modo migliore per riscriverlo, utilizzando l'operatore `&&`:

```
let user = {}; // l'utente non ha address
alert(user.address && user.address.street && user.address.street.name); // undefined (nessun e
```

Concatenare con `&&` l'intero percorso verso la proprietà ci assicura che tutti i componenti esistano (in caso contrario, la valutazione si interrompe), ma non è comunque l'ideale.

Come potete vedere, i nomi delle proprietà sono ancora duplicate nel codice. Ad esempio, nel codice sopra, `user.address` è ripetuto tre volte.

Questo è il motivo per cui la concatenazione opzionale `?.` è stata aggiunta al linguaggio. Per risolvere questo problema una volta per tutte!

## Concatenazione opzionale

La concatenazione opzionale `?.` interrompe la valutazione se il valore prima di `?.` è `undefined` o `null`, e ritorna `undefined`.

**D'ora in poi, in questo articolo, per brevità diremo che qualcosa “esiste” se non è né `null` né `undefined`.**

In altre parole, `value?.prop`:

- funziona come `value.prop`, se `value` esiste,
- altrimenti (quando `value` è `undefined/null`) ritorna `undefined`.

Vediamo un modo sicuro per accedere a `user.address.street` utilizzando `?.`:

```
let user = {}; // user non possiede l'address
alert(user?.address?.street); // undefined (nessun errore)
```

Il codice è corto e pulito, non c'è alcuna duplicazione.

Leggere l'indirizzo con `user?.address` funzionerebbe anche se l'oggetto `user` non esistesse:

```
let user = null;

alert(user?.address); // undefined
alert(user?.address.street); // undefined
```

Da notare: la sintassi `?.` rende opzionale il valore che la precede, nulla di più.

Ad esempio in `user?.address.street.name` il costrutto `?.` permette alla proprietà `user` di essere `null/undefined` in sicurezza (e ritornare `undefined` in questo caso), ma questo vale solamente per `user`. Si accederà alle altre proprietà normalmente. Se vogliamo che anche altre proprietà siano opzionali, dobbiamo rimpiazzare `.` con `?..`.

### Non abuse della concatenazione opzionale

Dovremmo utilizzare `?.` solamente quando va bene che una proprietà possa non esistere.

Ad esempio, considerando la logica del nostro codice, l'oggetto `user` deve necessariamente esistere, mentre `address` è opzionale, quindi dovremmo scrivere `user.address?.street`, non `user?.address?.street`.

Quindi, se `user` dovesse essere `undefined` per errore, otterremo un errore e potremmo sistemarlo. Altrimenti, gli errori di programmazione potrebbero essere silenziati in modo non appropriato, rendendo il debug molto difficile.

### La variabile che precede `?.` deve essere dichiarata

Se non esiste alcuna variabile `user`, allora `user?.anything` provocherà un errore:

```
// ReferenceError: user is not defined
user?.address;
```

La variabile deve essere dichiarata (ad esempio come `let/const/var user` o come parametro di funzione). La concatenazione opzionale funziona solamente con le variabili dichiarate.

## Corto circuito

Come detto in precedenza, il costrutto `?.` interrompe immediatamente (manda in “corto circuito”) la valutazione se la proprietà a destra non esiste.

Quindi, nel caso ci siano ulteriori chiamate a funzione o side-effects, questi non verranno eseguiti.

Ad esempio:

```
let user = null;
let x = 0;

user?.sayHi(x++); // non esiste "sayHi", quindi l'esecuzione non raggiungerà x++

alert(x); // 0, valore non incrementato
```

## Altre varianti: `?().` `?[]`

La concatenazione opzionale `?.` non è un operatore, ma uno speciale costrutto sintattico, che funziona anche con le funzioni e le parentesi quadre.

Ad esempio, `?.( )` viene utilizzato per invocare una funzione che potrebbe non esistere.

Nel codice sotto, alcuni dei nostri utenti possiedono il metodo `admin`, mentre altri no:

```
let userAdmin = {
 admin() {
 alert("I am admin");
 }
};

let userGuest = {};

userAdmin.admin?(); // I am admin

userGuest.admin?(); // niente (il metodo non esiste)
```

Qui, in entrambe le righe, come prima cosa abbiamo utilizzato il punto (`user1.admin`) per ottenere la proprietà `admin`, poiché l'oggetto `user` deve necessariamente esistere, quindi l'accesso è sicuro.

Successivamente `?.( )` controlla la parte sinistra: se la funzione `admin` esiste, allora viene eseguita (ciò che accade con `user1`). Altrimenti (con `user2`) la valutazione si interrompe senza errori.

La sintassi `?.` funziona anche con le parentesi `[]` (invece del punto `.`). Come nei casi precedenti, possiamo accedere con sicurezza alla proprietà di un oggetto che potrebbe non esistere.

```
let key = "firstName";

let user1 = {
 firstName: "John"
};

let user2 = null;

alert(user1?[key]); // John
alert(user2?[key]); // undefined
```

Possiamo anche utilizzare `?.` con `delete`:

```
delete user?.name; // cancella user.name se l'utente esiste
```



**Possiamo utilizzare `?.` per l'accesso e la rimozione sicura, ma non per la scrittura**

La concatenazione opzionale `?.` non ha alcun significato alla sinistra di un'assegnazione.

Ad esempio:

```
let user = null;

user?.name = "John"; // Errore, non funziona
// poiché valuta undefined = "John"
```

Non è così intelligente.

## Riepilogo

La concatenazione opzionale `?.` ha tre forme:

1. `obj?.prop` – ritorna `obj.prop` se `obj` esiste, altrimenti ritorna `undefined`.
2. `obj?.[prop]` – ritorna `obj[prop]` se `obj` esiste, altrimenti ritorna `undefined`.
3. `obj.method?.()` – invoca `obj.method()` se `obj.method` esiste, altrimenti ritorna `undefined`.

Come possiamo vedere, le tre forme sono semplici da utilizzare. Il costrutto `?.` verifica che la parte sinistra non sia `null/undefined`; se non lo è, permette alla valutazione di proseguire, altrimenti la interrompe immediatamente.

La concatenazione di `?.` permette di accedere in sicurezza a proprietà annidate.

In ogni caso, dovremmo applicare `?.` con prudenza, solamente nei casi in cui è accettabile che la parte sinistra possa non esistere. In questo modo evitiamo di nascondere errori di programmazione, nel caso ce ne siano.

## Il tipo Symbol

Secondo le specifiche, le chiavi delle proprietà di un oggetto possono essere di tipo stringa o di tipo symbol("simbolo"). Non sono accettati numeri o valori booleani, solamente stringhe e symbol.

Finora abbiamo utilizzato solo stringhe. Ora proviamo a vedere i vantaggi forniti dal tipo symbol.

## Symbol

Il valore "Symbol" rappresenta un identificatore univoco.

Un valore di questo tipo può essere creato usando `Symbol()`:

```
// id è un nuovo symbol
let id = Symbol();
```

Al momento della creazione, possiamo anche fornire una descrizione al symbol (chiamata nome del symbol), utile per il debugging:

```
// id è un symbol con descrizione "id"
let id = Symbol("id");
```

I Symbol garantiscono di essere unici. Anche se creiamo più simboli con la stessa descrizione, saranno comunque valori differenti. La descrizione è utile solamente come etichetta, non ha effetto su nulla.

Ad esempio, qui abbiamo due simboli con la stessa descrizione – ma non sono uguali:

```
let id1 = Symbol("id");
let id2 = Symbol("id");

alert(id1 == id2); // false
```

Se conosci Ruby, od altri linguaggi che possiedono la keyword “symbol”, fai attenzione a non confonderti. I symbol in JavaScript sono differenti.

### I symbol non si auto-convertono a stringa

Molti valori in JavaScript supportano la conversione implicita a stringa. Ad esempio, possiamo utilizzare `alert` con quasi tutti i valori, e funzionerà ugualmente. Symbol è un tipo speciale, non verrà convertito.

Ad esempio, questo `alert` vi mostrerà un errore:

```
let id = Symbol("id");
alert(id); // TypeError: Cannot convert a Symbol value to a string
```

Questo è un “controllo del linguaggio” per prevenire pasticci, perché le stringhe e i symbol sono fondamentalmente differenti e non dovrebbero essere accidentalmente convertiti gli uni negli altri.

Se vogliamo veramente mostrare un symbol, dobbiamo convertirlo esplicitamente utilizzando `.toString()`:

```
let id = Symbol("id");
alert(id.toString()); // Symbol(id), ora funziona
```

Oppure usare la proprietà `symbol.description` per mostrare solo la descrizione:

```
let id = Symbol("id");
alert(id.description); // id
```

## Proprietà “nascoste”

Symbol ci consente di creare delle proprietà “nascoste” dentro un oggetto, che nessun’altra parte del codice potrà leggere o modificare.

Ad esempio, se stiamo lavorando con l'oggetto `user`, che appartiene a un codice di terze parti, e vogliamo aggiungere identificatore.

```
let user = { // appartiene ad un altro codice
 name: "John"
};

let id = Symbol("id");

user[id] = "ID Value";

alert(user[id]); // possiamo accedere ai dati utilizzando il symbol come chiave
```

Qual'è il beneficio di utilizzare `Symbol("id")` piuttosto che `"id"`?

Poiché l'oggetto `user` appartiene a un altro codice che lo utilizza, non dovremmo aggiungervi alcun campo, non è sicuro. Ma un `symbol` non è accessibile accidentalmente, il codice di terze parti probabilmente non lo vedrà nemmeno, quindi andrà tutto bene.

Inoltre, immagina che un altro script necessiti di avere il proprio identificatore all'interno di `user`. Potrebbe essere un'altra libreria JavaScript, e gli script sarebbero completamente inconsapevoli l'uno dell'altro.

Quindi ogni script può creare il suo `Symbol("id")`:

```
// ...
let id = Symbol("id");

user[id] = "Their id value";
```

Non ci saranno conflitti, poiché i simboli saranno sempre differenti, anche se hanno lo stesso nome.

Invece se proviamo ad utilizzare una stringa `"id"` piuttosto che `symbol`, otterremo un conflitto:

```
let user = { name: "John" };

// il nostro script utilizza la proprietà "id"
user.id = "ID Value";

// ...se in seguito un altro script utilizza "id" per i suoi scopi...

user.id = "Their id value"
// boom! sovrascritto! non intendeva danneggiare il codice del collega, ma lo ha fatto!
```

## Symbol in un *object literal*

Se vogliamo utilizzare un `symbol` in un *object literal* `{ . . . }`, abbiamo bisogno di includerlo nelle parentesi quadre.

Come nell'esempio:

```
let id = Symbol("id");
```

```
let user = {
 name: "John",
 [id]: 123 // non "id: 123"
};
```

Questo è necessario perché abbiamo bisogno del valore di `id` come chiave, e non della stringa "id".

## I symbol vengono ignorati da for...in

Le proprietà di tipo symbol non vengono considerate dal ciclo `for .. in`.

Ad esempio:

```
let id = Symbol("id");
let user = {
 name: "John",
 age: 30,
 [id]: 123
};

for (let key in user) alert(key); // name, age (nessun symbol)

// l'accesso diretto al symbol funziona
alert("Direct: " + user[id]);
```

Anche `Object.keys(user)` li ignora. Questo fa parte del principio generale di occultazione delle proprietà symbol. Se uno script esterno o una libreria eseguisse un ciclo sul nostro oggetto, non avrebbe inaspettatamente accesso a una proprietà di tipo symbol.

Invece `Object.assign` esegue la copia sia delle proprietà di tipo stringa sia di quelle symbol:

```
let id = Symbol("id");
let user = {
 [id]: 123
};

let clone = Object.assign({}, user);

alert(clone[id]); // 123
```

Non c'è nulla di strano. E' una semplice scelta di design. L'idea è che quando vogliamo clonare o unire oggetti, solitamente abbiamo intenzione di copiarne *tutte* le proprietà (incluse quelle di tipo symbol come `id`).

## Symbol globali

Come abbiamo visto, solitamente i symbol sono differenti, anche se hanno lo stesso nome. Ma a volte vogliamo che symbol con nomi uguali vengano visti come la stessa entità. Ad esempio, parti differenti del codice potrebbero voler accedere al symbol `"id"`, riferendosi alla stessa proprietà.

Per soddisfare questa necessità, esiste un *registro globale dei symbol*. Possiamo creare dei symbol al suo interno ed accedervi successivamente, e questo garantirà che lo stesso nome

ritornerà esattamente lo stesso symbol.

Per poter leggere (o creare in caso di assenza) un symbol nel registro va usata la sintassi `Symbol.for(key)`.

Questa chiamata controlla il registro globale, se trova un symbol descritto dalla `key`, lo ritorna, altrimenti crea un nuovo simbolo `Symbol(key)` e lo memorizza nel registro con la chiave `key`.

Ad esempio:

```
// lettura dal registro globale
let id = Symbol.for("id"); // se il symbol non esiste, allora viene creato

// lo legge nuovamente
let idAgain = Symbol.for("id");

// lo stesso symbol
alert(id === idAgain); // true
```

I symbols dentro il registro vengono chiamati *symbol globali*. Se abbiamo bisogno di molti symbol, che siano accessibili ovunque nel codice – questo è il modo.

### Assomigliano a Ruby

In alcuni linguaggi di programmazione, come Ruby, c'è un solo symbol per nome.

In JavaScript, come possiamo vedere, questo è vero solo per i symbol globali.

## Symbol.keyFor

Per i symbol globali, non esiste solo `Symbol.for(key)` per accedere ad un symbol, è possibile anche la chiamata inversa: `Symbol.keyFor(sym)`, che fa l'opposto: ritorna il nome di un symbol globale.

Ad esempio:

```
// estraiamo symbol dal nome
let sym = Symbol.for("name");
let sym2 = Symbol.for("id");

// estraiamo il nome dal symbol
alert(Symbol.keyFor(sym)); // name
alert(Symbol.keyFor(sym2)); // id
```

La funzione `Symbol.keyFor` internamente utilizza il registro globale dei symbol per cercare la chiave del symbol. Quindi non avrà alcun effetto per symbol non globali. Se gli viene passato un symbol non globale, non sarà in grado di trovarlo e ritornerà `undefined`.

Detto questo, ogni symbol possiede una proprietà `description`.

Ad esempio:

```
let globalSymbol = Symbol.for("name");
let localSymbol = Symbol("name");
```

```
alert(Symbol.keyFor(globalSymbol)); // name, symbol globale
alert(Symbol.keyFor(localSymbol)); // undefined, non globale

alert(localSymbol.description); // name
```

## Symbol di sistema

In JavaScript esistono diversi *symbol di sistema*, e possiamo utilizzarli per gestire vari aspetti dei nostri oggetti.

Questi sono elencati in dettaglio nella tabella [Well-known symbols ↗](#) :

- `Symbol.hasInstance`
- `Symbol.isConcatSpreadable`
- `Symbol.iterator`
- `Symbol.toPrimitive`
- ...e molti altri.

Ad esempio, `Symbol.toPrimitive` ci consente di descrivere l'oggetto per la conversione ad un tipo primitivo. Vedremo meglio come utilizzarlo a breve.

Altri symbol diventeranno più familiari quando studieremo le corrispondenti caratteristiche del linguaggio.

## Riepilogo

`Symbol` è un tipo primitivo per definire identificatori univoci.

I symbol vengono creati con una chiamata a `Symbol()`, aggiungendo una descrizione opzionale (nome).

I symbol sono sempre differenti, anche se hanno lo stesso nome. Se abbiamo bisogno di avere symbol con lo stesso nome ed uguali tra loro, dovremmo utilizzare il registro globale:

`Symbol.for(key)` ritorna un symbol globale con la `key` (se non esiste la crea). Diverse chiamate di `Symbol.for` ritorneranno sempre lo stesso symbol.

I symbol hanno due principali ambiti d'uso:

1. “Nascondere” le proprietà di un oggetto. Se vogliamo aggiungere una proprietà ad un oggetto che “appartiene” ad un altro script (o libreria), possiamo creare un symbol ed utilizzarlo come chiave della proprietà. Una proprietà di tipo symbol non sarà disponibile in un `for..in`, quindi non sarà mai resa visibile. Non sarà nemmeno accessibile direttamente poiché uno script diverso non potrà avere i nostri symbol. Quindi la proprietà sarà protetta dall'uso accidentale o dalla sovrascrittura.

Possiamo quindi aggiungere “di nascosto” una proprietà in un oggetto se ne abbiamo la necessità, senza che nessun altro possa vederla, usando proprietà di tipo symbol.

2. Ci sono diversi symbol di sistema utilizzati da JavaScript che sono accessibili come `Symbol.*`. Possiamo utilizzarli per modificare alcune caratteristiche native. Ad esempio, più avanti nella guida utilizzeremo `Symbol.iterator` per `iterables`, `Symbol.toPrimitive` per impostare la [conversione da oggetto a primitivo](#).

Tecnicamente i symbol non sono nascosti al 100%. C'è un metodo nativo in JavaScript [Object.getOwnPropertySymbols\(obj\)](#) che ci consente di ottenere tutti i symbol. Esiste anche un metodo chiamato [Reflect.ownKeys\(obj\)](#) che ritorna *tutte* le chiavi di un oggetto incluse quelle di tipo symbol. Quindi non sono realmente invisibili. Ma la maggior parte delle librerie, delle funzioni native e dei costrutti non utilizzano questi metodi.

## Conversione da oggetto a primitivi

Cosa accade quando degli oggetti vengono sommati `obj1 + obj2`, sottratti `obj1 - obj2` o mostrati tramite `alert(obj)`?

JavaScript non consente di personalizzare come gli operatori lavorano sugli oggetti. Diversamente da alcuni linguaggi di programmazione, come Ruby o C++, non implementa nessun metodo speciale per gestire l'addizione (o altri operatori).

Nel caso si effettuassero queste operazioni, gli oggetti vengono convertiti automaticamente in primitivi e le operazioni vengono effettuate su questi, restituendo poi un valore anch'esso primitivo.

Questa è un'importante limitazione, in quanto il risultato di `obj1 + obj2` non può essere un altro oggetto!

Per esempio, non possiamo creare oggetti che rappresentano vettori o matrici (o achievements o altro), sommarli ed aspettarsi un oggetto "somma" come risultato. Tali architetture non sono contemplate.

Quindi, poiché non possiamo intervenire, non c'è matematica con oggetti in progetti reali. Quando succede, di solito è a causa di un errore di codice.

In questo capitolo tratteremo come un oggetto si converte in primitivo e come personalizzarlo.

Abbiamo due scopi:

1. Ci permetterà di capire cosa succede in caso di errori di programmazione, quando tali operazioni avvengono accidentalmente.
2. Ci sono eccezioni, dove tali operazioni sono possibili e funzionano bene. Per esempio, sottrazione o confronto di date (oggetti `Date`). Come vedremo più tardi.

## Regole per la conversione

Nel capitolo [Conversione di tipi](#) abbiamo visto le regole per le conversioni dei primitivi di tipo numerico, stringa e booleano. Però abbiamo lasciato un vuoto riguardo gli oggetti. Adesso che conosciamo i metodi e i symbol diventa più semplice parlarne.

1. Tutti gli oggetti sono `true` in contesto booleano. Ci sono solamente conversioni numeriche e a stringhe.
2. La conversione numerica avviene quando eseguiamo una sottrazione tra oggetti oppure applichiamo funzioni matematiche. Ad esempio, gli oggetti `Date` (che studieremo nel capitolo [Date e time](#)) possono essere sottratti, ed il risultato di `date1 - date2` è la differenza di tempo tra le due date.
3. Le conversioni a stringa – solitamente avvengono quando mostriamo un oggetto, come in `alert(obj)` e in altri contesti simili.

Possiamo perfezionare la conversione di stringhe e numeri, utilizzando metodi oggetto speciali.

Esistono tre varianti di conversione del tipo, che si verificano in varie situazioni.

Sono chiamate “hints”, come descritto in [specification ↗](#):

### "string"

Un’operazione di conversione oggetto a stringa, avviene quando un’operazione si aspetta una stringa, come `alert`:

```
// output
alert(obj);

// utilizziamo un oggetto come chiave di una proprietà
anotherObj[obj] = 123;
```

### "number"

Un’operazione di conversione oggetto a numero, come nel caso delle operazioni matematiche:

```
// conversione esplicita
let num = Number(obj);

// conversione matematica (ad eccezione per la somma binaria)
let n = +obj; // somma unaria
let delta = date1 - date2;

// confronto maggiore/minore
let greater = user1 > user2;
```

### "default"

Utilizzata in casi rari quando l’operatore “non è sicuro” del tipo da aspettarsi.

Ad esempio, la somma binaria `+` può essere utilizzata sia con le stringhe (per concatenarle) sia con i numeri (per eseguire la somma), quindi sia la conversione a stringa che quella a tipo numerico potrebbero andare bene. Oppure quando un oggetto viene confrontato usando `==` con una stringa, un numero o un symbol.

```
// somma binaria
let total = car1 + car2;

// obj == number uses the "default" hint
if (user == 1) { ... };
```

L’operatore maggiore/minore `<>` può funzionare sia con stringhe che con numeri. Ad oggi, per motivi storici, si suppone la conversione a “numero” e non quella di “default”.

Nella pratica, tutti gli oggetti integrati (tranne oggetti `Date`, che studieremo più avanti) implementano la conversione “default” nello stesso modo di quella “number”. Noi dovremmo quindi fare lo stesso.

Notate – ci sono solo tre hint. Semplice. Non esiste alcuna conversione al tipo “boolean” (tutti gli oggetti sono `true` nei contesti booleani). Se trattiamo “default” e “number” allo stesso

modo, come la maggior parte degli oggetti integrati, ci sono solo due conversioni.

**Per eseguire la conversione JavaScript tenta di chiamare tre metodi dell'oggetto:**

1. Chiama `obj[Symbol.toPrimitive](hint)` se il metodo esiste,
2. Altrimenti, se "hint" è di tipo "string"
  - prova `obj.toString()` e `obj.valueOf()`, sempre se esistono.
3. Altrimenti se "hint" è di tipo "number" o "default"
  - prova `obj.valueOf()` and `obj.toString()`, sempre se esistono.

## Symbol.toPrimitive

Iniziamo dal primo metodo. C'è un symbol integrato denominato `Symbol.toPrimitive` che dovrebbe essere utilizzato per etichettare il metodo che esegue la conversione, come nell'esempio:

```
obj[Symbol.toPrimitive] = function(hint) {
 // qui il codice per convertire questo oggetto a primitivo
 // deve ritornare un valore primitivo
 // hint = uno fra "string", "number", "default"
};
```

Se il metodo `Symbol.toPrimitive` esiste, viene utilizzato per tutti gli hint, e non sono necessari altri metodi.

Ad esempio, qui l'oggetto `user` lo implementa:

```
let user = {
 name: "John",
 money: 1000,

 [Symbol.toPrimitive](hint) {
 alert(`hint: ${hint}`);
 return hint == "string" ? `{"name": "${this.name}"}` : this.money;
 }
};

// esempi di conversione:
alert(user); // hint: string -> {"name": "John"}
alert(+user); // hint: number -> 1000
alert(user + 500); // hint: default -> 1500
```

Come possiamo vedere nel codice, `user` diventa una stringa auto-descrittiva o una quantità di soldi, in base al tipo di conversione. Il semplice metodo `user[Symbol.toPrimitive]` gestisce tutte le conversioni.

## toString/valueOf

Non esiste alcun `Symbol.toPrimitive` quindi JavaScript prova a trovare i metodi `toString` e `valueOf`:

- Per "string" hint: `toString`, e se non esiste, `valueOf` (quindi `toString` ha la priorità per la conversione di stringhe).
- Per altri hints: `valueOf`, e se non esiste, `toString` (quindi `valueOf` ha la priorità per le operazioni matematiche).

Mi metodi `toString` arrivano `valueOf` da molto lontano. Non sono symbols (i symbols non esistevano tempo fa), ma piuttosto "normali" metodi. Forniscono un modo alternativo "vecchio stile" per implementare la conversione.

Questi metodi devono restituire un valore primitivo. Se `toString` o `valueOf` ritornano un oggetto, vengono ignorati (come se non ci fosse il metodo).

Per impostazione predefinita, un oggetto semplice ha i seguenti metodi `toString` e `valueOf`:

- Il metodo `toString` ritorna una stringa `"[object Object]"`.
- Il metodo `valueOf` ritorna l'oggetto stesso.

Ecco una dimostrazione:

```
let user = {name: "John"};

alert(user); // [object Object]
alert(user.valueOf() === user); // true
```

Quindi, se proviamo a usare un oggetto come stringa, ad esempio in un `alert`, per impostazione predefinita vedremo `[object Object]`.

Il predefinito `valueOf` è menzionato qui solo per completezza, per evitare qualsiasi confusione. Come puoi vedere, restituisce l'oggetto stesso e quindi viene ignorato. Non chiedetemi perché, è per ragioni storiche. Quindi possiamo fare come se non esista.

Implementiamo questi metodi per personalizzare la conversione.

Ad esempio, qui `user` fa la stessa cosa vista sopra, utilizzando una combinazione di `toString` e `valueOf` invece di `Symbol.toPrimitive`:

```
let user = {
 name: "John",
 money: 1000,

 // per hint="string"
 toString() {
 return `name: ${this.name}`;
 },

 // per hint="number" or "default"
 valueOf() {
 return this.money;
 }
};

alert(user); // toString -> {name: "John"}
alert(+user); // valueOf -> 1000
alert(user + 500); // valueOf -> 1500
```

Spesso vogliamo un unico blocco che “catturi tutte” le conversioni a primitive. In questo caso possiamo implementare solamente `toString`:

```
let user = {
 name: "John",

 toString() {
 return this.name;
 }
};

alert(user); // toString -> John
alert(user + 500); // toString -> John500
```

In assenza di `Symbol.toPrimitive` e `valueOf`, `toString` gestirà tutte le conversioni a primitive.

## Una conversione può restituire qualsiasi tipo primitivo

Una cosa importante da sapere riguardo le conversioni primitive è che non devono necessariamente ritornare il tipo “hint” (suggerito).

Non c'è controllo riguardo al ritorno; ad esempio se `toString` ritorna effettivamente una stringa, o se `Symbol.toPrimitive` ritorna un numero per una `hint` “number”

L'unico obbligo: questi metodi devono ritornare un tipo primitivo, non un oggetto.

### 1 Note storiche

Per ragioni storiche, se `toString` o `valueOf` ritornassero un oggetto, non ci sarebbero errori, ma il risultato sarebbe ignorato (come se il metodo non esistesse). Questo accade perché inizialmente in JavaScript non c'era il concetto di “errore”.

Invece, `Symbol.toPrimitive` deve ritornare un tipo primitivo, altrimenti ci sarebbe un errore.

## Ulteriori conversioni

Come già sappiamo, molti operatori eseguono una conversione dei tipi, per esempio l'operatore `*`, che converte gli operandi a numeri.

Se passiamo un oggetto come argomento, ci sono due passaggi:

1. L'oggetto è convertito a primitivo (secondo le regole spiegate sopra).
2. Se il risultato primitivo non è del tipo giusto, viene convertito.

Ad esempio:

```
let obj = {
 // toString gestisce tutte le conversioni nel caso manchino gli altri metodi
 toString() {
```

```

 return "2";
 }
};

alert(obj * 2); // 4, l'oggetto viene convertito al primitivo "2", successivamente la moltiplica

```

1. La moltiplicazione `obj * 2` prima converte l'oggetto a primitivo (è una stringa, `"2"`).
2. Quindi `"2" * 2` diventa `2 * 2` (la stringa è convertita a numero).

Binary plus will concatenate strings in the same situation, as it gladly accepts a string:  
L'operatorio binario `+` concatenerebbe delle stringhe nella stessa situazione:

```

let obj = {
 toString() {
 return "2";
 }
};

alert(obj + 2); // 22 ("2" + 2), la conversione a primitivo ha restituito una stringa => concatene

```

## Riepilogo

La conversione di un oggetto a primitivo viene automaticamente effettuata da molte funzioni integrate e da operatori che si aspettano un primitivo come valore. Ce ne sono tre tipi (hint):

- `"string"` (per `alert` e altre conversioni al tipo `string`)
- `"number"` (per operazioni matematiche)
- `"default"` (alcuni operatori)

Le specifiche descrivono esplicitamente quali operatori utilizzano quali hint. Ci sono veramente pochi operatori che “non sanno quali utilizzare” e quindi scelgono quello di `"default"`. Solitamente per gli oggetti integrati l'hint `"default"` si comporta nello stesso modo di quello di tipo `"number"`, quindi nella pratica questi ultimi due sono spesso uniti.

L'algoritmo di conversione segue questi passi:

1. Chiama `obj[Symbol.toPrimitive](hint)` se il metodo esiste,
2. Altrimenti se `“hint”` è di tipo `"string"`
  - prova `obj.toString()` e `obj.valueOf()`, sempre se esiste.
3. Altrimenti se `“hint”` è di tipo `"number"` o `"default"`
  - prova `obj.valueOf()` and `obj.toString()`, sempre se esiste.

Nella pratica, spesso è sufficiente implementare solo `obj.toString()` come metodo che “cattura tutte” le conversioni e ritorna una rappresentazione dell'oggetto “interpretabile dall'uomo”, per mostrarlo o per il debugging.

Per quanto riguarda le operazioni matematiche, JavaScript non fornisce un modo per “sovrascriverle” utilizzando i metodi, quindi vengono raramente utilizzate sugli oggetti.

## Tipi di dato

Vedremo più strutture dati ed andremo più in profondità con lo studio dei tipi.

## Metodi dei tipi primitivi

JavaScript ci consente di trattare i tipi primitivi (stringhe, numeri, etc.) come se fossero oggetti.

Mette a disposizione diversi metodi per farlo, che molto presto studieremo, prima però dobbiamo capire come funzionano, perché ovviamente i tipi primitivi non sono oggetti (cercheremo quindi di fare chiarezza).

Vediamo quali sono i punti chiave che distinguono i tipi primitivi dagli oggetti.

Un primitivo

- E' un valore di tipo primitivo.
- Esistono 6 tipi primitivi: `string`, `number`, `bigint`, `boolean`, `symbol`, `null` e `undefined`.

Un oggetto

- E' in grado di memorizzare molti valori come proprietà.
- Può essere creato con `{}`, ad esempio: `{name: "John", age: 30}`. Ci sono altri tipi di oggetto in JavaScript; le funzioni ad esempio sono oggetti.

Uno dei principali vantaggi degli oggetti, è che questi possono essere utilizzati per memorizzare funzioni come loro proprietà.

```
let john = {
 name: "John",
 sayHi: function() {
 alert("Hi buddy!");
 }
};

john.sayHi(); // Hi buddy!
```

In questo esempio abbiamo creato un oggetto `john` con il metodo `sayHi`.

Esistono diversi oggetti built-in (integrati nel linguaggio), come quelli dedicati alla manipolazione delle date, degli errori, degli elementi HTML, etc. I quali possiedono diverse proprietà e metodi.

Ma tutte queste caratteristiche hanno un costo!

Gli oggetti sono più “pesanti” dei tipi primitivi. Richiedono risorse extra per supportarne il pieno funzionamento. Ma poiché queste proprietà sono fondamentali, JavaScript cerca di ottimizzarne l'utilizzo della memoria.

## Un primitivo come un oggetto

Questo è il paradosso contro cui si è scontato il creatore di JavaScript:

- Esistono molte operazioni che uno sviluppatore vorrebbe poter fare con i diversi tipi primitivi, come una stringhe o un numeri. Sarebbe molto bello poter accedere a dei metodi per questi tipi di dato.

- I tipi primitivi devono essere veloci e il più leggeri possibile.

La soluzione sembra un po' strana:

1. I primitivi rimangono primitivi. Contengono un singolo valore.
2. Il linguaggio consente di accedere alle proprietà e ai metodi di stringhe, numeri, booleani e symbol.
3. Quando questo accade, viene creato uno speciale “oggetto contenitore” che fornisce le funzionalità extra, successivamente questo verrà distrutto.

Gli “oggetti contenitore” sono diversi per ogni primitiva e sono chiamati: `String`, `Number`, `Boolean` e `Symbol`. Questi forniscono diversi insiemi di metodi.

Ad esempio, esiste un metodo `str.toUpperCase()` ↗ che ritorna la stringa con la prima lettera maiuscola.

Funziona in questo modo:

```
let str = "Hello";
alert(str.toUpperCase()); // HELLO
```

Semplice, vero? Questo è quello che accade realmente in `str.toUpperCase()`:

1. La stringa `str` è una variabile primitiva. Quindi nel momento in cui si accede ad una sua proprietà, viene creato uno speciale oggetto che memorizza il valore della stringa, e contiene metodi utili come `toUpperCase()`.
2. Questo metodo viene eseguito e ritorna una nuova stringa (mostrata da `alert`).
3. L’oggetto speciale viene distrutto, lasciando solamente la primitiva `str`.

In questo modo i primitivi possono sfruttare i vantaggi forniti dall’utilizzo dei metodi, rimanendo allo stesso tempo molto leggeri.

JavaScript cerca di ottimizzare il più possibile questo processo. In alcuni casi riesce ad evitare la creazione di oggetti inutili. Nonostante ciò, deve comunque aderire alle specifiche, quindi il comportamento deve essere simile a quello che si avrebbe con la creazione di un oggetto.

Un variabile di tipo number ha dei propri metodi, ad esempio `toFixed(n)` ↗ che arrotonda il numero alla precisione richiesta:

```
let n = 1.23456;
alert(n.toFixed(2)); // 1.23
```

Vedremo più nello specifico altri metodi nei capitoli [Numeri](#) e [Stringhe](#).

### I costruttori di `String/Number/Boolean` vengono utilizzati solo internamente

Alcuni linguaggi come Java ci consentono di creare “oggetti contenitori” esplicitamente utilizzando la sintassi `new Number(1)` o `new Boolean(false)`.

In JavaScript, è altrettanto possibile per ragioni storiche, ma è altamente **sconsigliato**. Poiché molte cose potrebbero non funzionare come dovrebbero.

Ad esempio:

```
alert(typeof 0); // "number"
alert(typeof new Number(0)); // "object"!
```

Gli oggetti valutati da un `if` sono sempre true, quindi il seguente alert verrà mostrato sempre:

```
let zero = new Number(0);

if (zero) { // zero è true, perché è un oggetto
 alert("zero is truthy?!?");
}
```

In altre parole, utilizzare le stesse funzioni con `String/Number/Boolean` senza `new` è completamente sicuro. Poiché le variabili primitive verranno convertite all’oggetto corrispondente: ad una stringa, ad un numero, o ad un bool.

Ad esempio, il seguente codice è corretto:

```
let num = Number("123"); // converte una string in number
```

### null/undefined non hanno metodi

I primitivi speciali `null` e `undefined` sono delle eccezioni. Non possiedono degli speciali “oggetti contenitori”, quindi non forniscono metodi. In questo senso, sono “molto primitivi”.

Un tentativo di accedere ad una proprietà con questi tipi di valore, ritornerà un errore:

```
alert(null.test); // errore
```

## Riepilogo

- I primitivi, ad eccezione di `null` e `undefined` forniscono molti metodi utili. Li studieremo nei prossimi capitoli.
- Formalmente, questi metodi lavorano su oggetti temporanei, JavaScript però è ottimizzato per sfruttare al meglio le risorse, non risultano quindi molto “costosi”.

## ✓ Esercizi

### Posso aggiungere una proprietà ad una stringa?

importanza: 5

Considerate il seguente codice:

```
let str = "Hello";
str.test = 5;
alert(str.test);
```

Cosa ne pensate, potrebbe funzionare? Cosa verrebbe mostrato?

[Alla soluzione](#)

## Numeri

Nella versione moderna di JavaScript ci sono due differenti tipi di numeri:

1. I numeri regolari, che vengono memorizzati nel formato a 64 bit [IEEE-754 ↗](#), conosciuti anche come “numeri in virgola mobile con doppia precisione”. Questi sono i numeri che utilizziamo la maggior parte del tempo, e sono quelli di cui parleremo in questo capitolo.
2. I BigInt, che vengono utilizzati per rappresentare numeri interi di lunghezza arbitraria. Talvolta possono tornare utili, poiché i numeri regolari non possono eccedere  $2^{53}$  od essere inferiori di  $-2^{53}$ . Poiché i BigInt vengono utilizzati in alcune aree speciali, gli abbiamo dedicato un capitolo [BigInt](#).

Quindi in questo capitolo parleremo dei numeri regolari.

### Diversi modi di scrivere un numero

Immaginiamo di dover scrivere 1 milione. La via più ovvia è:

```
let billion = 1000000000;
```

Possiamo anche usare il carattere underscore `_` come separatore:

```
let billion = 1_000_000_000;
```

Qui il carattere `_` gioca il ruolo di “zucchero sintattico”, cioè rende il numero più leggibile. Il motore JavaScript semplicemente ignorerà i caratteri `_` tra le cifre, quindi è equivalente al milione scritto sopra.

Nella vita reale però cerchiamo di evitare di scrivere lunghe file di zeri per evitare errori. E anche perché siamo pigri. Solitamente scriviamo qualcosa del tipo "1ml" per un milione o "7.3ml" 7 milioni e 300mila. Lo stesso vale per i numeri più grandi.

In JavaScript, possiamo abbreviare un numero inserendo la lettera "e" con il numero di zeri a seguire:

```
let billion = 1e9; // 1 miliardo, letteralmente: 1 e 9 zeri
alert(7.3e9); // 7.3 miliardi (equivale a 7300000000 o 7_300_000_000)
```

In altre parole, "e" moltiplica il numero 1 seguito dal numero di zeri dati.

```
1e3 = 1 * 1000 // e3 significa *1000
1.23e6 = 1.23 * 1000000 // e6 significa *1000000
```

Ora proviamo a scrivere qualcosa di molto piccolo. Ad esempio, 1 microsecondo (un milionesimo di secondo):

```
let ms = 0.000001;
```

Come prima, l'utilizzo di "e" può aiutare. Se volessimo evitare di scrivere esplicitamente tutti gli "0", potremmo scrivere:

```
let ms = 1e-6; // sei zeri alla sinistra di 1
```

Se contiamo gli zeri in 0.000001, ce ne sono 6. Quindi ovviamente 1e-6.

In altre parole, un numero negativo dopo "e" significa una divisione per 1 seguito dal numero di zeri dati:

```
// -3 divide 1 con 3 zeri
1e-3 = 1 / 1000; // 0.001

// -6 divide 1 con 6 zeri
1.23e-6 = 1.23 / 1000000; // 0.00000123
```

## Numeri esadecimali, binari e ottali

I numeri [esadecimali](#) (hexadecimal o hex) sono largamente utilizzati in JavaScript per rappresentare colori, codifiche di caratteri, e molte altre cose. Quindi ovviamente esiste un modo per abbreviarli: 0x e poi il numero.

Ad esempio:

```
alert(0xff); // 255
alert(0xFF); // 255 (equivalente)
```

I sistemi binario e ottale sono utilizzati raramente, ma sono comunque supportati con l'utilizzo dei prefissi `0b` e `0o`:

```
let a = 0b11111111; // forma binaria di 255
let b = 0o377; // forma ottale di 255

alert(a == b); // true, lo stesso numero 255 da entrambe le parti
```

Ci sono solo 3 sistemi di numerazione con questo livello di supporto. Per gli altri sistemi, dovremmo utilizzare la funzione `parseInt` (che vedremo più avanti in questo capitolo).

## toString(base)

Il metodo `num.toString(base)` ritorna una rappresentazione in stringa del numero `num` con il sistema di numerazione fornito `base`.

Ad esempio:

```
let num = 255;

alert(num.toString(16)); // ff
alert(num.toString(2)); // 11111111
```

La `base` può variare da `2` a `36`. Di default vale `10`.

Altri casi di uso comune sono:

- **base=16** si utilizza per colori in esadecimale, codifiche di caratteri, i caratteri supportati sono `0..9` o `A..F`.
- **base=2** viene utilizzato per debugging o operazioni bit a bit, i caratteri accettati sono `0` o `1`.
- **base=36** la base massima, i caratteri possono andare da `0..9` o `A..Z`. L'intero alfabeto latino viene utilizzato per rappresentare un numero. Un caso divertente di utilizzo per la base `36` è quando abbiamo bisogno che un identificatore molto lungo diventi qualcosa di più breve, come ad esempio per accorciare gli url. Possiamo semplicemente rappresentarlo in base `36`:

```
alert(123456..toString(36)); // 2n9c
```

### Due punti per chiamare un metodo

Da notare che i due punti in `123456..toString(36)` non sono un errore. Se vogliamo chiamare un metodo direttamente da un numero, come `toString` nell'esempio sopra, abbiamo bisogno di inserire due punti `..`.

Se inseriamo un solo punto: `123456.toString(36)`, otterremo un errore, perché la sintassi JavaScript implica una parte decimale a seguire del primo punto. Se invece inseriamo un ulteriore punto, allora JavaScript capirà che la parte decimale è vuota e procederà nel chiamare il metodo.

Potremmo anche scrivere `(123456).toString(36)`.

## Arrotondare

Una delle operazioni più utilizzate quando lavoriamo con i numeri è l'arrotondamento.

Ci sono diverse funzioni integrate per eseguire questa operazione:

### `Math.floor`

Arrotonda per difetto: `3.1` diventa `3`, e `-1.1` diventa `-2`.

### `Math.ceil`

Arrotonda per eccesso: `3.1` diventa `4`, e `-1.1` diventa `-1`.

### `Math.round`

Arrotonda all'intero più vicino: `3.1` diventa `3`, `3.6` diventa `4`, e `3.5` viene arrotondato anch'esso a `4`.

### `Math.trunc` (non supportato da Internet Explorer)

Rimuove tutto dopo la virgola decimale senza arrotondare: `3.1` diventa `3`, `-1.1` diventa `-1`.

Qui abbiamo una tabella che riassume le principali differenze:

	<code>Math.floor</code>	<code>Math.ceil</code>	<code>Math.round</code>	<code>Math.trunc</code>
<code>3.1</code>	<code>3</code>	<code>4</code>	<code>3</code>	<code>3</code>
<code>3.6</code>	<code>3</code>	<code>4</code>	<code>4</code>	<code>3</code>
<code>-1.1</code>	<code>-2</code>	<code>-1</code>	<code>-1</code>	<code>-1</code>
<code>-1.6</code>	<code>-2</code>	<code>-1</code>	<code>-2</code>	<code>-1</code>

Queste funzioni coprono tutti i possibili casi quando trattiamo numeri con una parte decimale. Come potremmo fare se volessimo arrotondare il numero ad `n` cifre dopo la virgola?

Ad esempio, abbiamo `1.2345` e vogliamo arrotondarlo a due cifre dopo la virgola, tenendo solo `1.23`.

Ci sono due modi per farlo:

1. Moltiplica e dividi.

Ad esempio, per arrotondare un numero alla seconda cifra decimale, possiamo moltiplicare il numero per `100`, chiamare la funzione di arrotondamento e dividerlo nuovamente.

```
let num = 1.23456;

alert(Math.round(num * 100) / 100); // 1.23456 -> 123.456 -> 123 -> 1.23
```

2. Il metodo `toFixed(n)` ↗ arrotonda il numero a `n` cifre dopo la virgola e ritorna una rappresentazione in stringa del risultato.

```
let num = 12.34;
alert(num.toFixed(1)); // "12.3"
```

Questo metodo arrotonderà per difetto o per eccesso in base al valore più vicino, similmente a `Math.round`:

```
let num = 12.36;
alert(num.toFixed(1)); // "12.4"
```

Da notare che il risultato di `toFixed` è una stringa. Se la parte decimale è più breve di quanto richiesto, verranno aggiunti degli zeri:

```
let num = 12.34;
alert(num.toFixed(5)); // "12.34000", aggiunti gli zeri per renderlo esattamente di 5 cifre
```

Possiamo convertire il risultato al tipo numerico utilizzando la somma unaria o chiamando il metodo `Number()`: `+num.toFixed(5)`.

## Calcoli imprecisi

Internamente, un numero è rappresentato in formato 64-bit [IEEE-754](#) ↗, quindi vengono utilizzati esattamente 64 bit per rappresentare un numero: 52 vengono utilizzati per rappresentare le cifre, 11 per la parte decimale, e infine 1 bit per il segno.

Se un numero è troppo grande, tale da superare i 64 bit disponibili, come ad esempio un numero potenzialmente infinito:

```
alert(1e500); // Infinity
```

Potrebbe essere poco ovvio, ma quello che accade è la perdita di precisione.

Consideriamo questo test (falso!):

```
alert(0.1 + 0.2 == 0.3); // false
```

Esatto, se provassimo a confrontare il risultato della somma tra `0.1` e `0.2` con `0.3`, otterremmo `false`.

Strano! Quale può essere il risultato se non `0.3`?

```
alert(0.1 + 0.2); // 0.30000000000000004
```

Ouch! Un confronto errato di questo tipo può generare diverse conseguenze. Immaginate di progettare un sito di e-shop in cui i visitatori aggiungono al carrello articoli da `$0.10` e `$0.20`. Poi come prezzo totale viene mostrato `$0.30000000000000004`. Questo risultato lascerebbe sorpreso chiunque.

Ma perché accade questo?

Un numero viene memorizzato nella sua forma binaria, una sequenza di “1” e “0”. I numeri con virgola come `0.1`, `0.2` che visti nella loro forma decimale sembrano semplici, sono in realtà una sequenza infinita di cifre nella forma binaria.

In altre parole, cos’è `0.1`? Vale 1 diviso 10 `1/10`, “un decimo”. Nel sistema decimale questi numeri sono facilmente rappresentabili. Prendiamo invece “un terzo”: `1/3`. Diventa un numero con infiniti decimali `0.33333(3)`.

Quindi, le divisioni per potenze di `10` funzionano molto bene nel sistema decimale, non vale lo stesso con la divisione per `3`. Per la stessa ragione, nel sistema binario le divisioni per potenze di `2` sono una garanzia, ma `1/10` diventa una sequenza infinita di cifre.

Non esiste alcun modo per rappresentare esattamente `0.1` o esattamente `0.2` usando il sistema binario, proprio come non è possibile memorizzare “un terzo” come decimale.

Il formato numerico IEEE-754 cerca di risolvere questo arrotondando al più vicino numero possibile. Questo tipo di arrotondamento non ci consente di vedere le “piccole perdite di precisione”, quindi il numero viene mostrato come `0.3`. Ma comunque è presente una perdita.

Possiamo vedere un esempio:

```
alert(0.1.toFixed(20)); // 0.1000000000000000555
```

E quando sommiamo due numeri, la loro “perdita di precisione” viene incrementata.

Questo è il motivo per cui `0.1 + 0.2` non vale esattamente `0.3`.

### Non solo JavaScript

Lo stesso problema esiste in molti altri linguaggi di programmazione.

PHP, Java, C, Perl, Ruby hanno lo stesso tipo di problema, poiché si basano sullo stesso formato numerico.

Possiamo risolvere questo problema? Certamente, ci sono diverse soluzioni:

1. Possiamo arrotondare il risultato con un metodo `toFixed(n)`:

```
let sum = 0.1 + 0.2;
```

```
alert(sum.toFixed(2)); // 0.30
```

Da notare che `toFixed` ritorna sempre una stringa. Viene così garantito che ci siano almeno due cifre dopo la virgola decimale. Questo ci torna molto utile se abbiamo un e-shopping e vogliamo mostrare `$0.30`. Per tutti gli altri casi possiamo semplicemente chiamare la conversione con l'operatore di somma unaria:

```
let sum = 0.1 + 0.2;
alert(+sum.toFixed(2)); // 0.3
```

2. Possiamo temporaneamente convertire i numeri ad interi per eseguire le operazioni e poi riconvertirli. In questo modo:

```
alert((0.1 * 10 + 0.2 * 10) / 10); // 0.3
alert((0.28 * 100 + 0.14 * 100) / 100); // 0.4200000000000001
```

Questo funziona perché quando facciamo `0.1 * 10 = 1` e `0.2 * 10 = 2` entrambi diventano interi, non vi è quindi perdita di precisione.

3. Se abbiamo a che fare con dei prezzi, la miglior soluzione rimane quella di memorizzare tutti i prezzi in centesimi, evitando quindi di utilizzare i numeri con virgola. Ma cosa succede se proviamo ad applicare uno sconto del 30%? Nella pratica, evitare completamente questo problema è difficile, in alcuni casi possono tornare utili entrambe le soluzioni viste sopra.

Quindi, l'approccio moltiplicazione/divisione riduce gli errori, ma non li elimina completamente.

Talvolta possiamo evitare le frazioni. Ad esempio se abbiamo a che fare con un negozio, allora possiamo memorizzare i prezzi in centesimi piuttosto che in euro.

### 1 La cosa divertente

Provate ad eseguire questo:

```
// Ciao! Sono un numero autoincrementante!
alert(9999999999999999); // mostra 10000000000000000
```

Questo esempio ha lo stesso problema: perdita di precisione. Per la rappresentazione di un numero sono disponibili 64bit, ne vengono utilizzati 52 per le cifre, questi potrebbero non essere sufficienti. Quindi le cifre meno significative vengono perse.

JavaScript non mostra errori in questi casi. Semplicemente fa del suo meglio per "farcì stare" il numero, anche se il formato non è "grande" abbastanza.

### Due zeri

Un'altra conseguenza divertente della rappresentazione interna è l'esistenza di due zeri: `0` e `-0`.

Questo perché il segno viene rappresentato con un solo bit, in questo modo ogni numero può essere positivo o negativo, lo stesso vale per lo zero.

Nella maggior parte dei casi questa differenza è impercettibile, poiché gli operatori sono studiati per trattarli allo stesso modo.

## Test: `isFinite` e `isNaN`

Ricordate questi due valori numerici speciali?

- `Infinity` (e `-Infinity`) è uno speciale valore che è più grande (o più piccolo) di qualsiasi altro numero.
- `NaN` rappresenta un errore.

Questi appartengono al tipo `number`, ma non sono dei numeri “normali”, esistono quindi delle funzioni dedicate per la loro verifica:

- `isNaN(value)` converte l'argomento al tipo numerico e successivamente verifica se è un `NaN`:

```
alert(isNaN(NaN)); // true
alert(isNaN("str")); // true
```

Ma abbiamo veramente bisogno di questa funzione? Non possiamo semplicemente usare il confronto `==` `NaN`? Purtroppo la risposta è no. Il valore `NaN` è unico in questo aspetto, non è uguale a niente, nemmeno a se stesso:

```
alert(NaN === NaN); // false
```

- `isFinite(value)` converte l'argomento al tipo numerico e ritorna `true` se questo è un numero diverso da `NaN/Infinity/-Infinity`:

```
alert(isFinite("15")); // true
alert(isFinite("str")); // false, perché rappresenta un valore speciale: NaN
alert(isFinite(Infinity)); // false, perché rappresenta un valore speciale: Infinity
```

In alcuni casi `isFinite` viene utilizzato per verificare se una stringa qualunque è un numero:

```
let num = +prompt("Enter a number", '');
// risulterà true se non verrà inserito Infinity, -Infinity o un NaN
alert(isFinite(num));
```

Da notare che una stringa vuota o contenente solo spazi viene trattata come `0` in qualsiasi funzione numerica, compresa `isFinite`.

### **i Confronto con `Object.is`**

Esiste uno speciale metodo integrato `Object.is` ↗ che confronta valori proprio come `==`, ma risulta molto più affidabile in due casi limite:

1. Funziona con `Nan`: `Object.is(NaN, NaN) === true`, e questo è un bene.
2. I valori `0` e `-0` sono diversi: `Object.is(0, -0) === false`, tecnicamente sarebbero uguali, però internamente vengono rappresentati con il bit di segno, che in questo caso è diverso.

In tutti gli altri casi, `Object.is(a, b)` equivale a `a === b`.

Questo metodo di confronto viene spesso utilizzato in JavaScript. Quando un algoritmo interno ha necessità di verificare che due valori siano esattamente la stessa cosa, si utilizza `Object.is` (internamente chiamato `SameValue` ↗).

## **parselnt e `parseFloat`**

Conversioni numeriche come `+` o `Number()` sono rigorose. Se il valore non è esattamente un numero, falliscono:

```
alert(+"100px"); // NaN
```

L'unica eccezione tollerata è la presenza di spazi all'inizio o alla fine della stringa, poiché vengono ignorati.

Ma nella vita reale spesso abbiamo valori in unità, come `"100px"` o `"12pt"` in CSS. In molti stati il simbolo della valuta va dopo il saldo, quindi abbiamo `"19€"` e vorremmo estrarre un valore numerico.

Questo è il motivo per cui sono stati pensati `parseInt` e `parseFloat`.

Questi metodi “leggono” numeri dalla stringa finché ne sono in grado. In caso di errore, vengono ritornati i numeri raccolti. La funzione `parseInt` ritorna un intero, analogamente `parseFloat` ritorna un numero in virgola mobile:

```
alert(parseInt('100px')); // 100
alert(parseFloat('12.5em')); // 12.5

alert(parseInt('12.3')); // 12, viene ritornata solamente la parte intera
alert(parseFloat('12.3.4')); // 12.3, il secondo punto interrompe la lettura
```

Ci possono essere situazioni in cui `parseInt/parseFloat` ritornano `NaN`. Questo accade quando non viene letta nessuna cifra:

```
alert(parseInt('a123')); // NaN, il primo simbolo interrompe la lettura
```

### **i Il secondo argomento di `parseInt(str, radix)`**

La funzione `parseInt()` possiede un secondo parametro opzionale. Questo specifica la base del sistema numerale utilizzato, quindi `parseInt` può anche analizzare stringhe di numeri esadecimales, binari e così via:

```
alert(parseInt('0xff', 16)); // 255
alert(parseInt('ff', 16)); // 255, senza 0x funziona ugualmente

alert(parseInt('2n9c', 36)); // 123456
```

## Altre funzioni matematiche

JavaScript ha un oggetto integrato [Math ↗](#) che contiene una piccola libreria di funzioni e costanti matematiche.

Un paio di esempi:

### **`Math.random()`**

Ritorna un numero casuale tra 0 e 1 (1 escluso)

```
alert(Math.random()); // 0.1234567894322
alert(Math.random()); // 0.5435252343232
alert(Math.random()); // ... (numero casuale)
```

### **`Math.max(a, b, c...)` / `Math.min(a, b, c...)`**

Ritorna il maggiore/minore fra una lista di argomenti.

```
alert(Math.max(3, 5, -10, 0, 1)); // 5
alert(Math.min(1, 2)); // 1
```

### **`Math.pow(n, power)`**

Ritorna `n` elevato alla `power`

```
alert(Math.pow(2, 10)); // 2 alla 10 = 1024
```

Ci sono molte altre funzioni e costanti nell'oggetto `Math`, incluse quelle trigonometriche, che potete trovare nella [documentazione dell'oggetto Math ↗](#).

## Riepilogo

Per scrivere numeri molto grandi:

- Accodate una "e" con il numero di zeri. Come: `123e6` che vale `123` con 6 zeri.

- Un numero negativo dopo "e" divide il numero dato per 1 seguito dal numero di zeri dati. Per numeri tipo "un millonesimo".

Per diversi sistemi numerici:

- Potete scrivere direttamente in esadecimale (`0x`), ottale (`0o`) e binario (`0b`)
- `parseInt(str, base)` analizza un numero intero con un qualsiasi sistema numerico con base:  $2 \leq \text{base} \leq 36$ .
- `num.toString(base)` converte un numero ad una stringa utilizzando il sistema numerico fornito in `base`.

Per convertire a numeri valori del tipo `12pt` e `100px`:

- Utilizzate `parseInt`/`parseFloat` per un conversione "soft", i quali provano a leggere un numero da una stringa e ritornano ciò che sono riusciti ad estrarre prima di interrompersi.

Per i numeri con la virgola:

- Arrotondate usando `Math.floor`, `Math.ceil`, `Math.trunc`, `Math.round` o `num.toFixed(precision)`.
- Ricordatevi che c'è una perdita di precisione quando operate con numeri decimali.

Altre funzioni matematiche:

- Guardate l'oggetto [Math ↗](#) in caso di necessità. La libreria non è molto ampia, ma è in grado di coprire le necessità di base.

## ✓ Esercizi

---

### Sommate i numeri forniti dall'utente

importanza: 5

Create uno script che richiede due numeri all'utente (tramite `prompt`) e successivamente mostra la somma.

[Esegui la demo](#)

P.S. Fate attenzione ai tipi.

[Alla soluzione](#)

---

### Perchè `6.35.toFixed(1) == 6.3?`

importanza: 4

Come da documentazione `Math.round` e `toFixed` arrotondano al numero più vicino: per difetto con decimali da `0..4`, per eccesso con decimali da `5..9`.

Ad esempio:

```
alert(1.35.toFixed(1)); // 1.4
```

Un altro esempio simile, perchè `6.35` viene arrotondato a `6.3`, e non a `6.4`?

```
alert(6.35.toFixed(1)); // 6.3
```

Come possiamo arrotondare correttamente `6.35`?

[Alla soluzione](#)

---

## Ripeti finché non viene inserito un numero

importanza: 5

Create una funzione `readNumber` che richiede (tramite `prompt`) all'utente un numero, quest'azione va ripetuta finché l'utente non inserisce un numero valido.

Il valore va poi restituito come numero.

L'utente potrebbe anche interrompere il processo con una linea vuota o premendo “CANCEL”. In questo caso la funzione dovrebbe ritornare `null`.

[Esegui la demo](#)

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

---

## Un ciclo infinito occasionale

importanza: 4

Questo è un ciclo infinito. Non terminerà mai. Perchè?

```
let i = 0;
while (i != 10) {
 i += 0.2;
}
```

[Alla soluzione](#)

---

## Un numero random fra min e max

importanza: 2

La funzione integrata `Math.random()` crea un valore random compreso tra `0` e `1` (1 escluso).

Scrivete la funzione `random(min, max)` per generare un numero random in virgola mobile che va da `min` a `max` (max escluso).

Esempi:

```
alert(random(1, 5)); // 1.2345623452
```

```
alert(random(1, 5)); // 3.7894332423
alert(random(1, 5)); // 4.3435234525
```

[Alla soluzione](#)

## Un intero casuale tra min e max

importanza: 2

Create una funzione `randomInteger(min, max)` che genera un numero *intero* casuale compreso tra `min` e `max` inclusi `min` e `max`.

Qualsiasi numero nell'intervallo `min..max` deve poter apparire con la stessa probabilità.

Esempi:

```
alert(randomInteger(1, 5)); // 1
alert(randomInteger(1, 5)); // 3
alert(randomInteger(1, 5)); // 5
```

Potete utilizzare la soluzione dell'[esercizio precedente](#) come base.

[Alla soluzione](#)

## Stringhe

In JavaScript, i dati di tipo testuale vengono memorizzati in stringhe. Non esiste un tipo separato per i caratteri singoli.

Il formato utilizzato per le stringhe è sempre [UTF-16 ↗](#), non viene influenzato dalla codifica della pagina.

## Apici

Ricapitoliamo i tipi di apice.

Le stringhe possono essere racchiuse tra singoli apici, doppi apici o backticks:

```
let single = 'single-quoted';
let double = "double-quoted";

let backticks = `backticks`;
```

Gli apici singoli e doppi sono essenzialmente uguali. I backticks, invece, ci consentono di includere una qualsiasi espressione all'interno della stringa, inserendola all'interno di  `${...}` :

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

```
alert(`1 + 2 = ${sum(1, 2)} `); // 1 + 2 = 3.
```

Un altro vantaggio nell'utilizzo di backticks è che consentono di dividere la stringa in più righe:

```
let guestList = `Guests:
* John
* Pete
* Mary
`;

alert(guestList); // una lista di guest, in piu righe
```

Se proviamo a utilizzare gli apici singoli o doppi allo stesso modo, otterremo un errore:

```
let guestList = "Guests: // Error: Unexpected token ILLEGAL
* John";
```

Gli apici singoli e doppi sono nati insieme al linguaggio, quando non era stato ancora messo in conto la possibilità di stringhe multilinea. Le backticks sono apparse più tardi, per questo risultano più versatili.

Le backticks ci consentono anche di specificare un “template di funzione” prima della backtick di apertura. La sintassi è: `func`string``. La funzione `func` viene chiamata automaticamente, gli viene passata la “string”, può essere così trattata dalla funzione. Potete approfondire leggendo la [documentazione ↗](#). Questo viene chiamata “funzione template”. Con questa caratteristica diventa più facile raccogliere stringhe da passare a funzioni, ma è raramente utilizzata.

## Caratteri speciali

E' comunque possibile creare stringhe multilinea con singoli apici utilizzando il “carattere nuova riga”, cioè `\n`, che significa appunto nuova riga:

```
let guestList = "Guests:\n * John\n * Pete\n * Mary";

alert(guestList); // una lista di guest multi riga
```

Ad esempio, queste due funzioni portano allo stesso risultato:

```
let str1 = "Hello\nWorld"; // due righe utilizzando il "carattere nuova riga"

// due righe utilizzando le backticks
let str2 = `Hello
World`;
```

Ci sono altri caratteri “speciali” meno comuni. Qui una lista:

Carattere	Descrizione
-----------	-------------

Carattere	Descrizione
\n	Nuova linea
\r	Ritorno a capo: non utilizzato da solo. I file di testo Windows utilizzano una combinazione di due caratteri \n\r per rappresentare il termine della riga.
\' , \"	Apici
\\\	Backslash
\t	Tab
\b , \f , \v	Backspace, Form Feed, Vertical Tab – mantenuti per retrocompatibilità, oggi non sono utilizzati.
\xXX	Carattere Unicode rappresentato dal codice esadecimale XX, esempio '\x7A' equivale a 'z'.
\uXXXX	Simbolo unicode rappresentato da codice esadecimale XXXX in codifica UTF-16, ad esempio \u00A9 – equivale a ©.
\u{X...XXXXXX} (da 1 a 6 caratteri esadecimali)	Un simbolo Unicode in codifica UTF-32. Alcuni caratteri vengono codificati da due simboli unicode, ovvero 4 byte.

Esempi di unicode:

```
alert("\u00A9"); // ©
alert("\u{20331}"); // 倍, un raro geroglifico cinese (long unicode)
alert("\u{1F60D}"); // 😊, un simbolo di faccia sorridente (long unicode)
```

Tutti i caratteri speciali iniziano con un backslash \. Che viene anche chiamato “carattere di escape”.

Dobbiamo utilizzarlo anche se abbiamo intenzione di inserire un apice all'interno della stringa.

Ad esempio:

```
alert('I'm the Walrus!'); // I'm the Walrus!
```

Avete visto che abbiamo inserito un backslash \' prima dell'apice interno, altrimenti questo avrebbe indicato la fine della stringa.

Ovviamente, questo è valido per un apice uguale a quello utilizzato in apertura. Quindi, possiamo optare per una soluzione più elegante, ad esempio i doppi apici o i backticks:

```
alert(`I'm the Walrus!`); // I'm the Walrus!
```

Da notare che il backslash \ ha l'unico scopo di aiutare JavaScript nella lettura della stringa, questo verrà poi rimosso. La stringa in memoria non avrà \. Lo avrete sicuramente notato con gli alert dei vari esempi sopra.

Ma se volessimo realmente mostrare un backslash \ dentro la stringa?

E' possibile farlo, ma dobbiamo esplicitarlo con un doppio backslash \\:

```
alert(`The backslash: \\`); // The backslash: \
```

## String length

La proprietà `length` (lunghezza) contiene la lunghezza della stringa:

```
alert(`My\n`.length); // 3
```

Da notare che `\n` è contato come unico carattere “speciale”, quindi la lunghezza risulta essere 3.

### ⚠ `length` è una proprietà

Alcune persone abituate ad altri linguaggi possono confondere al chiamata `str.length()` con `str.length`. Questo è un errore.

Infatti `str.length` è una proprietà numerica, non una funzione. Non c’è alcun bisogno delle parentesi.

## Accesso ai caratteri

Per ottenere un carattere alla posizione `pos`, si utilizzano le parentesi quadre `[pos]` oppure la chiamata al metodo `str.charAt(pos)` ↗. Il primo carattere parte dalla posizione zero:

```
let str = `Hello`;

// il primo carattere
alert(str[0]); // H
alert(str.charAt(0)); // H

// l'ultimo carattere
alert(str[str.length - 1]); // o
```

L’utilizzo delle parentesi quadre è il modo più classico per accedere ad un carattere, mentre `charAt` esiste principalmente per ragioni storiche.

L’unica differenza sta nel comportamento in casi di carattere non trovato, `[]` ritorna `undefined`, e `charAt` ritorna una stringa vuota:

```
let str = `Hello`;

alert(str[1000]); // undefined
alert(str.charAt(1000)); // '' (una stringa vuota)
```

Possiamo iterare sui caratteri utilizzando `for...of`:

```
for (let char of "Hello") {
 alert(char); // H,e,l,l,o (char diventa "H", poi "e", poi "l" etc)
```

```
}
```

## Le stringhe sono immutabili

Le stringhe in JavaScript non possono essere modificate. Risulta impossibile cambiare anche un solo carattere.

Possiamo anche provare a modificarla per vedere che non funziona:

```
let str = 'Hi';

str[0] = 'h'; // errore
alert(str[0]); // non funziona
```

Il metodo utilizzato per aggirare questo problema è creare una nuova stringa ed assegnarla a `str` sostituendo quella vecchia.

Ad esempio:

```
let str = 'Hi';

str = 'h' + str[1]; // rimpiazza str

alert(str); // hi
```

Nelle prossime sezioni vedremo ulteriori esempi.

## Cambiare il timbro delle lettere

I metodi come `toLowerCase()` ↗ e `toUpperCase()` ↗ cambiano il timbro delle lettere:

```
alert('Interface'.toUpperCase()); // INTERFACE
alert('Interface'.toLowerCase()); // interface
```

Altrimenti, possiamo agire anche su un singolo carattere:

```
alert('Interface'[0].toLowerCase()); // 'i'
```

## Cercare una sotto-stringa

Ci sono diversi modi per cercare una sotto-stringa all'interno di una stringa.

### `str.indexOf`

Il primo metodo è `str.indexOf(substr, pos)` ↗ .

Quello che fa è cercare `substr` in `str`, ad iniziare dalla posizione `pos`, e ne ritorna la posizione una volta trovata, se non trova corrispondenze ritorna `-1`.

Ad esempio:

```
let str = 'Widget with id';

alert(str.indexOf('Widget')); // 0, perché 'Widget' è stato trovato all'inizio
alert(str.indexOf('widget')); // -1, non trovato, la ricerca è case-sensitive

alert(str.indexOf("id")); // 1, "id" è stato trovato alla posizione di indice 1
```

Il secondo parametro opzionale ci consente di cercare a partire dalla posizione fornita.

Ad esempio, la prima occorrenza di `"id"` è alla posizione `1`. Per trovare la successiva occorrenza, dovremmo iniziare a cercare dalla posizione `2`:

```
let str = 'Widget with id';

alert(str.indexOf('id', 2)) // 12
```

Se siamo interessati a tutte le occorrenze, possiamo utilizzare `indexOf` in un ciclo. Ogni chiamata viene fatta a partire dalla posizione della precedente corrispondenza:

```
let str = 'As sly as a fox, as strong as an ox';

let target = 'as'; // procediamo con la ricerca

let pos = 0;
while (true) {
 let foundPos = str.indexOf(target, pos);
 if (foundPos == -1) break;

 alert(`Found at ${foundPos}`);
 pos = foundPos + 1; // continua la ricerca a partire dalla prossima posizione
}
```

Lo stesso algoritmo può essere riscritto più brevemente:

```
let str = "As sly as a fox, as strong as an ox";
let target = "as";

let pos = -1;
while ((pos = str.indexOf(target, pos + 1)) != -1) {
 alert(pos);
}
```

### **i** `str.lastIndexOf(substr, position)`

Un altro metodo simile è `str.lastIndexOf(substr, position)` ↗, che effettua la ricerca partendo dalla fine della stringa.

Elenca quindi le occorrenze in ordine inverso.

C'è solo un piccolo inconveniente dovuto all'utilizzo di `indexOf` all'interno delle espressioni `if`. Non possiamo inserirlo in un `if` in questo modo:

```
let str = "Widget with id";

if (str.indexOf("Widget")) {
 alert("We found it"); // non funziona!
}
```

L' `alert` nell'esempio sopra non viene mostrato perché `str.indexOf("Widget")` ritorna `0` (significa che è stata trovata una corrispondenza nella posizione iniziale). Ed è corretto, ma `if` considera `0` come `false`.

Quindi dovremmo verificare il `-1`, in questo modo:

```
let str = "Widget with id";

if (str.indexOf("Widget") != -1) {
 alert("We found it"); // ora funziona!
}
```

#### Il trucco del NOT bit a bit

Uno dei trucchi più utilizzati è l'operatore di **NOT bit a bit** ↗ `~`. Questo converte il numero ad un intero in 32bit (rimuovendo la parte decimale se presente) e successivamente inverte tutti i bit.

Per gli interi in 32bit la chiamata `~n` ha lo stesso risultato di `-(n+1)` (a causa del formato IEEE-754).

Ad esempio:

```
alert(~2); // -3, lo stesso di -(2+1)
alert(~1); // -2, lo stesso di -(1+1)
alert(~0); // -1, lo stesso di -(0+1)
alert(~-1); // 0, lo stesso di -(-1+1)
```

Come avete potuto osservare, `~n` vale zero solo se `n == -1`.

Quindi, il test `if (~str.indexOf(...))` è vero allora il risultato di `indexOf` non è `-1`. In altre parole, è stata trovata una corrispondenza.

Le persone lo utilizzano per abbreviare i controlli con `indexOf`:

```
let str = "Widget";

if (~str.indexOf("Widget")) {
 alert('Found it!'); // funziona
}
```

Solitamente è sconsigliato utilizzare caratteristiche del linguaggio per azioni che possono risultare poco ovvie, ma questo particolare trucco è ampiamente utilizzato, quindi è giusto conoscerlo.

Ricordatevi solo che: `if (~str.indexOf(...))` si legge come "se trovi".

Per essere precisi, numeri molto grandi vengono troncati a 32bit dall'operatore `~`, esistono altri numeri che potrebbero dare `0`, il più piccolo è `~4294967295=0`. Questo fa sì che questo tipo di controlli siano corretti solamente se una stringa non è troppo lunga.

Attualmente questo trucco lo troviamo solamente nei codici vecchi, poiché JavaScript moderno fornisce un metodo dedicato, `.includes` (vedi sotto).

### includes, startsWith, endsWith

Un metodo più moderno come `str.includes(substr, pos)` ritorna `true/false` basandosi solo sull'aver trovato in `str` la `substr`.

E' la scelta migliore se abbiamo bisogno di verificarne solo la corrispondenza, senza dover necessariamente conoscerne la posizione:

```
alert("Widget with id".includes("Widget")); // true
alert("Hello".includes("Bye")); // false
```

Il secondo argomento opzionale di `str.includes` è la posizioni da cui iniziare a cercare:

```
alert("Midget".includes("id")); // true
alert("Midget".includes("id", 3)); // false, dalla posizione 3 non c'è "id"
```

I metodi `str.startsWith` e `str.endsWith` fanno esattamente ciò che dicono i loro nomi:

```
alert("Widget".startsWith("Wid")); // true, "Widget" inizia con "Wid"
alert("Widget".endsWith("get")); // true, "Widget" finisce con "get"
```

### Estrarre una sotto-stringa

Ci sono 3 metodi in JavaScript per estrarre una sotto-stringa: `substring`, `substr` e `slice`.

`str.slice(start [, end])`

Ritorna la parte di stringa che va da `start` fino a `end` (esclusa).

Ad esempio:

```
let str = "stringify";
alert(str.slice(0, 5)); // 'strin', la sottostringa da 0 a 5 (escluso 5)
alert(str.slice(0, 1)); // 's', da 0 a 1, escluso 1, quindi solamente il carattere 0
```

Se non c'è un secondo argomento, allora `slice` si ferma alla fine della stringa:

```
let str = "stringify";
alert(str.slice(2)); // ringify, dalla seconda posizione fino alla fine
```

Sono possibili anche valori negativi per `start/end`. Questo significa che la posizione verrà contata a partire dalla fine della stringa:

```
let str = "stringify";

// incomincia dalla 4 posizione a partire da destra, e si termina alla prima a partire da destra
alert(str.slice(-4, -1)); // gif
```

### `str.substring(start [, end])`

Ritorna la parte di stringa compresa tra `start` e `end`.

E' molto simile a `slice`, ma consente di avere `start` maggiore di `end`.

Ad esempio:

```
let str = "stringify";

// questi sono identici per substring
alert(str.substring(2, 6)); // "ring"
alert(str.substring(6, 2)); // "ring"

// ...non per slice:
alert(str.slice(2, 6)); // "ring" (lo stesso)
alert(str.slice(6, 2)); // "" (una stringa vuota)
```

Argomenti negativi (a differenza di `slice`) non sono supportati, vengono trattati come `0`.

### `str.substr(start [, length])`

Ritorna la parte di stringa a partire da `start`, e della lunghezza `length` data.

Diversamente dai metodi precedenti, questo ci consente di specificare la `length` (lunghezza) piuttosto della posizione in cui terminare l'estrazione:

```
let str = "stringify";
alert(str.substr(2, 4)); // ring, dalla seconda posizione prende 4 caratteri
```

Il primo argomento può anche essere negativo come con `slice`:

```
let str = "stringify";
alert(str.substr(-4, 2)); // gi, dalla quarta posizione prende 4 caratteri
```

Ricapitoliamo questi metodi per evitare confusione:

metodo	selezione...	negativi
<code>slice(start, end)</code>	da <code>start</code> a <code>end</code> ( <code>end</code> escluso)	indici negativi ammessi
<code>substring(start, end)</code>	tra <code>start</code> e <code>end</code>	valori negativi valgono come <code>0</code>
<code>substr(start, length)</code>	da <code>start</code> per <code>length</code> caratteri	consente indice di <code>start</code> negativo

### **i** Quale scegliere?

Tutti i metodi esaminati possono portare a termine il lavoro. Formalmente, `substr` ha un piccolo inconveniente: non è descritto nelle specifiche del core JavaScript, ma in quelle di Annex B, che copre solo le caratteristiche utili nello sviluppo browser. Quindi ambienti diversi dal browser potrebbero non supportarla. Ma nella pratica viene utilizzata ovunque.

L'autore della guida si trova spesso ad utilizzare il metodo `slice`.

## Confronto tra stringhe

Come sappiamo dal capitolo [Confronti](#), le stringhe vengono confrontate carattere per carattere in ordine alfabetico.

Sebbene ci siano dei casi particolari.

1. Una lettera minuscola è sempre maggiore di una maiuscola:

```
alert('a' > 'Z'); // true
```

2. Le lettere con simboli particolari (come quelle tedesche) non vengono considerate:

```
alert('Österreich' > 'Zealand'); // true
```

Questo potrebbe portare a strani risultati se provassimo ad ordinare le città per nome. Solitamente ci si aspetta di trovare `Zealand` dopo `Österreich`.

Per capire cosa succede, dobbiamo guardare alla rappresentazione interna delle stringhe in JavaScript.

Tutte le stringhe vengono codificate utilizzando [UTF-16 ↗](#). Cioè: ogni carattere ha un suo codice numerico. Ci sono alcuni metodi che consentono di ottenere il carattere dal codice (e viceversa).

### `str.codePointAt(pos)`

Ritorna il codice per il carattere alla posizione `pos`:

```
// lettere di timbro differente possiedono codici differenti
alert("z".codePointAt(0)); // 122
alert("Z".codePointAt(0)); // 90
```

### `String.fromCodePoint(code)`

Crea un carattere preso dalla sua rappresentazione numerica `code`:

```
alert(String.fromCodePoint(90)); // Z
```

Possiamo anche aggiungere caratteri unicode tramite il loro codice utilizzando `\u` seguito dal codice esadecimale:

```
// 90 è 5a nel sistema esadecimale
alert('\u005a'); // Z
```

Ora vediamo i caratteri con il codice compreso tra 65 . . 220 (l'alfabeto latino e qualche extra) creando una stringa:

```
let str = '';
for (let i = 65; i <= 220; i++) {
 str += String.fromCodePoint(i);
}
alert(str);
// ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~"
// ¡¢£¤¥¦§©¤«¬®°±²³⁴µ¶·,¹º»¼²¾¿ÀÁÂÃÃÀÆÇÈÉÈÍÍÍÐÑòóôôö×øùûûÙ
```

Visto? Le lettere maiuscole vengono prima, poi ci sono un po di caratteri speciali e successivamente le lettere minuscole.

Ora è molto più ovvio il motivo per cui `a > Z` risulta vero.

I caratteri vengono confrontati utilizzando il loro codice numerico. Un codice maggiore significa che il carattere è maggiore. Il codice di `a` (97) è maggiore del codice di `Z` (90).

- Tutte le lettere minuscole vengono dopo quelle maiuscole perché il loro codice è maggiore.
- Alcune lettere come `Ö` vengono escluse dall'alfabeto. Il suo codice viene considerato maggiore di qualsiasi lettera compresa tra `a` e `z`.

## Confronto tra stringhe corretto

L'algoritmo più corretto da utilizzare per confrontare stringhe è più complesso di quanto si possa pensare, poiché l'alfabeto è diverso per ogni lingua. Lettere uguali possono avere posizioni diverse nei vari alfabeti.

Quindi il browser deve sapere quale lingua utilizzare nel confronto.

Fortunatamente, tutti i browser moderni (IE10 richiede una libreria esterna [Intl.js](#)) supportano lo standard internazionale [ECMA-402](#).

Questo fornisce uno speciale metodo per confrontare stringhe in lingue diverse, seguendo delle regole.

La chiamata `str.localeCompare(str2)`:

- Ritorna `1` se `str` è maggiore di `str2` seguendo le regole della lingua.
- Ritorna `-1` se `str` è minore di `str2`.
- Ritorna `0` se sono uguali.

Ad esempio:

```
alert('Österreich'.localeCompare('Zealand')); // -1
```

Questo metodo in realtà ha due argomenti opzionali specificati nella [documentazione](#), che consentono di specificare la lingua (di default viene presa dall'ambiente) e impostare delle regole

aggiuntive come il timbro delle lettere, oppure se "a" e "á" dovrebbero essere trattate ugualmente etc.

## Internamente, Unicode

### ⚠️ Apprendimento avanzato

Questa sezione andrà più in profondità riguardo le stringhe. Quello che leggerai ti potrà essere utile se hai intenzione di utilizzare emoji, simboli matematici o geroglifici.

Puoi semplicemente saltare questa sezione se non hai in programma di utilizzarle.

### Coppie surrogate

Molti simboli hanno un codice composto da 2 byte. Molte lettere di alfabeti europei, numeri, e anche molti geroglifici, hanno una rappresentazione in 2 byte.

Ma con 2 byte sono consentite solamente 65536 combinazioni, non sono comunque sufficienti per ogni tipo di simbolo possibile. Quindi molti simboli vengono codificati con una coppia di 2 byte chiamati "coppia surrogata".

La lunghezza di questi simboli è 2 :

```
alert('χ'.length); // 2, X matematica
alert('☺'.length); // 2, faccia con lacrime di felicità
alert('𩷶'.length); // 2, un raro geroglifico cinese
```

Da notare che le coppie surrogate non esistevano al momento della creazione di JavaScript, non vengono quindi processate correttamente dal linguaggio!

In realtà abbiamo un solo simbolo in ogni stringa sopra, ma la `length` vale 2.

`String.fromCodePoint` e `str.codePointAt` sono dei metodi utilizzati per lavorare con le coppie surrogate. Sono apparsi di recente nel linguaggio. Prima di loro, esisteva solo `String.fromCharCode ↗` e `str.charCodeAt ↗`. Sono esattamente la stessa cosa di `fromCodePoint/codePointAt`, semplicemente non funzionano con le coppie surrogate.

Però cercare di ottenere un simbolo può essere difficile, poiché una coppia surrogata viene trattata come due caratteri:

```
alert('χ'[0]); // uno strano simbolo...
alert('χ'[1]); // ...parte di una coppia surrogata
```

Da notare che un pezzo di una coppia surrogata non ha alcun senso senza l'altro. Quindi nell'esempio sopra verrà mostrata "spazzatura".

Tecnicamente, le coppie surrogate sono decifrabili anche per i loro codici: se il primo carattere ha un intervallo di codice di `0xd800..0xdbff`. Allora il successivo carattere (seconda parte) deve avere l'intervallo `0xdc00..0xffff`. Questi intervalli sono riservati esclusivamente per coppie surrogate definite dallo standard.

Nell'esempio sopra:

```
// charCodeAt non è consapevole delle coppie surrogate, quindi fornisce i codici delle due parti

alert('χ'.charCodeAt(0).toString(16)); // d835, tra 0xd800 e 0xdbff
alert('χ'.charCodeAt(1).toString(16)); // dc83, tra 0xdc00 e 0xffff
```

Nel capitolo [Iteratori](#) troverete molti altri modi per operare con le coppie surrogate. Ci sono anche delle librerie dedicate, ma nulla di abbastanza completo da meritare di essere menzionato.

## Lettere speciali e normalizzazione

In molte lingue ci sono lettere composte da un carattere di base completato da un simbolo che può stare sopra/sotto.

Ad esempio, la lettera `a` può essere il carattere di base per: `àáâääää`. Molti dei caratteri “composti” hanno una loro rappresentazione nella tabella UTF-16. Però non tutte, poiché le combinazioni possibili sono veramente molte.

Per supportare composizioni arbitrarie, UTF-16 ci consente di utilizzare diversi caratteri unicode. Un carattere di base ed uno o più “simboli” con cui “decorare” il carattere di base.

Ad esempio, se abbiamo `S` con uno speciale “punto sopra” (codice `\u0307`), viene mostrato `Ś`.

```
alert('S\u0307'); // Ś
```

Se abbiamo bisogno di un ulteriore segno sopra la lettera (o sotto) – nessun problema, è sufficiente aggiungere il simbolo necessario.

Ad esempio, se vogliamo aggiungere un “punto sotto” (codice `\u0323`), allora otterremo una “S con due punti, sopra e sotto”: `Ś̄`.

Ad esempio:

```
alert('S\u0307\u0323'); // Ś̄
```

Questo consente una grande flessibilità, ma crea anche un potenziale problema: due caratteri potrebbero sembrare uguali, ma differire per la loro composizione di codici unicode.

Ad esempio:

```
alert(s1 == s2); // false, nonostante i due caratteri sembrino identici (?!)
```

Per risolvere questo, esiste un algoritmo di “normalizzazione unicode” che porta ogni stringa alla forma “normale”.

Questo algoritmo viene implementato da [str.normalize\(\)](#).

```
alert("S\u0307\u0323".normalize() == "S\u0323\u0307".normalize()); // true
```

E’ divertente notare che nella nostra situazione `normalize()` trasforma una sequenza di 3 caratteri in una che ne contiene solo uno: `\u1e68` (S con due punti).

```
alert("S\u0307\u0323".normalize().length); // 1
alert("S\u0307\u0323".normalize() == "\u1e68"); // true
```

In realtà, non è sempre così. La ragione è che il simbolo ₧ è “abbastanza comune”, quindi la tabella UTF-16 lo contiene già.

Se volete approfondire il tema della normalizzazione e le sue varianti – vengono descritte nell'appendice dello standard Unicode: [Unicode Normalization Forms ↗](#), nella pratica le informazioni fornite in questa sezione, ti saranno sufficienti.

## Riepilogo

- Ci sono 3 tipi di apici. Le backticks consentono stringhe multi-linea ed espressioni integrate.
- Le stringhe in JavaScript vengono codificate usando UTF-16.
- Possiamo utilizzare caratteri speciali come \n ed inserire lettere tramite il codice unicode \u.....
- Per ottenere un carattere, si utilizza: [] .
- Per ottenere una sotto-stringa, si utilizza: slice o substring .
- Per cambiare il timbro delle lettere di una stringa si utilizza: toLowerCase/toUpperCase .
- Per cercare una sotto-stringa, usate: indexOf , o includes/startsWith/endsWith per controlli semplici.
- Per confrontare stringhe seguendo le regole della lingua, usate: localeCompare , altrimenti verranno comparate in base al codice del singolo carattere.

Ci sono molti altri metodi utili per operare con le stringhe:

- str.trim() – rimuove gli spazi all'inizio e alla fine della stringa.
- str.repeat(n) – ripete la stringa n volte.
- ...e molto altro. Guarda il [manuale ↗](#) per maggiori dettagli.

Le stringhe possiedono anche metodi per eseguire ricerche/rimpiazzi con le regular expression. Ma l'argomento merita un capitolo separato, quindi ci ritorneremo più avanti, [Regular expressions](#).

## ✓ Esercizi

### Prima lettera maiuscola

importanza: 5

Scrivete una funzione ucFirst(str) che ritorni la stringa str con la prima lettera maiuscola, ad esempio:

```
ucFirst("john") == "John";
```

Apri una sandbox con i test. ↗

[Alla soluzione](#)

## Controllo spam

importanza: 5

Scrivete una funzione `checkSpam(str)` che ritorna `true` se `str` contiene 'viagra' o 'XXX', altrimenti false.

La funzione non deve essere sensibile al timbro delle lettere (quindi lettere maiuscole e minuscole vengono trattate allo stesso modo):

```
checkSpam('buy ViAgRA now') == true
checkSpam('free xxxx') == true
checkSpam("innocent rabbit") == false
```

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

## Troncate il testo

importanza: 5

Create una funzione `truncate(str, maxlen)` che controlla la lunghezza di `str` e, se questa eccede `maxlength` – rimpiazzate la fine di `str` con il carattere `"..."`, in modo tale da ottenere una lunghezza pari a `maxlength`.

Come risultato la funzione dovrebbe troncare la stringa (se ce n'è bisogno).

Ad esempio:

```
truncate("What I'd like to tell on this topic is:", 20) = "What I'd like to te..."
truncate("Hi everyone!", 20) = "Hi everyone!"
```

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

## Estraete il denaro

importanza: 4

Abbiamo un costo pari a `"$120"`. Dove: il simbolo del dollaro viene per primo, poi segue il numero.

Create una funzione `extractCurrencyValue(str)` in grado di estrarre il valore numerico da una stringa, e che successivamente lo ritorni.

Un esempio:

```
alert(extractCurrencyValue('$120') === 120); // true
```

Apri una sandbox con i test. ↗

Alla soluzione

## Array

Gli oggetti consentono la memorizzazione di una collezione di valori associati alle rispettive chiavi.

Ma spesso abbiamo bisogno di una *collezione ordinata*, dove abbiamo un primo, un secondo, un terzo elemento e così via. Ad esempio, abbiamo bisogno di memorizzare una lista di cose: utenti, beni, elementi HTML etc.

Non è conveniente utilizzare un oggetto per questo tipo di lavori, poiché non avremmo alcun metodo per gestire l'ordine degli elementi. Non possiamo inserire una nuova proprietà “tra” due già esistenti. Gli oggetti non sono stati pensati per questo scopo.

Esistono delle speciali strutture dati chiamate `Array`, che consentono la memorizzazione di collezioni ordinate.

## Dichiarazione

Ci sono due differenti sintassi per la creazioni di un array vuoto:

```
let arr = new Array();
let arr = [];
```

Nella maggioranza dei casi, la seconda sintassi è quella preferita. Possiamo già fornire degli elementi da inserire, all'interno delle parentesi quadre:

```
let fruits = ["Apple", "Orange", "Plum"];
```

Gli elementi di un array sono numerati a partire dallo zero.

Possiamo ottenere un elemento tramite il suo numero di indice:

```
let fruits = ["Apple", "Orange", "Plum"];

alert(fruits[0]); // Apple
alert(fruits[1]); // Orange
alert(fruits[2]); // Plum
```

Possiamo rimpiazzare un elemento:

```
fruits[2] = 'Pear'; // ora ["Apple", "Orange", "Pear"]
```

...o aggiungerne uno nuovo:

```
fruits[3] = 'Lemon'; // ora ["Apple", "Orange", "Pear", "Lemon"]
```

Il numero degli elementi dell'array è `length`:

```
let fruits = ["Apple", "Orange", "Plum"];
alert(fruits.length); // 3
```

Possiamo anche utilizzare `alert` per mostrare l'intero array.

```
let fruits = ["Apple", "Orange", "Plum"];
alert(fruits); // Apple,Orange,Plum
```

Un array può memorizzare elementi di qualsiasi tipo.

Ad esempio:

```
// insieme di valori
let arr = ['Apple', { name: 'John' }, true, function() { alert('hello'); }];

// prende l'oggetto all'indice 1 e ne mostra il nome
alert(arr[1].name); // John

// prende la funzione all'indice 3 e la esegue
arr[3](); // hello
```

### **i** Virgola pendente

Gli array, proprio come gli oggetti, possono terminare con una virgola:

```
let fruits = [
 "Apple",
 "Orange",
 "Plum",
];
```

La “virgola pendente” rende più semplice inserire/rimuovere elementi, perché tutte le linee seguono la stessa struttura.

## I metodi `pop/push, shift/unshift`

Una [queue ↗](#) (coda) è una delle più comuni applicazioni di un array. In ambito informatico, questa è una collezione ordinata che consente due operazioni:

- `push` inserisce un elemento in coda.

- `shift` per estrarre un elemento dall'inizio, e scorrere in avanti la coda, di modo che il secondo elemento diventa il primo.



Gli array supportano entrambe le operazioni.

Nella pratica non è strano incontrare questo “tipo” di array. Ad esempio una coda di messaggi che devono essere mostrati sullo schermo.

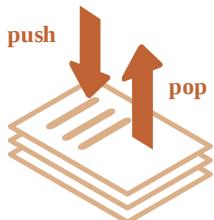
Esiste un altro caso d'uso degli array – la struttura dati chiamata [stack](#).

Questa supporta due operazioni:

- `push` aggiunge un elemento in coda.
- `pop` estrae un elemento dalla coda.

Quindi gli elementi vengono sempre aggiunti o presi dalla “fine”.

Uno stack viene spesso illustrato come un pacco di carte: le nuove carte vengono aggiunte sempre in cima o da lì estratte:



Negli stack, l'ultimo elemento inserito viene prelevato per primo. Questo comportamento viene definito LIFO (Last-In-First-Out). Nel caso delle code, il comportamento viene chiamato FIFO (First-In-First-Out).

Gli array in JavaScript possono implementare sia una queue che uno stack. C'è la possibilità di aggiungere/rimuovere elementi sia in cima che in coda.

In informatica questa struttura dati si chiama [deque](#).

### **Metodi che operano sulla coda di un array:**

#### **pop**

Estrae l'ultimo elemento dell'array e lo ritorna:

```
let fruits = ["Apple", "Orange", "Pear"];
alert(fruits.pop()); // rimuove "Pear" e lo ritorna con alert
alert(fruits); // Apple, Orange
```

#### **push**

Inserisce l'elemento in coda all'array:

```
let fruits = ["Apple", "Orange"];
fruits.push("Pear");
alert(fruits); // Apple, Orange, Pear
```

La chiamata `fruits.push(...)` è equivalente a `fruits[fruits.length] = ...`.

### Metodi che operano sull'inizio dell'array:

#### shift

Estrae il primo elemento dell'array e lo ritorna:

```
let fruits = ["Apple", "Orange", "Pear"];
alert(fruits.shift()); // rimuove Apple e lo ritorna con alert
alert(fruits); // Orange, Pear
```

#### unshift

Aggiunge l'elemento alla testa dell'array:

```
let fruits = ["Orange", "Pear"];
fruits.unshift('Apple');
alert(fruits); // Apple, Orange, Pear
```

I metodi `push` e `unshift` possono aggiungere anche più elementi in una volta sola:

```
let fruits = ["Apple"];
fruits.push("Orange", "Peach");
fruits.unshift("Pineapple", "Lemon");

// ["Pineapple", "Lemon", "Apple", "Orange", "Peach"]
alert(fruits);
```

## Internamente

Un array è uno speciale tipo di oggetto. Le parentesi quadre utilizzate per accedere alla proprietà `arr[0]` derivano dalla sintassi utilizzata con gli oggetti. Questo equivale a `obj[key]`, dove `arr` è l'oggetto, mentre i numeri vengono utilizzati come chiavi.

Inoltre estendono gli oggetti fornendo speciali metodi per operare ordinatamente su collezioni di dati, e contengono la proprietà `length`. Ma internamente rimane sempre un oggetto.

Ricordate, ci sono solo 7 tipi di base in JavaScript. Gli array sono oggetti e si comportano come tali.

Ad esempio, vengono copiati per riferimento:

```
let fruits = ["Banana"]

let arr = fruits; // copia per riferimento (due variabili fanno riferimento allo stesso array)

alert(arr === fruits); // true

arr.push("Pear"); // modifica l'array per riferimento

alert(fruits); // Banana, Pear - ora sono 2 elementi
```

... Ma ciò che li rende realmente speciali è la loro rappresentazione interna. Il motore prova a memorizzare gli elementi in aree di memoria contigue, uno dopo l'altro, proprio come nelle illustrazioni di questo capitolo, e ci sono anche altre ottimizzazioni per rendere gli array molto veloci.

Se iniziamo a trattare gli array come oggetti ordinari tutte le ottimizzazioni vengono meno.

Ad esempio, tecnicamente possiamo fare:

```
let fruits = []; // crea una array

fruits[9999] = 5; // assegna una proprietà con indice maggiore della sua lunghezza

fruits.age = 25; // crea una proprietà con un nome a scelta
```

Questo è possibile, perché gli array sono comunque degli oggetti. Possiamo aggiungervi qualsiasi proprietà.

Ma l'*engine* si accorgerà che stiamo utilizzando gli array come comuni oggetti. Le ottimizzazioni specifiche per gli array non sono studiate per questi casi, e verranno quindi disattivate.

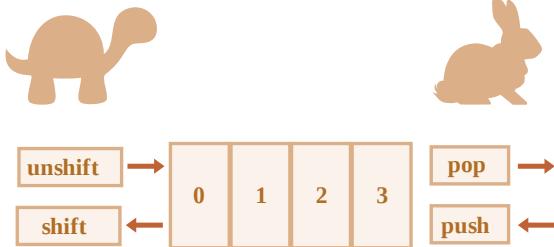
I modi per usare incorrettamente un array:

- Aggiungere una proprietà non numerica, come `arr.test = 5`.
- Creare buchi: aggiungendo `arr[0]` e poi `arr[1000]` (lasciando spazio vuoto tra di loro).
- Riempire l'array nell'ordine inverso, ad esempio `arr[1000], arr[999]`.

E' molto conveniente pensare agli array come delle speciali strutture utili a lavorare con *dati ordinati*. Forniscono speciali metodi utili a questo scopo, inoltre sono attentamente ottimizzati dal motore JavaScript per lavorare con dati ordinati e memorizzati in posizioni contigue. Quindi se doveste aver bisogno di utilizzare una proprietà con una chiave arbitraria, molto probabilmente un oggetto soddisferà le vostre necessità.

## Performance

I metodi `push/pop` vengono eseguiti rapidamente, mentre `shift/unshift` sono più lenti.



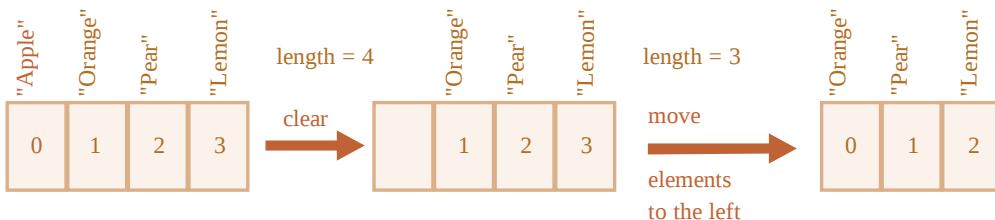
Perché è più veloce eseguire operazioni sulla coda degli array rispetto a quelle sulla testa?  
Andiamo a vedere cosa accade durante l'esecuzione:

```
fruits.shift(); // prende 1 elemento dall'inizio
```

Non è sufficiente prelevare e rimuovere l'elemento con l'indice 0. Gli altri elementi dovranno essere rinumerati.

L'operazione di `shift` deve seguire 3 passi:

1. Rimuovere l'elemento con indice 0.
2. Spostare tutti gli elementi a sinistra, rinumerare gli indici da 1 a 0, da 2 a 1 e così via.
3. Aggiornare la proprietà `length`.

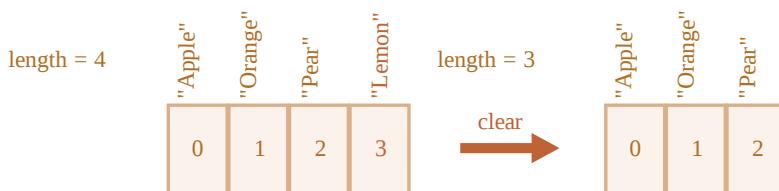


**Maggiore sarà il numero di elementi, maggiore sarà il tempo richiesto, e maggiori saranno il numero di operazioni in memoria.**

Una cosa simile accade con `unshift`: per aggiungere un elemento in testa all'array, abbiamo prima bisogno di spostare tutti gli elementi a destra e aggiornare gli indici. Invece con `push/pop`? Non richiedono lo spostamento di nulla in memoria. Per poter prelevare un elemento dalla coda, il metodo `pop` pulisce l'indirizzo e decremente `length`.

Le azioni eseguite da `pop`:

```
fruits.pop(); // prende 1 elemento dalla fine
```



**Il metodo `pop` non richiede spostamenti, perché ogni elemento mantiene il suo indice. Questo è il motivo per cui risulta essere un'operazione molto veloce.**

Una cosa simile accade con il metodo `push`.

## Cicli

Uno dei modi più vecchi per eseguire cicli sugli elementi di un array è il `for` utilizzando gli indici:

```
let arr = ["Apple", "Orange", "Pear"];

for (let i = 0; i < arr.length; i++) {
 alert(arr[i]);
}
```

Per gli array c'è un'altra forma di ciclo, `for .. of`:

```
let fruits = ["Apple", "Orange", "Plum"];

// itera sugli elementi dell'array
for (let fruit of fruits) {
 alert(fruit);
}
```

Il ciclo `for .. of` non fornisce il numero d'indice dell'elemento corrente, solo il suo valore; in molte situazioni questo è più che sufficiente. È più breve.

Tecnicamente, poiché gli array sono oggetti, è anche possibile utilizzare `for .. in`:

```
let arr = ["Apple", "Orange", "Pear"];

for (let key in arr) {
 alert(arr[key]); // Apple, Orange, Pear
}
```

Non è comunque un'ottima idea. Si possono verificare diversi errori:

1. Il ciclo `for .. in` itera su *tutte le proprietà*, non solo su quelle numeriche.

Ci sono anche degli oggetti chiamati “array-like” (simili ad array), nei browser e in altri ambienti, che *assomigliano ad array*. Infatti come proprietà possiedono `length` e degli indici, ma allo stesso tempo contengono proprietà e metodi di tipo non numerico, di cui solitamente non abbiamo bisogno. Il ciclo `for .. in` li passerà tutti. Quindi se stiamo utilizzando degli oggetti array-like, questi “extra” potrebbero rivelarsi un problema.

2. Il ciclo `for .. in` è ottimizzato per gli oggetti generici, non gli array, e può risultare quindi 10-100 volte più lento. Ovviamente rimane comunque un'operazione molto veloce. Può essere un problema solo in caso si verifichino ingorghi.

Generalmente, non dovremmo utilizzare `for .. in` per gli array.

## Una parola riguardo "length"

La proprietà `length` si aggiorna automaticamente ad ogni modifica. Volendo essere precisi non ne rappresenta la lunghezza, ma l'ultimo indice numerico più uno.

Ad esempio, un singolo elemento con un indice molto alto fornisce un altrettanto grande lunghezza:

```
let fruits = [];
fruits[123] = "Apple";

alert(fruits.length); // 124
```

Ovviamente questo non è il modo corretto di utilizzare un array.

Un'altra cosa interessante riguardo la proprietà `length` è che è sovrascrivibile.

Se provassimo ad incrementarla manualmente, non accadrebbe nulla di interessante. Se invece la decrementiamo, l'array verrà troncato. Il processo è irreversibile. Vediamo un esempio:

```
let arr = [1, 2, 3, 4, 5];

arr.length = 2; // tronca a 2 elementi
alert(arr); // [1, 2]

arr.length = 5; // ritorna alla lunghezza precedente
alert(arr[3]); // undefined: i valori non vengono ritornati
```

Quindi il modo più semplice per ripulire un array è: `arr.length = 0;`.

## `new Array()`

C'è un ulteriore sintassi per creare un array:

```
let arr = new Array("Apple", "Pear", "etc");
```

Viene utilizzata raramente, le parentesi `[]` risultano più brevi. Anche se c'è una caratteristica interessante che va osservata.

Se utilizziamo `new Array` con un solo argomento di tipo numerico, allora verrà creato un array vuoto, *ma con lunghezza data*.

Quindi vediamo come ci si potrebbe sparare da soli al piede:

```
let arr = new Array(2); // creerà un array [2] ?

alert(arr[0]); // undefined! nessun elemento.

alert(arr.length); // length 2
```

Nel codice sopra, `new Array(number)` ha tutti gli elementi `undefined`.

Per evitare queste spiacerevoli sorprese, solitamente si utilizzano le parentesi quadre, a meno di non sapere davvero che cosa si sta facendo.

## Array multi-dimensionali

Gli array possono contenere oggetti che sono a loro volta array. Possiamo quindi utilizzare questa proprietà per creare array multi-dimensional, ad esempio per memorizzare matrici:

```
let matrix = [
 [1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]
];

alert(matrix[1][1]); // l'elemento centrale
```

## toString

Gli array hanno una propria implementazione del metodo `toString`, il quale ritorna la lista degli elementi separati da una virgola.

Ad esempio:

```
let arr = [1, 2, 3];

alert(arr); // 1,2,3
alert(String(arr) === '1,2,3'); // true
```

Proviamo anche:

```
alert([] + 1); // "1"
alert([1] + 1); // "11"
alert([1,2] + 1); // "1,21"
```

Gli array non possiedono `Symbol.toPrimitive`, e nemmeno `valueOf`; implementano solamente la conversione `toString`, quindi `[]` diventa una stringa vuota, `[1]` diventa `"1"` e `[1,2]` diventa `"1,2"`.

Quando l'operatore di somma binaria `+"` aggiunge qualcosa ad una stringa, converte tutto a stringa, quindi l'esempio di prima sarà equivalente a:

```
alert("" + 1); // "1"
alert("1" + 1); // "11"
alert("1,2" + 1); // "1,21"
```

## Non confrontate gli array con ==

In JavaScript gli array, a differenza di altri linguaggi di programmazione, non dovrebbero essere confrontati con l'operatore `==`.

Questo operatore non offre alcun tipo di trattamento speciale per gli array: li considera come oggetti.

Ricordando velocemente le regole:

- Due oggetti sono uguali con `==` solamente se fanno riferimento allo stesso oggetto.
- Se uno dei due argomenti forniti all'operatore `==` è un oggetto, e l'altro è un tipo primitivo, allora l'oggetto viene convertito a primitivo, come spiegato nel capitolo [Conversione da oggetto a primitivi](#).
- ...Con l'eccezione di `null` e `undefined` che sono uguali solamente tra di loro.

Il confronto stretto con l'operatore `===` è ancora più semplice, poiché non converte i tipi.

Quindi, se confrontiamo array con `==`, non saranno mai equivalenti, a meno chè non confrontiamo due variabili che fanno riferimento allo stesso array.

Ad esempio:

```
alert([] == []); // false
alert([0] == [0]); // false
```

Questi array sono tecnicamente oggetti differenti. Quindi non si equivalgono. L'operatore `==` non effettua il confronto elemento per elemento.

Anche il confronto con tipi primitivi potrebbe dare risultati strani:

```
alert(0 == []); // true
alert('0' == []); // false
```

Qui, in entrambi i casi, stiamo confrontando un tipo primitivo con un array. Quindi l'array `[]` viene convertito in tipo primitivo per effettuare il confronto e diventa una stringa vuota `''`.

Successivamente il processo di confronto procede come descritto nel capitolo [Conversione di tipi](#):

```
// dopo averlo convertito, l'array [] equivale a ''
alert(0 == ''); // true, poiché '' viene convertito nel numero 0

alert('0' == ''); // false, nessuna conversione di tipo, sono stringhe differenti
```

Quindi, come possiamo confrontare gli array?

Molto semplice: non utilizzando l'operatore `==`. Invece, vanno confrontati con un ciclo che confronta ogni elemento dei due array, oppure utilizzando uno dei metodi di iterazione che vedremo nel prossimo capitolo.

## Riepilogo

Gli array sono uno speciale tipo di oggetto, studiati per immagazzinare e gestire collezioni ordinate di dati.

- La dichiarazione:

```
// parentesi quadre (usuale)
let arr = [item1, item2...];

// new Array (eccezionalmente raro)
let arr = new Array(item1, item2...);
```

La chiamata a `new Array(number)` crea un array con la lunghezza data, ma senza elementi.

- La proprietà `length` è la lunghezza dell'array; in realtà, per essere precisi, contiene l'indice dell'ultimo elemento più uno. Questo valore viene aggiornato automaticamente.
- Se decrementiamo manualmente `length`, l'array viene troncato.

Possiamo eseguire sugli arrays le seguenti operazioni:

- `push(...items)` aggiunge `items` in coda.
- `pop()` rimuove un elemento dalla coda e lo ritorna.
- `shift()` rimuove un elemento dalla testa e lo ritorna.
- `unshift(...items)` aggiunge un elemento alla testa.

Per iterare sugli elementi di un array:

- `for (let i = 0; i < arr.length; i++)` – il più veloce, compatibile con i vecchi browsers.
- `for (let item of arr)` – la moderna sintassi, solo per gli elementi.
- `for (let i in arr)` – da non usare.

Per confrontare gli array, non utilizzate l'operatore `==` (lo stesso vale per `>`, `<` ecc), poiché non riservano alcun trattamento speciale per gli array. Li trattano come degli oggetti comuni, e solitamente non è quello che vogliamo.

Piuttosto si possono utilizzare i cicli come `for ... of` per confrontare ogni elemento dei due array.

Continueremo con lo studio degli array e di altri metodi per aggiungere, rimuovere, estrarre elementi ed ordinarli, nel prossimo capitolo [Metodi per gli array](#).

## ✓ Esercizi

### L'array è stato copiato?

importanza: 3

Cosa mostrerà il codice sotto?

```
let fruits = ["Apples", "Pear", "Orange"];
```

```
// inserisci un nuovo elemento dentro a "copy"
let shoppingCart = fruits;
shoppingCart.push("Banana");

// che cosa c'è in fruits?
alert(fruits.length); // ?
```

[Alla soluzione](#)

## Operazioni sugli array.

importanza: 5

Proviamo 5 operazioni su un array.

1. Create un array `styles` con gli elementi “Jazz” e “Blues”.
2. Aggiungete “Rock-n-Roll” in coda.
3. Rimpiazzate l’elemento al centro con “Classics”. Il codice che utilizzerete per trovare il centro dovrebbe funzionare con qualsiasi array, anche di lunghezza dispari.
4. Prelevate il primo elemento dell’array e mostratelo.
5. Aggiungete in testa `Rap` e `Reggae`.

Le evoluzioni dell’array:

```
Jazz, Blues
Jazz, Blues, Rock-n-Roll
Jazz, Classics, Rock-n-Roll
Classics, Rock-n-Roll
Rap, Reggae, Classics, Rock-n-Roll
```

[Alla soluzione](#)

## Chiamata di funzione

importanza: 5

Qual è il risultato? Perché?

```
let arr = ["a", "b"];

arr.push(function() {
 alert(this);
})

arr[2](); // ?
```

[Alla soluzione](#)

## Somma dei numeri inseriti

importanza: 4

Scrivete una funzione `sumInput()` che:

- Richiede all'utente dei valori tramite `prompt` e memorizza i valori in un array.
- Termina se l'utente inserisce un valore non numerico, una stringa vuota, o preme "Cancel".
- Calcola e ritorna la somma degli elementi dell'array.

P.S. Lo `0` è un numero valido, non deve interrompere l'input.

[Esegui la demo](#)

[Alla soluzione](#)

## Il sub-array massimo

importanza: 2

Come input si ha un array di numeri, ad esempio `arr = [1, -2, 3, 4, -9, 6]`.

Il compito è: trovate il sub-array contiguo di `arr` con la massima somma degli elementi.

Scrivete la funzione `getMaxSubSum(arr)` che ritorna quella somma.

Ad esempio:

```
getMaxSubSum([-1, 2, 3, -9]) == 5 // (la somma degli elementi selezionati)
getMaxSubSum([2, -1, 2, 3, -9]) == 6
getMaxSubSum([-1, 2, 3, -9, 11]) == 11
getMaxSubSum([-2, -1, 1, 2]) == 3
getMaxSubSum([100, -9, 2, -3, 5]) == 100
getMaxSubSum([1, 2, 3]) == 6 // (include tutti)
```

Se tutti gli elementi sono negativi, non prendiamo nulla (il sotto-array è vuoto), quindi la somma è zero:

```
getMaxSubSum([-1, -2, -3]) = 0
```

Provate a pensare ad una soluzione rapida: [O\(n<sup>2</sup>\)](#) o addirittura O(n) se riuscite.

[Apri una sandbox con i test.](#)

[Alla soluzione](#)

## Metodi per gli array

Gli array forniscono una gran quantità di metodi. Per rendere le cose più semplici, in questo capitolo li abbiamo divisi per gruppi.

### Aggiungere/rimuovere elementi

Conosciamo già i metodi che consentono di aggiungere e rimuovere elementi:

- `arr.push(...items)` – aggiunge un elemento in coda,
- `arr.pop()` – estrae un elemento dalla coda,
- `arr.shift()` – estrae un elemento dalla testa,
- `arr.unshift(...items)` – aggiunge un elemento in testa.

Vediamone altri.

## splice

Come cancellare un elemento dall'array?

Gli array sono oggetti, quindi possiamo provare ad utilizzare `delete`:

```
let arr = ["I", "go", "home"];

delete arr[1]; // rimuove "go"

alert(arr[1]); // undefined

// ora arr = ["I", , "home"];
alert(arr.length); // 3
```

L'elemento viene rimosso, ma l'array ha ancora 3 elementi, possiamo vederlo tramite `arr.length == 3`.

Non è una sorpresa, perché `delete obj.key` rimuove un valore dalla `key`. Questo è tutto quello che fa. Può andare bene per gli oggetti. Con gli array vorremmo che il resto degli elementi scalassero, andando ad occupare il posto che si è liberato. Per questo ci aspetteremmo di avere un array più corto.

Per questo scopo sono stati sviluppati dei metodi dedicati.

Il metodo `arr.splice ↗` è un coltellino svizzero per array. Può fare qualsiasi cosa: aggiungere e rimuovere elementi, ovunque.

La sintassi è:

```
arr.splice(start[, deleteCount, elem1, ..., elemN])
```

Modifica l'array partendo dall'indice `start`; rimuove `deleteCount` elementi ed inserisce `elem1, ..., elemN`. Infine ritorna un array contenente gli elementi rimossi.

Questo metodo è facile da capire tramite esempi.

Proviamo ad eliminare degli elementi:

```
let arr = ["I", "study", "JavaScript"];

arr.splice(1, 1); // a partire da indice 1 rimuove 1 elemento

alert(arr); // ["I", "JavaScript"]
```

Facile, vero? Ha rimosso 1 elemento, a partire dall'elemento 1.

Nel prossimo esempio, rimuoviamo 3 elementi e li rimpiazziamo con altri due:

```
let arr = ["I", "study", "JavaScript", "right", "now"];

// rimuove i primi 3 elementi e li rimpiazza con altri
arr.splice(0, 3, "Let's", "dance");

alert(arr) // ora ["Let's", "dance", "right", "now"]
```

Possiamo vedere l'array ritornato da `splice` contenente gli elementi rimossi:

```
let arr = ["I", "study", "JavaScript", "right", "now"];

// rimuove i primi 2 elementi
let removed = arr.splice(0, 2);

alert(removed); // "I", "study" <-- array di elementi rimossi
```

Il metodo `splice` è anche in grado di inserire elementi senza alcuna rimozione. Per ottenere questo dobbiamo impostare `deleteCount` a 0:

```
let arr = ["I", "study", "JavaScript"];

// da indice 2
// ne rimuove 0
// poi inserisce "complex" e "language"
arr.splice(2, 0, "complex", "language");

alert(arr); // "I", "study", "complex", "language", "JavaScript"
```

### **i** Sono permessi indici negativi

In questo come in altri metodi dedicati agli array, sono permessi indici negativi. Essi specificano la posizione dalla fine dell'array, come:

```
let arr = [1, 2, 5];

// dall'indice -1 (un passo dalla fine)
// cancella 0 elementi,
// poi inserisce 3 e 4
arr.splice(-1, 0, 3, 4);

alert(arr); // 1,2,3,4,5
```

## **slice**

Il metodo `arr.slice ↗` è più semplice di `arr.splice`.

La sintassi è:

```
arr.slice([start], [end])
```

Ritorna un nuovo array contenente tutti gli elementi a partire da `"start"` fino ad `"end"` (`"end"` escluso). Sia `start` che `end` possono essere negativi; in tal caso si inizierà a contare dalla coda dell'array.

Funziona come `str.slice`, ma crea dei sotto-array piuttosto che sotto-stringhe.

Ad esempio:

```
let arr = ["t", "e", "s", "t"];
alert(arr.slice(1, 3)); // e,s (copia da 1 a 3)
alert(arr.slice(-2)); // s,t (copia da -2 fino alla fine)
```

Possiamo anche utilizzarlo senza argomenti: `arr.slice()` crea una copia di `arr`. Questo tipo di chiamata è spesso utilizzata per creare una copia con cui poter liberamente lavorare senza modificare l'array originale.

## concat

Il metodo `arr.concat ↗` crea un nuovo array che include valori di altri array, o elementi aggiuntivi.

La sintassi è:

```
arr.concat(arg1, arg2...)
```

Accetta un numero arbitrario di argomenti – sia array che valori.

Il risultato è un nuovo array contenente gli elementi di `arr`, seguiti da `arg1`, `arg2` etc.

Se un argomento `argN` è un array, tutti i suoi elementi vengono copiati. Altrimenti viene copiato solamente l'argomento stesso.

Un esempio:

```
let arr = [1, 2];
// unisce arr con [3,4]
alert(arr.concat([3, 4])); // 1,2,3,4
// unisce arr con [3,4] e [5,6]
alert(arr.concat([3, 4], [5, 6])); // 1,2,3,4,5,6
// unisce arr con [3,4], poi aggiunge i valori 5 e 6
alert(arr.concat([3, 4], 5, 6)); // 1,2,3,4,5,6
```

Normalmente copia elementi da un array. Gli altri oggetti, anche se assomigliano molto ad un array, vengono aggiunti interamente:

```
let arr = [1, 2];
```

```

let arrayLike = {
 0: "something",
 length: 1
};

alert(arr.concat(arrayLike)); // 1,2,[object Object]

```

...Se, invece, un oggetto simile ad un array possiede la proprietà `Symbol.isConcatSpreadable`, allora viene trattato come un array e i suoi elementi vengono copiati:

```

let arr = [1, 2];

let arrayLike = {
 0: "something",
 1: "else",
 [Symbol.isConcatSpreadable]: true,
 length: 2
};

alert(arr.concat(arrayLike)); // 1,2,something,else

```

## Iterate: `forEach`

Il metodo `arr.forEach ↗` consente di eseguire una funzione su ogni elemento dell'array.

La sintassi:

```

arr.forEach(function(item, index, array) {
 // ... fa qualcosa con l'elemento
});

```

Ad esempio, il codice sotto mostra ogni elemento dell'array:

```

// per ogni elemento chiama alert
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);

```

Invece questo codice ne mostra anche la posizione:

```

["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
 alert(`#${item} is at index ${index} in ${array}`);
});

```

Il risultato di questa funzione (sempre che ci sia) viene scartato.

## Ricerca in un array

Ora vedremo dei metodi per effettuare ricerche in un array.

## indexOf/lastIndexOf e includes

I metodi `arr.indexOf`, `arr.lastIndexOf` e `arr.includes` hanno la stessa sintassi, e fanno praticamente la stessa cosa della loro controparte per stringhe, ma operano su elementi invece che su caratteri:

- `arr.indexOf(item, from)` cerca un `item` a partire dall'indirizzo `from`, e ritorna l'indirizzo in cui è stato trovato, altrimenti ritorna `-1`.
- `arr.lastIndexOf(item, from)` – lo stesso, ma esegue la ricerca a partire da destra verso sinistra.
- `arr.includes(item, from)` – cerca un `item` a partire dall'indice `from`, e ritorna `true` se lo trova.

Ad esempio:

```
let arr = [1, 0, false];

alert(arr.indexOf(0)); // 1
alert(arr.indexOf(false)); // 2
alert(arr.indexOf(null)); // -1

alert(arr.includes(1)); // true
```

Da notare che questi metodi usano il confronto `==`. Quindi, se cerchiamo `false`, troveremo esattamente `false` e non zero.

Se vogliamo solo verificare la presenza di un elemento, senza voler conoscere l'indirizzo, è preferibile utilizzare il metodo `arr.includes`.

Inoltre, una piccola differenza è che `includes` gestisce correttamente `Nan`, a differenza di `indexof/lastIndexof`:

```
const arr = [NaN];
alert(arr.indexOf(NaN)); // -1 (dovrebbe essere 0, ma l'uguaglianza == non funziona con NaN)
alert(arr.includes(NaN));// true (corretto)
```

## find e findIndex

Immaginiamo di avere un array di oggetti. Come possiamo trovare un oggetto che soddisfi specifiche condizioni?

In questi casi si utilizza il metodo `arr.find`.

La sintassi è:

```
let result = arr.find(function(item, index, array) {
 // se viene ritornato true, viene ritornato l'elemento e l'iterazione si ferma
 // altrimenti ritorna undefined
});
```

La funzione viene chiamata per ogni elemento dell'array:

- `item` è l'elemento.

- `index` è il suo indice.
- `array` è l'array stesso.

Se la chiamata ritorna `true`, la ricerca viene interrotta e viene ritornato `item`. Se non viene trovato nulla verrà ritornato `undefined`.

Ad esempio, abbiamo un array di utenti, ognuno con i campi `id` e `name`. Cerchiamo quello con `id == 1`:

```
let users = [
 {id: 1, name: "John"},
 {id: 2, name: "Pete"},
 {id: 3, name: "Mary"}
];
let user = users.find(item => item.id == 1);
alert(user.name); // John
```

Gli array di oggetti sono molto comuni, quindi il metodo `find` risulta molto utile.

Da notare che nell'esempio noi forniamo a `find` un singolo argomento `item => item.id == 1`. Gli altri parametri di `find` sono raramente utilizzati.

Il metodo `arr.findIndex` ↪ fa essenzialmente la stessa cosa, ma ritorna l'indice in cui è stata trovata la corrispondenza piuttosto di ritornare l'oggetto stesso; se l'oggetto non viene trovato ritorna `-1`.

## filter

Il metodo `find` cerca un singola occorrenza dell'elemento, la prima, e se trovata ritorna `true`.

Se vogliamo cercare più occorrenze, possiamo utilizzare `arr.filter(fn)` ↪ .

La sintassi è pressoché la stessa di `find`, ma ritorna un array contenente tutte le corrispondenze trovate:

```
let results = arr.filter(function(item, index, array) {
 // se un item è true viene messo dentro results e l'iterazione continua
 // ritorna un array vuoto qualora nessun elemento ritornasse true
});
```

Ad esempio:

```
let users = [
 {id: 1, name: "John"},
 {id: 2, name: "Pete"},
 {id: 3, name: "Mary"}
];
// ritorna un array con i primi due users
let someUsers = users.filter(item => item.id < 3);
alert(someUsers.length); // 2
```

## Trasformare un array

Questa sezione di occupa dei metodi che trasformano o riordinano gli array.

### map

Il metodo `arr.map` è uno dei più utili e maggiormente utilizzati.

La sintassi è:

```
let result = arr.map(function(item, index, array) {
 // ritorna il nuovo valore piuttosto di item
})
```

La funzione viene chiamata per ogni elemento dell'array e ritorna un array con i risultati.

Ad esempio, qui trasformiamo ogni elemento nella propria `length`:

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
alert(lengths); // 5,7,6
```

### sort(fn)

Il metodo `arr.sort` ordina l'array *sul posto*, ovvero cambia la posizione originale dei suoi elementi.

Ritorna altresì l'array riordinato, ma il risultato viene di solito ignorato, essendo l'`arr` originale modificato.

Ad esempio:

```
let arr = [1, 2, 15];

// il metodo riordina il contenuto di arr (e lo ritorna)
arr.sort();

alert(arr); // 1, 15, 2
```

Notate qualcosa di strano nel risultato?

L'ordine degli elementi è diventato `1, 15, 2`. Errato. Ma perché?

**Di default gli elementi vengono ordinati come stringhe.**

Letteralmente, tutti gli elementi vengono convertiti in stringhe e confrontati. Quindi, viene applicato l'algoritmo di ordinamento lessicografico, perciò `"2" > "15"`.

Per utilizzare un ordinamento arbitrario, dobbiamo fornire una funzione come argomento di `arr.sort()`.

La funzione dovrebbe essere simile a questa:

```
function compare(a, b) {
 if (a > b) return 1; // se il primo valore è maggiore del secondo
 if (a == b) return 0; // se i valori sono uguali
```

```
 if (a < b) return -1; // se il primo valore è inferiore al secondo
}
```

Ad esempio, per ordinare dei numeri:

```
function compareNumeric(a, b) {
 if (a > b) return 1;
 if (a == b) return 0;
 if (a < b) return -1;
}

let arr = [1, 2, 15];

arr.sort(compareNumeric);

alert(arr); // 1, 2, 15
```

Ora funziona come dovrebbe.

Proviamo un attimo a capire cosa sta succedendo. L'array `arr` può contenere qualsiasi cosa, giusto? Può contenere numeri, stringhe, elementi HTML o qualsiasi altra cosa. Abbiamo quindi un insieme di *qualsiasi cosa*. Per poterlo ordinare abbiamo bisogno di una *funzione di ordinamento* che sappia ordinare gli elementi passati come argomenti. L'ordinamento di default è di tipo stringa.

Il metodo `arr.sort(fn)` implementa un algoritmo di ordinamento. Non dovremmo preoccuparci di come funzioni esattamente (la maggior parte delle volte è un [quicksort ↗](#) ottimizzato). Questo algoritmo attraverserà l'intero array e confronterà i suoi valori; tutto quello che dobbiamo fare sarà fornirgli una funzione `fn` che esegua il confronto.

In ogni caso, se mai volessimo conoscere quali elementi vengono comparati – nulla ci vieta di utilizzare `alert`:

```
[1, -2, 15, 2, 0, 8].sort(function(a, b) {
 alert(a + " <> " + b);
 return a - b;
});
```

L'algoritmo potrebbe confrontare un elemento più volte durante il processo, sebbene tenti di fare il minor numero di confronti possibili.

### **i Una funzione di confronto può ritornare qualsiasi numero**

In realtà, ad una funzione di confronto è solamente richiesto di ritornare un numero positivo per dire "maggiore" ed uno negativo per dire "minore".

Questo consente di scrivere funzioni più brevi:

```
let arr = [1, 2, 15];

arr.sort(function(a, b) { return a - b; });

alert(arr); // 1, 2, 15
```

### **i Le arrow functions sono le migliori**

Ricordate le Articolo "function-expressions-arrows" non trovato? Possiamo utilizzarle per più conciso il codice di ordinamento:

```
arr.sort((a, b) => a - b);
```

Questa funziona esattamente come le altre versioni sopra, ma è più breve.

### **i Use `localeCompare` for strings**

Ricordate l'algoritmo di comparazione delle [strings](#)? Di default, compara le lettere usando il loro codice.

Per molti alfabeti è meglio utilizzare `str.localeCompare` per ordinare correttamente lettere come Ö.

Per esempio, ordiniamo alcuni paesi in tedesco:

```
let countries = ['Österreich', 'Andorra', 'Vietnam'];

alert(countries.sort((a, b) => a > b ? 1 : -1)); // Andorra, Vietnam, Österreich (wrong)

alert(countries.sort((a, b) => a.localeCompare(b))); // Andorra, Österreich, Vietnam (correct)
```

## **reverse**

Il metodo `arr.reverse` ↗ inverte l'ordine degli elementi contenuti in `arr`.

Ad esempio:

```
let arr = [1, 2, 3, 4, 5];
arr.reverse();

alert(arr); // 5,4,3,2,1
```

Inoltre ritorna `arr` dopo averlo invertito.

## split e join

Vediamo una situazione reale. Stiamo scrivendo un'applicazione di messaggistica, e l'utente inserisce una lista di destinatari: `John, Pete, Mary`. Per noi sarebbe più comodo avere un array di nomi piuttosto di una singola stringa. Come possiamo ottenerlo?

Il metodo `str.split(delim)` ↗ fa esattamente questo. Divide la stringa in un array utilizzando il delimitatore `delim`.

Nell'esempio sotto, utilizziamo come delimitatore una virgola seguita da uno spazio:

```
let names = 'Bilbo, Gandalf, Nazgul';

let arr = names.split(', ');

for (let name of arr) {
 alert(`A message to ${name}.`); // A message to Bilbo (e altri name)
}
```

Il metodo `split` ha un secondo argomento opzionale di tipo numerico – è un limite di lunghezza per l'array. Se questo argomento viene fornito, allora gli elementi extra verranno ignorati. Ma nella pratica è raramente utilizzato:

```
let arr = 'Bilbo, Gandalf, Nazgul, Saruman'.split(', ', 2);

alert(arr); // Bilbo, Gandalf
```

### Split in lettere

La chiamata a `split(s)` con l'argomento vuoto, dividerà la stringa in un array di lettere:

```
let str = "test";

alert(str.split('')); // t,e,s,t
```

La chiamata ad `arr.join(separatore)` ↗ fa esattamente l'inverso di `split`. Crea una stringa con gli elementi di `arr` incollati tra loro dal `separatore`.

Ad esempio:

```
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];

let str = arr.join(';'); // incolla l'array utilizzando ;

alert(str); // Bilbo;Gandalf;Nazgul
```

## reduce/reduceRight

Quando vogliamo iterare su un array – possiamo utilizzare `forEach`, `for` o `for..of`.

Quando invece abbiamo la necessità di iterare e ritornare dati per ogni elemento – possiamo usare `map`.

I metodi `arr.reduce` ↗ e `arr.reduceRight` ↗ fanno parte della stessa categoria, ma sono leggermente più complessi. Vengono utilizzati per calcolare un singolo valore basato sul contenuto dell'array.

La sintassi è:

```
let value = arr.reduce(function(accumulator, item, index, array) {
 // ...
}, [initial]);
```

La funzione viene applicata ad ogni elemento dell'array uno dopo l'altro, passando il risultato alla chiamata successiva.

Argomenti:

- `accumulator` – è il risultato della precedente chiamata, uguale ad `initial` per la prima chiamata (se `initial` viene fornito).
- `item` – è l'attuale elemento dell'array.
- `index` – la sua posizione.
- `array` – l'array.

Quando la funzione è stata applicata, il risultato viene passato alla chiamata successiva.

Sembra complicato, ma non lo è se pensate al primo argomento come un “accumulatore” che memorizza il risultato delle precedenti esecuzioni. E alla fine diventa il risultato di `reduce`.

Il modo più semplice per spiegarlo è tramite esempi.

Qui otterremo una somma degli elementi dell'array in una riga:

```
let arr = [1, 2, 3, 4, 5];

let result = arr.reduce((sum, current) => sum + current, 0);

alert(result); // 15
```

Qui abbiamo utilizzato la variante più comune di `reduce` con solo 2 argomenti.

Proviamo a vedere nel dettaglio cosa succede.

1. Nella prima esecuzione, `sum` è il valore iniziale (l'ultimo argomento di `reduce`), cioè `0`, e `current` è il primo elemento dell'array, cioè `1`. Quindi il risultato è `1`.
2. Nella seconda esecuzione, `sum = 1`; gli sommiamo il secondo elemento dell'array(`2`) e ritorniamo il risultato.
3. Nella terza esecuzione, `sum = 3`; gli sommiamo l'elemento successivo, e così via...

Il flusso di calcolo:

sum 0	sum 0+1	sum 0+1+2	sum 0+1+2+3	sum 0+1+2+3+4
current 1	current 2	current 3	current 4	current 5
1	2	3	4	5

→  $0+1+2+3+4+5 = 15$

O nella forma tabellare, in cui ogni riga rappresenta una chiamata di funzione:

	sum	current	result
prima chiamata	0	1	1
seconda chiamata	1	2	3
terza chiamata	3	3	6
quarta chiamata	6	4	10
quinta chiamata	10	5	15

Come abbiamo potuto osservare, il risultato della chiamata precedente diventa il primo argomento della chiamata successiva.

Possiamo anche omettere il valore iniziale:

```
let arr = [1, 2, 3, 4, 5];

// rimosso il valore iniziale da reduce (no 0)
let result = arr.reduce((sum, current) => sum + current);

alert(result); // 15
```

Il risultato sarebbe lo stesso. Questo perché se non c'è un valore iniziale, allora `reduce` prende il primo elemento dell'array come valore iniziale ed inizia l'iterazione dal secondo elemento.

La tabella dei calcoli è uguale a quella precedente, viene saltata solo la prima riga.

Questo tipo di utilizzo richiede particolare cura. Se l'array è vuoto, allora `reduce` effettua la chiamata senza valore iniziale e provoca un errore.

Vediamo un esempio:

```
let arr = [];

// Errore: Riduzione di un array vuoto senza valore iniziale
// se il valore iniziale esistesse, reduce lo restituirebbe all'array vuoto.
arr.reduce((sum, current) => sum + current);
```

Quindi è fortemente consigliato di specificare sempre un valore iniziale.

Il metodo `arr.reduceRight ↗` fa esattamente la stessa cosa, ma da destra verso sinistra.

## Array.isArray

Gli array non sono un tipo di dato a sé del linguaggio. Sono basati sulla sintassi degli oggetti.

Quindi `typeof` non aiuta a distinguere un oggetto da un array:

```
alert(typeof {}); // object
alert(typeof []); // lo stesso
```

...Ma gli array vengono utilizzati così spesso che esiste un metodo dedicato per questo: `Array.isArray(value)`. Ritorna `true` se `value` è un array, `false` altrimenti.

```
alert(Array.isArray({})); // false
alert(Array.isArray([])); // true
```

## La maggior parte dei metodi accetta “`thisArg`”

Quasi tutti i metodi degli array che richiedono una funzione – come `find`, `filter`, `map`, fatta eccezione per `sort`, accettano un parametro opzionale `thisArg`.

Questo parametro non è stato spiegato nella sezione sopra, perché viene raramente utilizzato. Studiamolo per completezza.

Vediamo la sintassi di questi metodi:

```
arr.find(func, thisArg);
arr.filter(func, thisArg);
arr.map(func, thisArg);
// ...
// thisArg è l'ultimo argomento opzionale
```

Il valore del parametro `thisArg` diventa `this` per `func`.

Ad esempio, qui utilizziamo il metodo di un oggetto come filtro e `thisArg` ci risulta utile:

```
let army = {
 minAge: 18,
 maxAge: 27,
 canJoin(user) {
 return user.age >= this.minAge && user.age < this.maxAge;
 }
};

let users = [
 {age: 16},
 {age: 20},
 {age: 23},
 {age: 30}
];

// trova tutti gli users più giovani di user
let soldiers = users.filter(army.canJoin, army);

alert(soldiers.length); // 2
```

```
alert(soldiers[0].age); // 20
alert(soldiers[1].age); // 23
```

Nella chiamata sopra, utilizziamo `army.canJoin` come filtro e forniamo `army` come contesto. Se non avessimo fornito il contesto, `users.filter(army.canJoin)` avrebbe chiamato `army.canJoin` come funzione a sé stante, con `this=undefined`. Che avrebbe provocato un errore.

Una chiamata a `user.filter(army.canJoin, army)` può essere sostituita da `users.filter(user => army.canJoin(user))`, che fa lo stesso. L'ultima versione viene utilizzata più spesso e per molte persone è più semplice da capire.

## Riepilogo

Un breve riepilogo dei metodi per array:

- Per aggiungere/rimuovere elementi:
  - `push(...items)` – aggiunge elementi in coda,
  - `pop()` – estrae un elemento dalla coda,
  - `shift()` – estrae un elemento dalla testa,
  - `unshift(...items)` – aggiunge un elemento alla testa.
  - `splice(pos, deleteCount, ...items)` – all'indice `pos` cancella `deleteCount` elementi e al loro posto inserisce `items`.
  - `slice(start, end)` – crea un nuovo array e copia al suo interno gli elementi da `start` fino ad `end` (escluso).
  - `concat(...items)` – ritorna un nuovo array: copia tutti gli elementi di quello corrente e ci aggiunge `items`. Se uno degli `items` è un array, allora vengono presi anche i suoi elementi.
- Ricercare elementi:
  - `indexOf/lastIndexOf(item, pos)` – cerca `item` a partire da `pos`, e ritorna l'indice, oppure `-1` se non lo trova.
  - `includes(value)` – ritorna `true` se l'array contiene `value`, altrimenti `false`.
  - `find/filter(func)` – filtra gli elementi tramite una funzione, ritorna il primo/tutti i valori che ritornano `true`.
  - `findIndex` è simile a `find`, ma ritorna l'indice piuttosto del valore.
- Per iterare sugli elementi:
  - `forEach(func)` – invoca `func` su ogni elemento; non ritorna nulla.
- Per modificare un array:
  - `map(func)` – crea un nuovo array con i risultati della chiamata `func` su tutti i suoi elementi.
  - `sort(func)` – ordina l'array “sul posto”, e lo ritorna.
  - `reverse()` – inverte l'array sul posto, e lo ritorna.
  - `split/join` – converte una stringa in array e vice versa.

- `reduce/reduceRight(func, initial)` – calcola un singolo valore chiamando `func` per ogni elemento e passando un risultato temporaneo tra una chiamata e l'altra
- Un altro metodo utile:
  - `Array.isArray(arr)` controlla che `arr` sia un array.

Da notare che i metodi `sort`, `reverse` e `splice` modificano l'array stesso.

I metodi elencati sono quelli utilizzati più spesso e sono in grado di coprire il 99% dei casi d'uso. Ce ne sono altri che possono tornare utili:

- `arr.some(fn)` /`arr.every(fn)` controlla l'array.

La funzione `fn` viene invocata su ogni elemento dell'array in maniera simile a `map`. Se qualcuno/tutti i risultati sono `true`, ritorna `true`, altrimenti `false`.

Questi metodi si comportano quasi come gli operatori `||` e `&&`: se `fn` ritorna un valore vero, `arr.some()` ritorna immediatamente `true` e conclude l'iterazione; se `fn` ritorna un valore falso, `arr.every()` ritorna immediatamente `false` e smette di iterare.

Possiamo utilizzare `every` per confrontare gli array:

```
function arraysEqual(arr1, arr2) {
 return arr1.length === arr2.length && arr1.every((value, index) => value === arr2[index]);
}

alert(arraysEqual([1, 2], [1, 2])); // true
```

- `arr.fill(value, start, end)` – riempie l'array con `value` da `start` fino a `end`.
- `arr.copyWithin(target, start, end)` – copia gli elementi da `start` fino a `end` dentro se stesso, nella posizione `target` (sovrascrivendo gli elementi contenuti).
- `arr.flat(depth)` /`arr.flatMap(fn)` crea un nuovo array monodimensionale partendo da un array multidimensionale.

Per la lista completa, vedere il [manuale](#).

A prima vista potrebbero sembrare molti metodi da ricordare. Ma in realtà è molto più semplice di quanto sembri.

Tenete sempre un occhio al riassunto fornito sopra. Provate anche a risolvere gli esercizi di questo capitolo.

In futuro quando avrete bisogno di fare qualcosa con un array, e non saprete come fare – tornate qui, guardate il riassunto e trovate il metodo corretto. Gli esempi vi aiuteranno molto. In poco tempo vi risulterà naturale ricordare questi metodi, senza troppi sforzi.

## ✓ Esercizi

### Traducete border-left-width in borderLeftWidth

importanza: 5

Scrivete una funzione `camelize(str)` che trasforma le parole separate da un trattino come "la-mia-stringa" nella notazione a cammello "laMiaStringa".

Quindi: rimuove tutti i trattini; ogni parola dopo un trattino avrà una lettera maiuscola.

Esempi:

```
camelize("background-color") == 'backgroundColor';
camelize("list-style-image") == 'listStyleImage';
camelize("-webkit-transition") == 'WebkitTransition';
```

P.S. Suggerimento: usate `split` per dividere una stringa in un array, trasformatela e infinite riunite tutto con `join`.

Apri una sandbox con i test. ↗

Alla soluzione

---

## Filtri

importanza: 4

Scrivete una funzione `filterRange(arr, a, b)` che accetta come argomento un array `arr`, filtra gli elementi tra `a` e `b` e ne ritorna un array.

La funzione non dovrebbe modificare l'array. Dovrebbe invece ritornare il nuovo array.

Ad esempio:

```
let arr = [5, 3, 8, 1];
let filtered = filterRange(arr, 1, 4);
alert(filtered); // 3,1 (i valori filtrati)
alert(arr); // 5,3,8,1 (non modificato)
```

Apri una sandbox con i test. ↗

Alla soluzione

---

## Filtrare un range "sul post"

importanza: 4

Scrivi una funzione `filterRangeInPlace(arr, a, b)` che prenda un array `arr` e ne rimuova tutti i valori, tranne quelli contenuti tra `a` e `b`. Il test è: `a ≤ arr[i] ≤ b`.

La funzione dovrebbe solamente modificare l'array. Senza ritornare nulla.

Ad esempio:

```
let arr = [5, 3, 8, 1];

filterRangeInPlace(arr, 1, 4); // rimuove tutti i numeri tranne quelli da 1 a 4

alert(arr); // [3, 1]
```

Apri una sandbox con i test. [↗](#)

[Alla soluzione](#)

---

## Riordinare in ordine decrescente

importanza: 4

```
let arr = [5, 2, 1, -10, 8];

// ... il tuo codice per ordinare in ordine decrescente

alert(arr); // 8, 5, 2, 1, -10
```

[Alla soluzione](#)

---

## Copiare e ordinare un array

importanza: 5

Abbiamo un array di stringhe `arr`. Vorremmo ottenerne una sua copia ordinata, mantenendone `arr` inalterato.

Create una funzione `copySorted(arr)` che ritorni questo tipo di copia.

```
let arr = ["HTML", "JavaScript", "CSS"];

let sorted = copySorted(arr);

alert(sorted); // CSS, HTML, JavaScript
alert(arr); // HTML, JavaScript, CSS (nessuna modifica)
```

[Alla soluzione](#)

---

## Create una calcolatrice estensibile

importanza: 5

Create un costruttore `Calculator` che crei oggetti calcolatrice “estensibili”.

Il compito consiste in due parti.

1.

La prima parte consiste nell’implementare il metodo `calculate(str)` che accetti una stringa come `"1 + 2"` nel formato “NUMERO operatore NUMERO” (delimitata da spazi) e ne ritorni il risultato. Dovrebbe saper interpretare sia `+` che `-`.

Esempio d'uso:

```
let calc = new Calculator;

alert(calc.calculate("3 + 7")); // 10
```

2.

Successivamente aggiungete il metodo `addMethod(name, func)` che ha lo scopo di insegnare alla calcolatrice una nuova operazione. Questo prende il nome dell'operatore `name` e i due argomenti della funzione `func(a, b)` che lo implementa.

Ad esempio, proviamo ad aggiungere la moltiplicazione `*`, divisione `/` e la potenza `**`:

```
let powerCalc = new Calculator;
powerCalc.addMethod("*", (a, b) => a * b);
powerCalc.addMethod("/", (a, b) => a / b);
powerCalc.addMethod("**", (a, b) => a ** b);

let result = powerCalc.calculate("2 ** 3");
alert(result); // 8
```

- Non è richiesta la gestione delle parentesi o di operazioni complesse.
- I numeri e l'operatore sono separati esattamente da un singolo spazio.
- Se ne hai voglia potresti provare ad aggiungere un minimo di gestione degli errori.

Apri una sandbox con i test. ↗

[Alla soluzione](#)

## Map di nomi

importanza: 5

Avete un array di oggetti `user`; ognuno di essi ha la proprietà `user.name`. Scrivete il codice che converte gli oggetti in un array di nomi.

Ad esempio:

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let users = [john, pete, mary];

let names = /* ... il vostro codice */

alert(names); // John, Pete, Mary
```

[Alla soluzione](#)

## Map di oggetti

importanza: 5

Avete un array di oggetti `user`, ognuno di questi possiede `name`, `surname` e `id`.

Scrivete il codice per creare un altro array che derivi da questo, sempre composto da oggetti con `id` e `fullName`, dove `fullName` viene generato da `name` e `surname`.

Un esempio:

```
let john = { name: "John", surname: "Smith", id: 1 };
let pete = { name: "Pete", surname: "Hunt", id: 2 };
let mary = { name: "Mary", surname: "Key", id: 3 };

let users = [john, pete, mary];

let usersMapped = /* ... il vostro codice ... */

/*
usersMapped = [
 { fullName: "John Smith", id: 1 },
 { fullName: "Pete Hunt", id: 2 },
 { fullName: "Mary Key", id: 3 }
]
*/

alert(usersMapped[0].id) // 1
alert(usersMapped[0].fullName) // John Smith
```

Quindi, in realtà avrete bisogno di mappare un array di oggetti in un altro. Provate ad utilizzare `=>`.

[Alla soluzione](#)

## Riordinare oggetti per età

importanza: 5

Scrivete una funzione `sortByAge(users)` che accetti un array di oggetti con proprietà `age` e lo riordini per `age`.

Ad esempio:

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let arr = [pete, john, mary];

sortByAge(arr);

// ora: [john, mary, pete]
alert(arr[0].name); // John
```

```
alert(arr[1].name); // Mary
alert(arr[2].name); // Pete
```

[Alla soluzione](#)

## Rimescolare un array

importanza: 3

Scrivete una funzione `shuffle(array)` che rimescoli (riordini casualmente) gli elementi di un array.

Esecuzioni multiple di `shuffle` dovrebbero portare a diversi ordinamenti degli elementi. Ad esempio:

```
let arr = [1, 2, 3];

shuffle(arr);
// arr = [3, 2, 1]

shuffle(arr);
// arr = [2, 1, 3]

shuffle(arr);
// arr = [3, 1, 2]
// ...
```

Tutti gli elementi ordinati dovrebbero avere una probabilità identica. Ad esempio, `[1, 2, 3]` può essere riordinato come `[1, 2, 3]` o `[1, 3, 2]` o `[3, 1, 2]` etc; ognuno dei casi deve avere la stessa probabilità.

[Alla soluzione](#)

## Ottenere l'età media+

importanza: 4

Scrivete una funzione `getAverageAge(users)` che accetti un array di oggetti con la proprietà `age` e ritorni l'età media.

La formula della media è:  $(\text{age1} + \text{age2} + \dots + \text{ageN}) / N$ .

Ad esempio:

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 29 };

let arr = [john, pete, mary];

alert(getAverageAge(arr)); // (25 + 30 + 29) / 3 = 28
```

[Alla soluzione](#)

## Filtrare un array per ottenere elementi unici

importanza: 4

Abbiamo un array `arr`.

Create una funzione `unique(arr)` che ritorni un array con elementi unici.

Ad esempio:

```
function unique(arr) {
 /* your code */
}

let strings = ["Hare", "Krishna", "Hare", "Krishna",
 "Krishna", "Krishna", "Hare", "Hare", ":-0"
];

alert(unique(strings)); // Hare, Krishna, :-0
```

Apri una sandbox con i test. ↗

[Alla soluzione](#)

---

## Create un oggetto da un array

importanza: 4

Immaginiamo di ricevere un array di utenti nella forma `{id:..., name:..., age...}`.

Scrivi una funzione `groupById(arr)` che ricavi un oggetto da esso, con `id` come chiave e gli elementi dell'array come valori

Ad esempio:

```
let users = [
 {id: 'john', name: "John Smith", age: 20},
 {id: 'ann', name: "Ann Smith", age: 24},
 {id: 'pete', name: "Pete Peterson", age: 31},
];

let usersById = groupById(users);

/*
// dopo la chiamata dovremmo avere:

usersById = {
 john: {id: 'john', name: "John Smith", age: 20},
 ann: {id: 'ann', name: "Ann Smith", age: 24},
 pete: {id: 'pete', name: "Pete Peterson", age: 31},
}
```

Una funzione simile è molto utile quando si lavora con dati provenienti da un server.

In questo esercizio sappiamo che `id` è unico. Non ci saranno due array con lo stesso `id`.

Per favore utilizza il metodo `.reduce` nella soluzione.

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

## Iteratori

Gli oggetti *iterabili* sono una generalizzazione degli array. Questo concetto consente a qualsiasi oggetto di essere utilizzato in un ciclo `for..of`.

Ovviamente, gli array sono oggetti iterabili. Ma ci sono molti altri oggetti integrati, che sono altresì iterabili. Ad esempio, le stringhe.

Se un oggetto rappresenta una collezione (lista, insieme) di qualcosa, allora `for..of` è un ottimo modo per eseguire un ciclo, quindi ora vedremo come farlo funzionare correttamente.

## Symbol.iterator

Possiamo spiegare meglio il funzionamento degli oggetti iterabili costruendone uno nostro.

Ad esempio, abbiamo un oggetto, che non è un array, ma sembra essere adatto ad un `for..of`.

Come un oggetto `range` che rappresenta un intervallo numerico:

```
let range = {
 from: 1,
 to: 5
};

// Vorremmo che il for..of funzioni:
// for(let num of range) ... num=1,2,3,4,5
```

Per rendere iterabile l'oggetto `range` (e poter quindi utilizzare correttamente `for..of`) abbiamo bisogno di aggiungervi un metodo chiamato `Symbol.iterator` (uno speciale simbolo integrato).

1. Quando `for..of` inizia, prova a chiamare questo metodo (o ritorna un errore se non lo trova). Il metodo deve ritornare un *iteratore* – un oggetto con il metodo `next`.
2. La possibilità di avanzare di `for..of` funziona *soltamente con l'oggetto ritornato*.
3. Quando `for..of` vuole il prossimo valore, chiama `next()` su quell'oggetto.
4. Il risultato di `next()` deve avere la forma `{done: Boolean, value: any}`, dove `done=true` significa che l'iterazione è completa, altrimenti `value` deve contenere il valore successivo.

Qui l'implementazione completa per `range`:

```

let range = {
 from: 1,
 to: 5
};

// 1. la chiamata a for..of inizialmente chiama questo
range[Symbol.iterator] = function() {

 // ...ritorna l'oggetto iteratore:
 // 2. Da qui in poi, for..of lavora solamente con questo iteratore, richiedendogli il prossimo
 return {
 current: this.from,
 last: this.to,

 // 3. next() viene invocato ad ogni iterazione del ciclo for..of
 next() {
 // 4. dovrebbe ritornare il valore sotto forma di oggetto {done:..., value :...}
 if (this.current <= this.last) {
 return { done: false, value: this.current++ };
 } else {
 return { done: true };
 }
 }
 };
};

// ora funziona !
for (let num of range) {
 alert(num); // 1, poi 2, 3, 4, 5
}

```

Da notare la caratteristica fondamentale degli oggetti iterabili: un importante separazione di concetti:

- Il `range` stesso non possiede un metodo `next()`.
- Invece, un altro oggetto, detto “iteratore”, viene creato dalla chiamata `range[Symbol.iterator]()`, e gestisce l’intera iterazione.

Quindi, l’oggetto iteratore è separato da quello su cui itera.

Tecnicamente, per rendere il codice più semplice, potremmo unirli e utilizzare `range` stesso.

Come nel seguente codice:

```

let range = {
 from: 1,
 to: 5,

 [Symbol.iterator]() {
 this.current = this.from;
 return this;
 },

 next() {
 if (this.current <= this.to) {
 return { done: false, value: this.current++ };
 } else {
 return { done: true };
 }
 }
};

```

```
 }
 }
};

for (let num of range) {
 alert(num); // 1, poi 2, 3, 4, 5
}
```

Ora `range[Symbol.iterator]()` ritorna l'oggetto `range` stesso: questo ha il metodo `next()` necessario e memorizza il progresso dell'iterazione in `this.current`. Più corto? Sì. E molte volte può andare bene.

Il lato negativo è che ora è impossibile avere due cicli `for .. of` che iterano sull'oggetto contemporaneamente: infatti condividerebbero lo stesso stato di iterazione, poiché c'è solo un oggetto iteratore – l'oggetto stesso. In ogni caso due cicli `for .. of` simultanei sono molto rari, persino in alcuni scenari asincroni.

### Iteratori infiniti

Sono possibili anche iteratori infiniti. Ad esempio, l'oggetto `range` diventa infinito quando `range.to = Infinity`. Oppure possiamo creare un oggetto iterabile che generi un'infinita sequenza di numeri pseudo-casuali.

Non c'è alcun limite per `next`: può ritornare più e più valori.

Ovviamente, il ciclo `for .. of` diventerebbe infinito. Possiamo comunque fermarlo con `break`.

## Le stringhe sono iterabili

Gli array e le stringhe sono gli oggetti su cui si utilizzano di più gli iteratori.

Per una stringa, `for .. of` itera sui caratteri:

```
for (let char of "test") {
 // attivato 4 volte: una volta per ogni carattere
 alert(char); // t, poi e, poi s, poi t
}
```

E funziona correttamente anche con le coppie surrogate!

```
let str = '𠮷𠮷';
for (let char of str) {
 alert(char); // 𠮷, e poi 𠮷
}
```

## Chiamare un iteratore esplicitamente

Normalmente, il funzionamento degli iteratori è nascosto al codice esterno. C'è un ciclo `for .. of`, che funziona, e questo è tutto ciò che serve sapere.

Ma per approfondire, vediamo come creare esplicitamente un iteratore.

Proveremo ad iterare su una stringa allo stesso modo di un ciclo `for .. of`, ma con una chiamata diretta. Questo codice crea un iteratore per stringhe e lo richiama “manualmente”:

```
let str = "Hello";

// fa la stessa cosa di
// for (let char of str) alert(char);

let iterator = str[Symbol.iterator]();

while (true) {
 let result = iterator.next();
 if (result.done) break;
 alert(result.value); // stampa i caratteri uno ad uno
}
```

Raramente è necessario, ma ci fornisce maggiore controllo sul processo di iterazione rispetto a `for .. of`. Ad esempio, possiamo dividere il processo di iterazione: eseguiamo un paio di iterazioni, ci fermiamo, facciamo altro, e riprendiamo l’iterazione.

## Iteratori e simil-array

Ci sono due termini ufficiali che sembrano simili, ma sono diversi. Dobbiamo essere certi di aver ben capito la differenza per evitare confusione.

- Gli *oggetti iterabili* sono oggetti che implementano il metodo `Symbol.iterator`, come descritto sopra.
- *Array-like* (simil-array) sono oggetti che hanno indici e una proprietà `length`, per questo assomigliano ai classici array.

Naturalmente, queste proprietà possono essere combinate. Ad esempio, le stringhe sono sia iterabili (`for .. of` funziona), sia array-like (possiedono indici numerici e una proprietà `length`).

Ma un oggetto iterabile potrebbe non essere array-like. E vice versa un array-like potrebbe non essere un oggetto iterabile.

Ad esempio, l’esempio `range` utilizzato sopra è un oggetto iterabile, ma non array-like, poiché non possiede degli indici e la proprietà `length`.

Qui invece vediamo un esempio di oggetto array-like, ma che non è iterabile:

```
let arrayLike = { // possiede indici e lenght => array-like
 0: "Hello",
 1: "World",
 length: 2
};

// Errore (Symbol.iterator non trovato)
for (let item of arrayLike) {}
```

Cosa hanno in comune? Entrambi, sia gli array-like che gli oggetti iterabili, solitamente *non sono array*, non hanno metodi come `push`, `pop` etc. Questo potrebbe essere scomodo se lavoriamo con uno di questi oggetti trattandoli come fossero un array.

## Array.from

Esiste un metodo universale `Array.from` ↗ che può risolvere questo problema. Questo prende un oggetto iterabile o un array-like e ne crea un `Array` “vero e proprio”. Possiamo così utilizzare i classici metodi per array.

Ad esempio:

```
let arrayLike = {
 0: "Hello",
 1: "World",
 length: 2
};

let arr = Array.from(arrayLike); // (*)
alert(arr.pop()); // World (il metodo funziona)
```

`Array.from` alla riga (\*) prende l'oggetto, controlla se questo è un iterabile o un array-like, e successivamente crea un nuovo array e copia al suo interno tutti gli elementi.

Si comporta allo stesso modo con un oggetto iterabile:

```
// assumiamo che il range venga preso dall'esempio sopra
let arr = Array.from(range);
alert(arr); // 1,2,3,4,5 (la conversione array toString funziona)
```

La sintassi completa di `Array.from` consente di fornire una funzione opzionale di “map”:

```
Array.from(obj[, mapFn, thisArg])
```

Il secondo argomento `mapFn` (opzionale) è una funzione da applicare ad ogni elemento prima di aggiungerlo all'array, mentre `thisArg` ci consente di impostare `this`.

Ad esempio:

```
// assumiamo che il range venga preso dall'esempio sopra

// quadrato di ogni numero
let arr = Array.from(range, num => num * num);

alert(arr); // 1,4,9,16,25
```

Qui utilizziamo `Array.from` per convertire una stringa in un array di caratteri:

```
let str = 'X@';
```

```
// spezza str in un array di caratteri
let chars = Array.from(str);

alert(chars[0]); // X
alert(chars[1]); // ☺
alert(chars.length); // 2
```

A differenza di `str.split`, si basa sulla caratteristica di oggetto iterabile del tipo stringa e quindi, proprio come `for..of`, funziona correttamente con le coppie surrogate.

Tecnicamente qui facciamo la stessa cosa:

```
let str = 'X☺𩲷';

let chars = []; // Array.from internamente esegue lo stesso ciclo
for (let char of str) {
 chars.push(char);
}

alert(chars);
```

...Ma è più breve.

Possiamo anche eseguire un surrogato di `slice`:

```
function slice(str, start, end) {
 return Array.from(str).slice(start, end).join('');
}

let str = 'X☺𩲷';

alert(slice(str, 1, 3)); // ☺𩲷

// il metodo nativo non supporta le coppie surrogate
alert(str.slice(1, 3)); // spazzatura (due pezzi da coppie surrogate differenti)
```

## Riepilogo

Gli oggetti che possono essere utilizzati in `for..of` vengono detti *iterabili*.

- Tecnicamente, gli oggetti iterabili devono implementare un metodo chiamato `Symbol.iterator`.
  - Il risultato di `obj[Symbol.iterator]` viene chiamato *iteratore*. Esso si occupa di gestire l'intero processo di iterazione.
  - Un iteratore deve avere un metodo denominato `next()` che ritorni un oggetto `{done: Boolean, value: any}`; `done:true` indica la fine dell'iterazione, altrimenti `value` contiene il prossimo valore.
- Il metodo `Symbol.iterator` viene invocato automaticamente da `for..of`, ma possiamo anche farlo manualmente.
- Gli oggetti iterabili integrati, come le stringhe o gli array, implementano `Symbol.iterator`.

- L'iteratore che opera con le stringhe è a conoscenza dell'esistenza delle coppie surrogate.

Gli oggetti che hanno indici e la proprietà `length` vengono definiti *array-like*. Questo tipo di oggetti possono anche possedere altri metodi e proprietà, ma non possiedono gli stessi metodi integrati dagli array.

Se guardassimo dentro la specifica – vedremmo che la maggior parte dei metodi integrati suppongono di operare con oggetti iterabili o array-like invece che con “veri” array, poiché con i primi si riesce ad operare in maniera più astratta.

`Array.from(obj[, mapFn, thisArg])` crea un vero `Array` a partire da un oggetto `obj` iterabile o da un array-like; possiamo così applicare i classici metodi dedicati agli array. C’è anche la possibilità di fornire due argomenti opzionali `mapFn` e `thisArg` che consentono di applicare una funzione ad ogni elemento prima di aggiungerlo all’array finale.

## Map e Set

Finora abbiamo appreso le nozioni di base riguardo le seguenti strutture dati:

- Oggetti, per la memorizzazione di collezioni identificate da una chiave.
- Array, per la memorizzazione di collezioni ordinate.

Queste però non sono sufficienti. Esistono ulteriori strutture dati, come `Map` e `Set`.

## Map

[Map ↗](#) è una collezione di dati identificati da chiavi, proprio come un `Object` (Oggetto). La principale differenza è che `Map` accetta chiavi di qualsiasi tipo.

I metodi e le proprietà sono:

- `new Map()` – crea la mappa.
- `map.set(key, value)` – memorizza il valore `value` con la chiave `key`.
- `map.get(key)` – ritorna il valore associato alla chiave `key`, `undefined` nel caos in cui `key` non esista.
- `map.has(key)` – ritorna `true` se la chiave `key` esiste, `false` altrimenti.
- `map.delete(key)` – rimuove il valore con la chiave `key`.
- `map.clear()` – rimuove tutti gli elementi.
- `map.size` – ritorna il numero di elementi contenuti.

Ad esempio:

```
let map = new Map();

map.set('1', 'str1'); // una chiave di tipo stringa
map.set(1, 'num1'); // una chiave di tipo numerico
map.set(true, 'bool1'); // una chiave di tipo booleano

// ricordi gli oggetti standard? convertirebbero le chiavi a stringa
// Map invece mantiene il tipo, quindi i seguenti esempi sono differenti:
alert(map.get(1)); // 'num1'
```

```
alert(map.get('1')); // 'str1'
```

```
alert(map.size); // 3
```

Come abbiamo potuto osservare, a differenza degli oggetti, le chiavi non vengono convertite a stringa. Sono quindi ammesse chiavi di qualunque tipo.

**i map[key] non è il modo corretto di utilizzare una Map**

Anche se `map[key]` funziona, ad esempio possiamo impostare `map[key] = 2`, questo equivale a trattare `map` come un oggetto semplice, con tutte le limitazioni correlate.

Quindi dovremmo utilizzare i metodi dedicati a `map: set, get` e gli altri.

### Map può utilizzare anche oggetti come chiave.

Ad esempio:

```
let john = { name: "John" };

// per ogni utente, memorizziamo il contatore delle visite
let visitsCountMap = new Map();

// john è la chiave
visitsCountMap.set(john, 123);

alert(visitsCountMap.get(john)); // 123
```

Il fatto di poter utilizzare oggetti come chiavi è una delle caratteristiche più importanti fornite dalla struttura dati `Map`. In un normale `Object` una chiave di tipo stringa può andare bene, ma non vale lo stesso per le chiavi di tipo oggetto.

Proviamo:

```
let john = { name: "John" };
let ben = { name: "Ben" };

visitsCountObj[ben] = 234; // proviamo ad utilizzare l'oggetto ben come chiave
visitsCountObj[john] = 123; // proviamo ad utilizzare l'oggetto jhon come chiave, l'oggetto ben

// Questo è quello che otteniamo!
alert(visitsCountObj["[object Object]"]); // 123
```

Dal momento che `visitsCountObj` è un oggetto, converte tutte le chiavi, come `john` e `ben`, a stringhe, quindi otteniamo la chiave `"[object Object]"`. Senza dubbio non ciò che ci aspettavamo.

### **i** Come Map confronta le chiavi

Per verificare l'equivalenza delle chiavi, Map utilizza l'algoritmo [SameValueZero ↗](#). E' quasi la stessa cosa dell'uguaglianza stretta `==`, con la differenza che `Nan` viene considerato uguale a `Nan`. Quindi anche `Nan` può essere utilizzato come chiave.

L'algoritmo di confronto non può essere né cambiato né modificato.

### **i** concatenamento

Ogni chiamata a `map.set` ritorna la mappa stessa, quindi possiamo concatenare le chiamate:

```
map.set('1', 'str1')
 .set(1, 'num1')
 .set(true, 'bool1');
```

## Iterare su Map

Per iterare attraverso gli elementi di `Map`, esistono 3 metodi:

- `map.keys()` – ritorna un oggetto per iterare sulle chiavi,
- `map.values()` – ritorna un oggetto per iterare sui valori,
- `map.entries()` – ritorna un oggetto per iterare sulle coppie `[key, value]`, ed è il metodo utilizzato di default nel ciclo `for..of`.

Ad esempio:

```
let recipeMap = new Map([
 ['cucumber', 500],
 ['tomatoes', 350],
 ['onion', 50]
]);

// itera sulle chiavi (vegetables)
for (let vegetable of recipeMap.keys()) {
 alert(vegetable); // cucumber, tomatoes, onion
}

// itera sui valori (amounts)
for (let amount of recipeMap.values()) {
 alert(amount); // 500, 350, 50
}

// itera sulle voci [key, value]
for (let entry of recipeMap) { // equivale a recipeMap.entries()
 alert(entry); // cucumber,500 (and so on)
}
```

### Viene utilizzato l'ordine di inserimento

L'iterazione segue l'ordine di inserimento dei valori. `Map` mantiene l'ordine, a differenza degli `Object`.

Inoltre, `Map` possiede un suo metodo `forEach`, simile a quello utilizzato dagli `Array`:

```
// esegue la funzione per ogni coppia (chiave, valore)
recipeMap.forEach((value, key, map) => {
 alert(` ${key}: ${value}`); // cucumber: 500 etc
});
```

### `Object.entries: Map da Object`

Durante la fase di creazione di una `Map`, possiamo passarle un array (o qualsiasi altra struttura dati iterabile) con coppie chiave/valore per inizializzare la `Map`, come nel seguente esempio:

```
// array di coppie [chiave, valore]
let map = new Map([
 ['1', 'str1'],
 [1, 'num1'],
 [true, 'bool1']
]);

alert(map.get('1')); // str1
```

Se abbiamo un semplice oggetto, e vogliamo utilizzarlo per creare una `Map`, possiamo utilizzare un metodo integrato degli oggetti `Object.entries(obj)` ↗ il quale ritorna un array di coppie chiave/valore nello stesso formato.

Quindi possiamo creare una `Map` da un oggetto, così:

```
let obj = {
 name: "John",
 age: 30
};

let map = new Map(Object.entries(obj));

alert(map.get('name')); // John
```

In questo esempio, `Object.entries` ritorna un array di coppie chiave/valore: `[["name", "John"], ["age", 30]]`. Che è quello di cui `Map` ha bisogno.

### `Object.fromEntries: Object da Map`

Abbiamo appena visto come creare una `Map` partendo da un oggetto con `Object.entries(obj)`.

Esiste un metodo `Object.fromEntries` che fa esattamente l'opposto: dato un array di coppie `[key, value]`, ne crea un oggetto:

```
let prices = Object.fromEntries([
 ['banana', 1],
 ['orange', 2],
 ['meat', 4]
]);

// ora prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2
```

Possiamo utilizzare il metodo `Object.fromEntries` per ottenere un oggetto partendo da una `Map`.

Ad esempio, memorizziamo i dati in una `Map`, ma abbiamo bisogno di passarla ad un codice di terze parti che si aspetta un oggetto.

Quindi:

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('meat', 4);

let obj = Object.fromEntries(map.entries()); // costruisce un oggetto (*)

// fatto!
// obj = { banana: 1, orange: 2, meat: 4 }

alert(obj.orange); // 2
```

Una chiamata a `map.entries()` ritorna un array di coppie chiave/valore, esattamente nel formato richiesto da `Object.fromEntries`.

Possiamo rendere la riga (\*) ancora più corta:

```
let obj = Object.fromEntries(map); // omettendo .entries()
```

L'espressione è equivalente, poiché `Object.fromEntries` si aspetta di ricevere un oggetto iterabile come argomento. Non necessariamente un array. E l'iterazione standard per `Map` ritorna le stesse coppie chiave/valore di `map.entries()`. Quindi abbiamo ottenuto un oggetto con le stesse coppie chiave/valore della `map`.

## Set

Un `Set` è un tipo di collezione speciale – “set di valori” (senza chiavi), dove ogni valore può apparire una sola volta.

I suoi metodi principali sono:

- `new Set(iterable)` – crea il set, e se gli viene fornito un oggetto `iterabile` (solitamente un array), ne copia i valori nel set.
- `set.add(value)` – aggiunge un valore, ritorna il set.
- `set.delete(value)` – rimuove il valore, ritorna `true` se `value` esiste, altrimenti `false`.
- `set.has(value)` – ritorna `true` se il valore esiste nel set, altrimenti `false`.
- `set.clear()` – rimuove tutti i valori dal set.
- `set.size` – ritorna il numero dei valori contenuti.

La principale caratteristica dei set è che ripetute chiamate di `set.add(value)` con lo stesso valore non fanno nulla. Questo è il motivo per cui in un `Set` ogni valore può comparire una sola volta.

Ad esempio, abbiamo diversi arrivi di visitatori, e vorremmo ricordarli tutti. Ma visite ripetute dello stesso utente non dovrebbe portare a duplicati. Un visitatore deve essere conteggiato una volta sola.

`Set` è esattamente la struttura dati che fa al caso nostro:

```
let set = new Set();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

// visitatori, alcuni potrebbero tornare più volte
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);

// set mantiene solo valori unici
alert(set.size); // 3

for (let user of set) {
 alert(user.name); // John (poi Pete e Mary)
}
```

L'alternativa a `Set` potrebbe essere un array di visitatori, e un codice per verificare ogni inserimento ed evitare duplicati, utilizzando `arr.find ↗`. Ma la performance sarebbe molto inferiore, perché questo metodo attraversa tutto l'array per verificare ogni elemento. `Set` è ottimizzato internamente per il controllo di unicità.

## Iterare un Set

Possiamo iterare un set sia con `for .. of` che con `forEach`:

```
let set = new Set(["oranges", "apples", "bananas"]);

for (let value of set) alert(value);
```

```
// equivalente con forEach:
set.forEach((value, valueAgain, set) => {
 alert(value);
});
```

Da notare una cosa divertente. La funzione callback fornita a `forEach` ha 3 argomenti: un `value`, poi *lo stesso valore* `valueAgain`, e poi l'oggetto riferito da `this`. Proprio così, lo stesso valore appare due volte nella lista degli argomenti.

Questo accade per questioni di compatibilità con `Map`, in cui la funzione callback fornita al `forEach` possiede tre argomenti. E' un po' strano, ma in alcuni casi può aiutare rimpiazzare `Map` con `Set`, e vice versa.

Sono supportati anche i metodi di iterazione di `Map`:

- `set.keys()` – ritorna un oggetto per iterare sui valori,
- `set.values()` – lo stesso di `set.keys()`, per compatibilità con `Map`,
- `set.entries()` – ritorna un oggetto per iterare sulle voci `[value, value]`, esiste per compatibilità con `Map`.

## Riepilogo

`Map` è una collezione di valori identificati da chiave.

Metodi e proprietà:

- `new Map([iterable])` – crea la mappa, accetta un oggetto iterabile (opzionale, e.g. array) di coppie `[key, value]` per l'inizializzazione.
- `map.set(key, value)` – memorizza il valore con la chiave fornita.
- `map.get(key)` – ritorna il valore associato alla chiave, `undefined` se la `key` non è presente nella `Map`.
- `map.has(key)` – ritorna `true` se la `key` esiste, `false` altrimenti.
- `map.delete(key)` – rimuove il valore associato alla chiave.
- `map.clear()` – rimuove ogni elemento dalla mappa.
- `map.size` – ritorna il numero di elementi contenuti nella `map`.

Le differenze da un `Object` standard:

- Le chiavi possono essere di qualsiasi tipo, anche oggetti.
- Possiede metodi aggiuntivi, come la proprietà `size`.

`Set` è una collezione di valori unici.

Metodi e proprietà:

- `new Set([iterable])` – crea un set, accetta un oggetto iterabile (opzionale, e.g. array) per l'inizializzazione.
- `set.add(value)` – aggiunge un valore (non fa nulla nel caso in cui il valore sia già contenuto nel set), e ritorna il set.

- `set.delete(value)` – rimuove il valore, ritorna `true` se `value` esiste, `false` altrimenti.
- `set.has(value)` – ritorna `true` se il valore esiste nel set, `false` altrimenti.
- `set.clear()` – rimuove tutti i valori dal set.
- `set.size` – ritorna il numero di valori contenuti.

L'iterazione su `Map` e `Set` segue sempre l'ordine di inserimento, quindi possono essere definite delle collezioni ordinate; non è però possibile riordinare gli elementi oppure ottenere un valore tramite il suo indice.

## ✓ Esercizi

---

### Filtrare gli elementi dell'array unici

importanza: 5

Avete un array `arr`.

Create una funzione `unique(arr)` che ritorni un array con tutti gli elementi unici presi da `arr`.

Ad esempio:

```
function unique(arr) {
 /* your code */
}

let values = ["Hare", "Krishna", "Hare", "Krishna",
 "Krishna", "Krishna", "Hare", "Hare", ":-0"
];

alert(unique(values)); // Hare, Krishna, :-0
```

P.S. Qui vengono utilizzate stringhe, ma potrebbero essere valori di qualsiasi tipo.

P.P.S. utilizzate `Set` per memorizzare valori unici.

Apri una sandbox con i test. ↗

Alla soluzione

---

### Filtrare anagrammi

importanza: 4

Gli [anagrammi](#) ↗ sono parole che hanno le stesse lettere, ma in un ordine differente.

Ad esempio:

```
nap - pan
ear - are - era
cheaters - hectares - teachers
```

Scrivete una funzione `aclean(arr)` che ritorna un array ripulito dagli anagrammi.

Ad esempio:

```
let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"];
alert(aclean(arr)); // "nap,teachers,ear" or "PAN,cheaters,era"
```

Da ogni gruppo di anagrammi dovrebbe rimanere solamente una parola, non ha importanza quale.

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

## Chiavi iterabili

importanza: 5

Vorremmo avere un array di `map.keys()` in una variabile, quindi potergli applicare un metodo specifico degli arrays, ad esempio `.push`.

Ma c'è un problema:

```
let map = new Map();
map.set("name", "John");
let keys = map.keys();
// Error: keys.push is not a function
keys.push("more");
```

Perché? Come possiamo sistemare il codice per rendere `keys.push` funzionante?

[Alla soluzione](#)

## WeakMap e WeakSet

Come abbiamo già visto nel capitolo [Garbage collection \("Spazzatura"\)](#), il motore JavaScript mantiene un valore in memoria fino a che questo risulta accessibile (e potrebbe potenzialmente essere utilizzato).

Ad esempio:

```
let john = { name: "John" };
// l'oggetto è accessibile, john è un suo riferimento
// sovrascriviamo il riferimento
```

```
john = null;

// l'oggetto verrà rimosso dalla memoria
```

Solitamente, le proprietà di un oggetto o gli elementi di un array o di qualsiasi altra struttura dati vengono considerati accessibili fino a che questi rimangono mantenuti in memoria.

Ad esempio, se inseriamo un oggetto in un array, fino a che l'array rimane “vivo”, anche l'oggetto rimarrà in memoria, anche se non sono presenti riferimenti.

Come nell'esempio:

```
let john = { name: "John" };

let array = [john];

john = null; // sovrascriviamo il riferimento

// john è memorizzato all'interno dell'array
// quindi non verrà toccato dal garbage collector
// possiamo estrarlo tramite array[0]
```

O, se utilizziamo un oggetto come chiave in una `Map`, fino a che la `Map` esiste, anche l'oggetto esisterà. Occuperà memoria e non potrà essere ripulito dal garbage collector.

Ad esempio:

```
let john = { name: "John" };

let map = new Map();
map.set(john, "...");

john = null; // sovrascriviamo il riferimento

// john viene memorizzato all'interno di map,
// possiamo estrarlo utilizzando map.keys()
```

`WeakMap` è fondamentalmente diverso sotto questo aspetto. Infatti non previene la garbage-collection degli oggetti utilizzati come chiave.

Vediamo cosa significa questo, utilizzando degli esempi.

## WeakMap

La prima differenza tra `Map` e `WeakMap` è che le chiavi devono essere oggetti, non valori primitivi:

```
let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok"); // funziona
```

```
// non possiamo utilizzare una stringa come chiave
weakMap.set("test", "Whoops"); // Errore, perché "test" non è un oggetto
```

Ora, se utilizziamo un oggetto come chiave, e non ci sono altri riferimenti a quell'oggetto – questo verrà rimosso dalla memoria (e dalla map) automaticamente.

```
let john = { name: "John" };

let weakMap = new WeakMap();
weakMap.set(john, "...");

john = null; // sovrascriviamo il riferimento

// john è stato rimosso dalla memoria!
```

Confrontiamolo con l'esempio di `Map` visto sopra. Ora, se `john` esiste solo come chiave della `WeakMap` – verrà eliminato automaticamente dalla map (e anche dalla memoria).

`WeakMap` non supporta gli iteratori e i metodi `keys()`, `values()`, `entries()`, quindi non c'è alcun modo di ottenere tutte le chiavi o i valori tramite questi metodi.

`WeakMap` possiede solamente i seguenti metodi:

- `weakMap.get(key)`
- `weakMap.set(key, value)`
- `weakMap.delete(key)`
- `weakMap.has(key)`

Perché questa limitazione? Per ragioni tecniche. Se un oggetto ha perso tutti i riferimenti (come `john` nel codice sopra), allora verrà automaticamente eliminato. Ma tecnicamente non è specificato esattamente *quando avverrà la pulizia*.

Sarà il motore JavaScript a deciderlo. Potrebbe decidere di effettuare subito la pulizia della memoria oppure aspettare più oggetti per eliminarli in blocco. Quindi, tecnicamente il numero degli elementi di una `WeakMap` non è conosciuto. Il motore potrebbe già aver effettuato la pulizia oppure no, o averlo fatto solo parzialmente. Per questo motivo, i metodi che accedono a `WeakMap` per intero non sono supportati.

Dove potremmo avere bisogno di una struttura simile?

## Caso d'uso: dati aggiuntivi

Il principale campo di applicazione di `WeakMap` è quello di un *additional data storage*.

Se stiamo lavorando con un oggetto che “appartiene” ad un altro codice, magari una libreria di terze parti, e vogliamo memorizzare alcuni dati associati ad esso, che però dovrebbero esistere solamente finché l'oggetto esiste – allora una `WeakMap` è proprio ciò di cui abbiamo bisogno.

Inseriamo i dati in una `WeakMap`, utilizzando l'oggetto come chiave; quando l'oggetto verrà ripulito dal garbage collector, anche i dati associati verranno ripuliti.

```
weakMap.set(john, "secret documents");
// se john muore, i documenti segreti verranno distrutti automaticamente
```

Proviamo a guardare un esempio.

Immaginiamo di avere del codice che tiene nota del numero di visite per ogni utente. L'informazione viene memorizzata in un map: l'utente è la chiave, mentre il conteggio delle visite è il valore. Quando l'utente esce, vogliamo smettere di mantenere in memoria il conteggio delle visite.

Qui vediamo un esempio di conteggio utilizzando `Map`:

```
// visitsCount.js
let visitsCountMap = new Map(); // map: user => conteggio visite

// incrementa il conteggio delle visite
function countUser(user) {
 let count = visitsCountMap.get(user) || 0;
 visitsCountMap.set(user, count + 1);
}
```

E qui abbiamo un'altra porzione di codice, magari in un altro file, che la utilizza:

```
// main.js
let john = { name: "John" };

countUser(john); // conta le sue visite

// più tardi John se ne va
john = null;
```

Ora, l'oggetto `john` dovrebbe essere ripulito dal garbage collector, ma rimane in memoria, in quanto chiave in `visitsCountMap`.

Dobbiamo ripulire `visitsCountMap` quando rimuoviamo l'utente, altrimenti continuerà a crescere nella memoria indefinitamente. Una pulizia di questo tipo potrebbe essere complessa in architetture più elaborate.

Possiamo risolvere questo problema utilizzando una `WeakMap`:

```
// visitsCount.js
let visitsCountMap = new WeakMap(); // weakmap: user => conteggio visite

// incrementa il conteggio delle visite
function countUser(user) {
 let count = visitsCountMap.get(user) || 0;
 visitsCountMap.set(user, count + 1);
}
```

Ora non dobbiamo più ripulire `visitsCountMap`. Una volta che `john` non sarà più accessibile, ad eccezione che come chiave della `WeakMap`, verrà rimosso dalla memoria, insieme a tutte le informazioni associate contenute nella `WeakMap`.

## Caso d'uso: caching

Un altro caso d'uso comune è il caching. Possiamo memorizzare i risultati di una funzione, così che le successive chiamate alla funzione possano riutilizzarli.

Per fare questo possiamo utilizzare una `Map` (non la scelta ottimale):

```
// cache.js
let cache = new Map();

// calcola e memorizza il risultato
function process(obj) {
 if (!cache.has(obj)) {
 let result = /* calcola il risultato per */ obj;

 cache.set(obj, result);
 }

 return cache.get(obj);
}

// Ora utilizziamo process() in un altro file:

// main.js
let obj = {/* ipotizziamo di avere un oggetto */};

let result1 = process(obj); // calcolato

// ...più tardi, da un'altra parte del codice...
let result2 = process(obj); // prendiamo il risultato dalla cache

// ...più avanti, quando non abbiamo più bisogno dell'oggetto:
obj = null;

alert(cache.size); // 1 (Ouch! L'oggetto è ancora in cache, sta occupando memoria!)
```

Per chiamate multiple di `process(obj)` con lo stesso oggetto, il risultato viene calcolato solamente la prima volta, le successive chiamate lo prenderanno dalla `cache`. Il lato negativo è che dobbiamo ricordarci di pulire la `cache` quando non è più necessaria.

Se sostituiamo `Map` con `WeakMap`, il problema si risolve. I risultati in cache vengono automaticamente rimossi una volta che l'oggetto viene ripulito dal garbage collector.

```
// cache.js
let cache = new WeakMap();

// calcola e memorizza il risultato
function process(obj) {
 if (!cache.has(obj)) {
 let result = /* calcola il risultato per */ obj;

 cache.set(obj, result);
 }

 return cache.get(obj);
}
```

```
// main.js
let obj = /* un oggetto */;

let result1 = process(obj);
let result2 = process(obj);

// ...più tardi, quando non abbiamo più bisogno dell'oggetto
obj = null;

// Non possiamo ottenere la dimensione della cache, poiché è una WeakMap,
// ma è 0 oppure lo sarà presto
// Quando un oggetto viene ripulito dal garbage collector, anche i dati associati vengono ripuliti
```

## WeakSet

`WeakSet` si comporta in maniera simile:

- E' analogo a `Set`, ma possiamo aggiungere solamente oggetti a `WeakSet` (non primitivi).
- Un oggetto esiste in un set solamente finché rimane accessibile in un altro punto del codice.
- Come `Set`, supporta `add`, `has` e `delete`, ma non `size`, `keys()` e nemmeno gli iteratori.

Il fatto che sia "weak" la rende utile come spazio di archiviazione aggiuntivo. Non per dati arbitrari, ma piuttosto per questioni di tipo "si/no". Il fatto di appartenere ad un `WeakSet` può significare qualcosa sull'oggetto.

Ad esempio, possiamo aggiungere gli utenti ad un `WeakSet` per tenere traccia di chi ha visitato il nostro sito:

```
let visitedSet = new WeakSet();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

visitedSet.add(john); // John ci ha visitato
visitedSet.add(pete); // Poi Pete
visitedSet.add(john); // John di nuovo

// visitedSet ha 2 utenti ora

// controlliamo se John ci ha visitato
alert(visitedSet.has(john)); // true

// controlliamo se Mary ci ha visitato
alert(visitedSet.has(mary)); // false

john = null;

// visitedSet verrà ripulito automaticamente
```

La maggior limitazione di `WeakMap` e `WeakSet` è l'assenza di iteratori, e la mancanza della possibilità di ottenere tutti gli elementi contenuti. Potrebbe sembrare un inconveniente, ma non

vieta a `WeakMap`/`WeakSet` di compiere il proprio lavoro – essere una struttura “addizionale” per memorizzare informazioni relative a dati memorizzati in un altro posto.

## Riepilogo

`WeakMap` è una collezione simile a `Map`, ma permette di utilizzare solamente oggetti come chiavi; inoltre, la rimozione di un oggetto rimuove anche il valore associato.

`WeakSet` è una collezione simile a `Set`, che memorizza solamente oggetti, e li rimuove completamente una volta che diventano inaccessibili.

Il loro principale vantaggio è che possiedono un riferimento debole agli oggetti, in questo modo possono essere facilmente ripuliti dal garbage collector.

Il lato negativo è di non poter utilizzare `clear`, `size`, `keys`, `values` ...

`WeakMap` e `WeakSet` vengono utilizzati come strutture dati “secondarie” in aggiunta a quelle “principali”. Una volta che l’oggetto viene rimosso dalla struttura dati “principale”, se l’unico riferimento rimasto è una chiave di `WeakMap` o `WeakSet`, allora verrà rimosso.

## ✓ Esercizi

### Memorizzare le bandiere non visualizzate

importanza: 5

Abbiamo un array di messaggi:

```
let messages = [
 {text: "Hello", from: "John"},
 {text: "How goes?", from: "John"},
 {text: "See you soon", from: "Alice"}
];
```

Il vostro codice vi può accedere, ma i messaggi sono gestiti dal codice di qualcun altro. Vengono aggiunti nuovi messaggi, quelli vecchi vengono rimossi, e voi non avete modo di sapere quando ciò accade.

Ora, quale struttura dati potresti utilizzare per memorizzare quali messaggi “sono stati letti”? La struttura deve calzare bene al problema, e rispondere alla domanda “è stato letto?”.

P.S. Quando un messaggio viene rimosso da `messages`, dovrebbe essere rimosso anche dalla vostra struttura.

P.P.S. Non dovremmo modificare l’oggetto `message`. Poiché se viene gestito dal codice di qualcun altro, aggiungere nuove proprietà potrebbe avere conseguenze disastrose.

[Alla soluzione](#)

### Memorizzare le date di lettura

importanza: 5

Abbiamo un array di messaggi come nel [compito precedente](#). La situazione è simile.

```
let messages = [
 {text: "Hello", from: "John"},
 {text: "How goes?", from: "John"},
 {text: "See you soon", from: "Alice"}
];
```

Ora la domanda è: quale struttura di dati converrebbe utilizzare per memorizzare l'informazione: "quando è stato letto il messaggio?".

Nel compito precedente la necessità era semplicemente di memorizzare la lettura del messaggio. Ora abbiamo bisogno di memorizzare anche la data; anche in questo caso, se il messaggio viene eliminato questa dovrebbe sparire.

[Alla soluzione](#)

## Object.keys, values, entries

Facciamo un passo oltre le strutture dati in sé e discutiamo dei metodi di iterazione su di esse.

Nel capitolo precedente abbiamo visto i metodi `map.keys()`, `map.values()`, `map.entries()`.

Questi sono dei metodi generici, c'è un comune accordo sul loro utilizzo per le strutture dati. Se dovessimo mai creare una nostra struttura dati personale, dovremmo implementare anche questi metodi.

Vengono supportati da:

- `Map`
- `Set`
- `Array`

Anche gli oggetti supportano dei metodi simili, ma la loro sintassi è leggermente differente.

## Object.keys, values, entries

Per i semplici oggetti, sono disponibili i seguenti metodi:

- [Object.keys\(obj\)](#) ↗ – ritorna un array di chiavi.
- [Object.values\(obj\)](#) ↗ – ritorna un array di valori.
- [Object.entries\(obj\)](#) ↗ – ritorna un array di coppie `[key, value]`.

Da notare le differenze (confrontandoli con quelli delle map):

	Map	Object
Chiamata	<code>map.keys()</code>	<code>Object.keys(obj)</code> , non <code>obj.keys()</code>
Valore di ritorno	oggetti iterabile	Array

La prima differenza è che dobbiamo chiamare `Object.keys(obj)`, non `obj.keys()`.

Perché? La principale motivazione è la flessibilità. Ricordate, gli oggetti sono la base di tutte le strutture complesse in JavaScript. Quindi potremmo avere un nostro oggetto come `order` che implementa il proprio metodo `order.values()`. E potremmo ancora chiamare `Object.values(order)`.

La seconda differenza è che i metodi `Object.*` ritornano un array, non un oggetto iterabile. Questo comportamento è dovuto a ragioni storiche.

Ad esempio:

```
let user = {
 name: "John",
 age: 30
};
```

- `Object.keys(user) = ["name", "age"]`
- `Object.values(user) = ["John", 30]`
- `Object.entries(user) = [ ["name", "John"], ["age", 30] ]`

Qui un esempio di utilizzo di `Object.values` per eseguire cicli sui valori delle proprietà:

```
let user = {
 name: "John",
 age: 30
};

// ciclo sui valori
for (let value of Object.values(user)) {
 alert(value); // John, poi 30
}
```

### ⚠️ `Object.keys/values/entries` ignorano le proprietà di tipo symbol

Proprio come nel caso del ciclo `for .. in`, questi metodi ignorano le proprietà che utilizzano `Symbol(...)` come chiave.

Solitamente questo è un vantaggio. Ma se volessimo ottenere anche le chiavi di tipo symbol, esiste un secondo metodo [Object.getOwnPropertySymbols](#) ↗ che ritorna un array di chiavi di tipo symbol. Invece, il metodo [Reflect.ownKeys\(obj\)](#) ↗ ritorna *tutte* le chiavi.

## Trasformare gli oggetti

Per gli oggetti mancano molti metodi che sono invece presenti per gli array, ad esempio `map`, `filter` e molti altri.

Se volessimo comunque applicarli, allora possiamo utilizzare `Object.entries` seguito da `Object.fromEntries`:

1. Applichiamo `Object.entries(obj)` per ottenere un array di coppie chiave/valore da `obj`.
2. Applichiamo il metodo, ad esempio `map`.
3. Applichiamo `Object.fromEntries(array)` all'array risultante per ottenere nuovamente un oggetto.

Ad esempio, se abbiamo un oggetto di prezzi che vogliamo raddoppiare:

```
let prices = {
 banana: 1,
 orange: 2,
 meat: 4,
};

let doublePrices = Object.fromEntries(
 // converte ad array, chiama map, e successivamente fromEntries ci ritorna l'oggetto
 Object.entries(prices).map(([key, value]) => [key, value * 2])
);

alert(doublePrices.meat); // 8
```

Ad un primo sguardo potrebbe risultare complesso, ma diventa molto più familiare dopo un paio di utilizzi. In questo modo possono essere create potenti catene per la trasformazione.

## ✓ Esercizi

---

### Sommare le proprietà

importanza: 5

Abbiamo un oggetto `salaries` con un numero arbitrario di salari.

Scrivete la funzione `sumSalaries(salaries)` che ritorna la somma di tutti i salari utilizzando `Object.values` e il ciclo `for..of`.

Se `salaries` è vuoto, allora il risultato deve essere `0`.

Ad esempio:

```
let salaries = {
 "John": 100,
 "Pete": 300,
 "Mary": 250
};

alert(sumSalaries(salaries)); // 650
```

Apri una sandbox con i test. [↗](#)

[Alla soluzione](#)

## Conteggio proprietà

importanza: 5

Scrivete una funzione `count(obj)` che ritorna il numero di proprietà dell'oggetto:

```
let user = {
 name: 'John',
 age: 30
};

alert(count(user)); // 2
```

Cercate di rendere il codice il più breve possibile.

P.S. Ignorate le proprietà di tipo symbol, tenete conto solamente di quelle “regolari”.

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

## Assegnamento di destrutturazione

Le due strutture dati più utilizzate in JavaScript sono `Object` e `Array`.

- Gli oggetti ci consentono di creare un'unica entità che memorizza elementi nel formato chiave-valore
- Gli array ci consentono di raccogliere elementi in elenchi ordinati.

A volte, quando li passiamo ad una funzione, potrebbe non essere necessario tutto l'oggetto/array, ma solo una parte di esso.

*L'assegnamento di destrutturazione (Destructuring assignment)* è una speciale sintassi che ci consente di “spacchettare” oggetti o array in gruppi di variabili; questo a volte risulta molto conveniente.

La destrutturazione funziona alla grande anche con funzioni complesse che hanno molti parametri, valori predefiniti e così via. Presto lo vedremo.

### Destrutturazione di un array

Ecco un esempio di come un array viene destrutturato in variabili:

```
// abbiamo un array con nome e cognome
let arr = ["John", "Smith"]

// assegnamento di destrutturazione
// imposta firstName = arr[0]
// e surname = arr[1]
let [firstName, surname] = arr;
```

```
alert(firstName); // John
alert(surname); // Smith
```

Ora possiamo lavorare con le variabili invece che con gli elementi dell'array.

Risulta utilissima se combinata con `split` o altri metodi che ritornano un array:

```
let [firstName, surname] = "John Smith".split(' ');\nalert(firstName); // John\nalert(surname); // Smith
```

Come puoi vedere, la sintassi è semplice. Ci sono però molti dettagli peculiari. Vediamo altri esempi, per capirlo meglio.

#### **i** “Destruzione” non significa “distruzione”.

Viene chiamato “assegnamento di destrutturazione” perché “destruttura” copiando gli elementi all’interno di variabili. Ma l’array in sé non viene modificato.

E’ solo un modo breve per scrivere:

```
// let [firstName, surname] = arr;\nlet firstName = arr[0];\nlet surname = arr[1];
```

#### **i** Ignora gli elementi usando la virgola

Gli elementi indesiderati dell’array possono essere ignorati tramite una virgola aggiuntiva:

```
// il secondo elemento non è necessario\nlet [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];\n\nalert(title); // Consul
```

Nel codice sopra, il secondo elemento viene saltato, il terzo viene assegnato a `title`, il resto degli elementi vengono ignorati (visto che per loro non ci sono variabili).

#### **i** Funziona con qualsiasi iterabile alla destra

... In realtà, possiamo utilizzarlo con qualsiasi iterabile, non solamente con un array:

```
let [a, b, c] = "abc"; // ["a", "b", "c"]\nlet [one, two, three] = new Set([1, 2, 3]);
```

Funziona, perché internamente un’assegnazione di destrutturazione lavora iterando sul valore a destra. E’ una specie di “zucchero sintattico” per chiamare `for .. of` sul valore a destra di `=`, assegnandone i valori.

### **i Assegna a qualsiasi cosa ci sia dalla parte sinistra**

Possiamo inserire qualsiasi cosa “assegnabile” a sinistra.

Ad esempio, la proprietà di un oggetto:

```
let user = {};
[user.name, user.surname] = "John Smith".split(' ');
alert(user.name); // John
alert(user.surname); // Smith
```

### **i Eseguire cicli con .entries()**

Nel capitolo precedente abbiamo visto il metodo [Object.entries\(obj\)](#).

Possiamo utilizzarlo con la destrutturazione per eseguire cicli sulle coppie chiave/valore di un oggetto:

```
let user = {
 name: "John",
 age: 30
};

// ciclo su chiavi/valori
for (let [key, value] of Object.entries(user)) {
 alert(`[${key}]:${value}`); // name:John, poi age:30
}
```

Un codice simile usando `Map` è più semplice, visto che è iterabile:

```
let user = new Map();
user.set("name", "John");
user.set("age", "30");

// Map itera le coppie [key, value], molto comodo per la destrutturazione
for (let [key, value] of user) {
 alert(`[${key}]:${value}`); // name:John, then age:30
}
```

## Il trucco dello scambio di variabili

c'è un trucco molto conosciuto per scambiare i valori di due variabili usando l'assegnamento di destrutturazione:

```
let guest = "Jane";
let admin = "Pete";

// Scambio dei valori: rende guest=Pete, admin=Jane
[guest, admin] = [admin, guest];

alert(`${guest} ${admin}`); // Pete Jane (scambiati con successo!)
```

Nell'esempio creiamo un array temporaneo con due variabili e lo destrutturiamo immediatamente invertendo l'ordine delle variabili. Nello stesso modo potremmo scambiare i valori di più variabili.

## L'operatore rest '...'

Di solito, se l'array è più lungo della lista a sinistra, gli elementi in eccesso vengono ignorati.

Ad esempio, qui vengono presi solo i primi 2 elementi, i restanti vengono semplicemente ignorati:

```
let [name1, name2] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert(name1); // Julius
alert(name2); // Caesar

// Gli elementi successivi non vengono assegnati
```

Se vogliamo ottenere anche tutto ciò che segue, possiamo aggiungere un altro parametro che raccoglie "il resto" utilizzando tre punti " . . . " :

```
let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

// rest è un array con gli elementi a destra, partendo dal terzo
alert(rest[0]); // Consul
alert(rest[1]); // of the Roman Republic
alert(rest.length); // 2
```

La variabile `rest` è un array con i valori rimanenti dell'array a destra.

Possiamo utilizzare qualsiasi altro nome di variabile al posto di `rest`; è sufficiente accertarsi di inserire i tre punti prima del nome e di posizionarlo alla fine nell'assegnamento di destrutturazione.

```
let [name1, name2, ...titles] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];
// now titles = ["Consul", "of the Roman Republic"]
```

## Valori di default

Se ci sono meno elementi nell'array delle variabili da assegnare, non ci sarà alcun errore. I valori assenti vengono considerati `undefined`:

```
let [firstName, surname] = [];

alert(firstName); // undefined
alert(surname); // undefined
```

Se volessimo assegnare un nostro valore di “default”, potremmo indicarlo con la sintassi := :

```
// valori di default
let [name = "Guest", surname = "Anonymous"] = ["Julius"];

alert(name); // Julius (dall'array)
alert(surname); // Anonymous (valore di default)
```

I valori di default possono essere anche espressioni complesse o chiamate a funzione. Verranno presi in considerazione solamente se non verrà fornito alcun valore.

Ad esempio, qui usiamo la funzione `prompt` per due defaults:

```
// viene eseguito solo il prompt per il cognome
let [name = prompt('name?'), surname = prompt('surname?')] = ["Julius"];

alert(name); // Julius (dall'array)
alert(surname); // qualsiasi cosa provenga dal prompt
```

Attenzione: la funzione `prompt` verrà eseguita solo per il valore mancante (`surname`).

## Destrutturazione di oggetti

L’assegnamento di destrutturazione funziona anche con gli oggetti.

La sintassi è:

```
let {var1, var2} = {var1:..., var2:...}
```

Abbiamo un oggetto alla destra dell’assegnazione, che vogliamo dividere in variabili. Nel lato sinistro abbiamo un “pattern” di proprietà corrispondenti. In questo semplice caso, abbiamo una lista di variabili raggruppate tra parentesi `{ . . . }`.

Ad esempio:

```
let options = {
 title: "Menu",
 width: 100,
 height: 200
};

let {title, width, height} = options;

alert(title); // Menu
```

```
alert(width); // 100
alert(height); // 200
```

Le proprietà `options.title`, `options.width` e `options.height` vengono assegnate alle variabili corrispondenti.

L'ordine non ha importanza. Questo codice funzionerebbe comunque:

```
// cambiato l'ordine delle proprietà in let {...}
let {height, width, title} = { title: "Menu", height: 200, width: 100 }
```

Il pattern a sinistra potrebbe essere anche più complesso e specificare una mappatura tra proprietà e variabili.

Se volessimo assegnare una proprietà ad una variabile con un altro nome, ad esempio la proprietà `options.width` ad una variabile chiamata `w`, allora possiamo specificarlo con i due punti:

```
let options = {
 title: "Menu",
 width: 100,
 height: 200
};

// { sourceProperty: targetVariable }
let {width: w, height: h, title} = options;

// width -> w
// height -> h
// title -> title

alert(title); // Menu
alert(w); // 100
alert(h); // 200
```

I due punti specificano “cosa : va dove”. Nell'esempio sopra la proprietà `width` va in `w`, la proprietà `height` va in `h`, e `title` viene assegnata ad una variabile con lo stesso nome.

Per delle potenziali proprietà mancanti possiamo impostare dei valori di default utilizzando `"= "`, come nell'esempio:

```
let options = {
 title: "Menu"
};

let {width = 100, height = 200, title} = options;

alert(title); // Menu
alert(width); // 100
alert(height); // 200
```

Proprio come nel caso degli array o dei parametri di funzione, i valori di default possono essere espressioni più complesse o chiamate a funzioni. Questi verranno valutati solo nel caso in cui il valore non verrà fornito.

Il codice richiederà tramite `prompt` la `width` (larghezza), ma non il `title` (titolo):

```
let options = {
 title: "Menu"
};

let {width = prompt("width?"), title = prompt("title?")} = options;

alert(title); // Menu
alert(width); // (qualsiasi cosa arrivi dal prompt)
```

Possiamo anche combinare entrambi, i due punti e l'uguaglianza:

```
let options = {
 title: "Menu"
};

let {width: w = 100, height: h = 200, title} = options;

alert(title); // Menu
alert(w); // 100
alert(h); // 200
```

Se abbiamo un oggetto complesso con molte proprietà, possiamo estrarre solamente ciò che ci serve:

```
let options = {
 title: "Menu",
 width: 100,
 height: 200
};

// estraiamo solamente title come proprietà
let { title } = options;

alert(title); // Menu
```

## Il modello rest “...”

Cosa succede se l'oggetto possiede più proprietà delle variabili da noi fornite? Possiamo prenderne solamente alcune ed assegnare tutto ciò che avanza da un'altra parte?

La specifica per l'utilizzo dell'operatore rest `...` fa quasi parte dello standard, ma molti browser non lo supportano ancora.

Appare così:

```
let options = {
 title: "Menu",
 height: 200,
```

```
width: 100
};

// title = proprietà con il titolo
// rest = oggetto con il resto dei parametri
let {title, ...rest} = options;

// ora title="Menu", rest={height: 200, width: 100}
alert(rest.height); // 200
alert(rest.width); // 100
```

### Catturare senza let

Negli esempi sopra le variabili vengono dichiarate appena prima di essere assegnate: `let {...} = {...}`. Ovviamente, potremmo anche utilizzare delle variabili già esistenti. Ma c'è un tranello.

Questo non funzionerebbe:

```
let title, width, height;

// errore in questa riga
{title, width, height} = {title: "Menu", width: 200, height: 100};
```

Il problema è che JavaScript tratta `{...}` come un blocco di codice. Questo blocco di codice può essere utilizzato per raggruppare istruzioni, come nell'esempio:

```
{
 // un blocco di codice
 let message = "Hello";
 // ...
 alert(message);
}
```

Per informare JavaScript che non ci troviamo in un blocco di codice, possiamo raggruppare l'intera assegnazione tra parentesi `(...)`:

```
let title, width, height;

// ora funziona
({title, width, height} = {title: "Menu", width: 200, height: 100});

alert(title); // Menu
```

## Destrutturazione annidata

Se un oggetto o un array contiene altri oggetti o array, possiamo utilizzare sequenze (pattern) di estrazione più complesse per andare più in profondità con l'estrazione.

Nel codice sotto `options` possiede un ulteriore oggetto nella proprietà `size` ed un array nella proprietà `items`. Il pattern alla sinistra dell'assegnazione ha la stessa struttura:

```
let options = {
 size: {
 width: 100,
 height: 200
 },
 items: ["Cake", "Donut"],
 extra: true // qualche extra che non destruttureremo
};

// destructuring assignment split in multiple lines for clarity
let {
 size: { // mettiamo size qui
 width,
 height
 },
 items: [item1, item2], // assegniamo gli items qui
 title = "Menu" // non presente nell'oggetto (viene utilizzato il valore di default)
} = options;

alert(title); // Menu
alert(width); // 100
alert(height); // 200
alert(item1); // Cake
alert(item2); // Donut
```

L'intero oggetto `options` ad eccezione di `extra`, il quale non viene menzionato, viene assegnato alle corrispondenti variabili.

```
let {
 size: {
 width,
 height
 },
 items: [item1, item2],
 title = "Menu"
}

let options = {
 size: {
 width: 100,
 height: 200
 },
 items: ["Cake", "Donut"],
 extra: true
}
```

Infine, abbiamo `width`, `height`, `item1`, `item2` e `title` che assumono il valore di default.

Nota che non ci sono variabili per `size` e `items`: prendiamo invece il loro contenuto. Questo accade spesso con l'assegnamento di destrutturazione. Abbiamo un oggetto complesso e vogliamo estrarre solamente ciò di cui abbiamo bisogno.

## Parametri di funzione intelligenti

Ci sono casi in cui una funzione può accettare più parametri, molti dei quali opzionali. Questo è vero specialmente per le interfacce utente. Immaginate una funzione che crea un menu. Può avere una larghezza, un'altezza, un titolo, una lista di elementi e molto altro.

Vediamo un pessimo modo per scrivere questo tipo di funzioni:

```
function showMenu(title = "Untitled", width = 200, height = 100, items = []) {
```

```
// ...
}
```

Nella vita reale, il problema è ricordarsi l'ordine degli argomenti. Solitamente gli IDE ci aiutano in questo, specialmente se il codice è ben documentato, eppure... Un ulteriore problema è quello di chiamare una funzione nel caso in cui molti parametri ci vadano bene di default.

Come qui?

```
// undefined where default values are fine
showMenu("My Menu", undefined, undefined, ["Item1", "Item2"])
```

E' brutto a vedersi. E diventa illeggibile quando il numero di parametri aumenta.

La destrutturazione ci viene in soccorso!

Possiamo passare i parametri come un oggetto, e la funzione immediatamente lo destrutturerà in variabili:

```
// passiamo l'oggetto alla funzione
let options = {
 title: "My menu",
 items: ["Item1", "Item2"]
};

// ...e immediatamente lo distribuisce alle variabili
function showMenu({title = "Untitled", width = 200, height = 100, items = []}) {
 // title, items - presi da options,
 // width, height - valori di default
 alert(`#${title} ${width} ${height}`); // My Menu 200 100
 alert(items); // Item1, Item2
}

showMenu(options);
```

Possiamo anche utilizzare una destrutturazione più complessa con oggetti annidati e diverse mappature:

```
let options = {
 title: "My menu",
 items: ["Item1", "Item2"]
};

function showMenu({
 title = "Untitled",
 width: w = 100, // width va su w
 height: h = 200, // height va su h
 items: [item1, item2] // il primo elemento va su item1, il secondo su item2
}) {
 alert(`#${title} ${w} ${h}`); // My Menu 100 200
 alert(item1); // Item1
 alert(item2); // Item2
}
```

```
showMenu(options);
```

La sintassi è la stessa dell'assegnamento di destrutturazione:

```
function({
 incomingProperty: varName = defaultValue
 ...
})
```

Da notare che la destrutturazione presuppone che `showMenu()` abbia un argomento. Se vogliamo tutti i valori di default, allora dovremmo specificare un oggetto vuoto:

```
showMenu({}); // ok, tutti i valori sono di default
showMenu(); // questo darà errore
```

Possiamo farlo ponendo `{}` come valore di default per l'intera destrutturazione:

```
// parametri semplificati per chiarezza
function showMenu({ title = "Menu", width = 100, height = 200 } = {}) {
 alert(` ${title} ${width} ${height}`);
}

showMenu(); // Menu 100 200
```

Nel codice sopra, l'oggetto degli argomenti è `{}` di default, quindi ci sarà sempre qualcosa da destrutturare.

## Riepilogo

- L'assegnamento di destrutturazione ci consente di mappare un oggetto o un array passandolo a variabili.
- La sintassi per gli oggetti:

```
let {prop : varName = default, ...rest} = object
```

Questo significa che la proprietà `prop` dovrebbe andare nella variabile `varName` e, se non esiste alcuna proprietà, allora verrà utilizzato il valore di `default`.

- La sintassi per gli array:

```
let [item1 = default, item2, ...rest] = array
```

Il primo elemento va in `item1`; il secondo va in `item2`, tutti gli altri finiscono nell'array `rest`.

- Per casi più complessi, la parte sinistra deve possedere la stessa struttura di quella destra.

## ✓ Esercizi

---

### Assegnamento di destrutturazione

importanza: 5

Abbiamo un oggetto:

```
let user = {
 name: "John",
 years: 30
};
```

Scrivete l'assegnamento di destrutturazione che legge:

- la proprietà `name` nella variabile `name`.
- la proprietà `years` nella variabile `age`.
- la proprietà `isAdmin` nella variabile `isAdmin` (falsa se assente)

I valori dopo l'assegnazione dovrebbero essere:

```
let user = { name: "John", years: 30 };

// il tuo codice a sinistra:
// ... = user

alert(name); // John
alert(age); // 30
alert(isAdmin); // false
```

[Alla soluzione](#)

---

### Il salario massimo

importanza: 5

Abbiamo un oggetto `salaries`:

```
let salaries = {
 "John": 100,
 "Pete": 300,
 "Mary": 250
};
```

Create la funzione `topSalary(salaries)` che ritorna il nome della persona con il salario maggiore.

- Se `salaries` è vuoto, dovrebbe ritornare `null`.
- Se ci sono più persone con lo stesso salario massimo, ritornatene una.

P.S. utilizzate `Object.entries` e la destrutturazione per iterare sulle coppie chiave/valore.

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

## Date e time

Ora analizziamo un nuovo oggetto integrato: `Date` ↗. Esso memorizza la data e l'ora e fornisce dei metodi utili per il trattamento di queste.

Ad esempio, possiamo utilizzarlo per memorizzare modifiche su orari, o per misurare il tempo, o solamente per ottenere l'informazione della data corrente.

### Creazione

Per creare un nuovo oggetto `Date`, chiamiamo `new Date()` con uno dei seguenti argomenti:

`new Date()`

Senza argomenti – crea un oggetto `Date` con la data e l'ora corrente:

```
let now = new Date();
alert(now); // mostra l'attuale data/tempo
```

`new Date(milliseconds)`

Crea un oggetto `Date` con l'ora impostata al numero di millisecondi trascorsi dal 1 Gennaio 1970 UTC+0.

```
// 0 significa 01.01.1970 UTC+0
let Jan01_1970 = new Date(0);
alert(Jan01_1970);

// ora vengono aggiunte 24 ore, si ha 02.01.1970 UTC+0
let Jan02_1970 = new Date(24 * 3600 * 1000);
alert(Jan02_1970);
```

Il numero di millisecondi passati da questa data vengono detti *timestamp*.

E' un modo semplice di rappresentare una data. Possiamo sempre creare una data a partire da un timestamp utilizzando `new Date(timestamp)`, o possiamo convertire un oggetto `Date` esistente utilizzando il metodo `date.getTime()` (vedi sotto).

Le date prima del 01.01.1970 hanno timestamps negativi, ad esempio:

```
// 31 Dec 1969
let Dec31_1969 = new Date(-24 * 3600 * 1000);
alert(Dec31_1969);
```

## `new Date(datestring)`

Se viene fornito un solo argomento, ed è una stringa, allora viene analizzato tramite l'algoritmo `Date.parse` (che vedremo tra poco).

```
let date = new Date("2017-01-26");
alert(date);
// Il tempo non è specificato, quindi si da per scontato che sia mezzanotte GMT e viene
// corretto a seconda della timezone dove il codice viene eseguito
// Quindi il risultato potrebbe essere
// Thu Jan 26 2017 11:00:00 GMT+1100 (Australian Eastern Daylight Time)
// o
// Wed Jan 25 2017 16:00:00 GMT-0800 (Pacific Standard Time)
```

## `new Date(year, month, date, hours, minutes, seconds, ms)`

Crea la data con le informazioni fornite. Solo i primi due argomenti sono obbligatori.

Nota:

- Il campo `year` deve essere composto da 4 cifre: `2013` va bene, `98` non è corretto.
- Il numero `month` inizia da `0` (Gennaio), fino a `11` (Dicembre).
- Il parametro `date` rappresenta il giorno del mese, se non viene fornito il valore di default è `1`.
- Se non vengono forniti `hours/minutes/seconds/ms`, il valore di default è `0`.

Ad esempio:

```
new Date(2011, 0, 1, 0, 0, 0, 0); // 1 Gennaio 2011, 00:00:00
new Date(2011, 0, 1); // // lo stesso, le ore ecc sono 0 di default
```

La precisione minima è 1 ms (1/1000 sec):

```
let date = new Date(2011, 0, 1, 2, 3, 4, 567);
alert(date); // 1.01.2011, 02:03:04.567
```

## Accedere ai componenti di date

Ci sono diversi metodi per poter accedere a 'year', 'month' e agli altri parametri dell'oggetto `Date`. Per ricordarli meglio li divideremo in categorie.

### `getFullYear()` ↗

Fornisce il valore di 'year' (anno), (4 cifre).

### `getMonth()` ↗

Fornisce il valore di 'month' (mese), **da 0 a 11**.

### `getDate()` ↗

Fornisce il giorno del mese, da 1 a 31, il nome del metodo potrebbe confondere.

## [getHours\(\)](#) , [getMinutes\(\)](#) , [getSeconds\(\)](#) , [getMilliseconds\(\)](#)

Fornisce il valore del corrispettivo parametro.

### **⚠️ Not `getYear()`, but `getFullYear()`**

Molti motori JavaScript implementano una versione non standard del metodo `getYear()`, che in alcuni casi ritorna un valore a 2 cifre. Per questo è un metodo deprecato ed è sconsigliato utilizzarlo. In sostituzione esiste un metodo più completo, `getFullYear()`.

In più potremmo anche prelevare il giorno della settimana:

## [getDay\(\)](#)

Restituisce il giorno della settimana, da `0` (Domenica) a `6` (Sabato). Il primo giorno è sempre Domenica; in alcuni stati non è così, ma non può essere modificato.

**Tutti i metodi sopra ritornano componenti relativi all'orario locale.**

Esistono anche le controparti UTC, che ritornano giorno, mese, anno e molto altro per la zona temporale UTC+0: [getUTCFullYear\(\)](#) , [getUTCMonth\(\)](#) , [getUTCDay\(\)](#) . E' sufficiente inserire "UTC" appena dopo "get".

Se il vostro orario locale è scostato dal UTC, allora il codice sotto potrebbe mostrare orari differenti:

```
// data corrente
let date = new Date();

// l'ora nella tua time zone corrente
alert(date.getHours());

// L'ora in UTC+0 time zone (l'orario di Londra senza l'ora legale)
alert(date.getUTCHours());
```

Oltre ai metodi forniti, ce ne sono altri due speciali, che non possiedono la variante UTC:

## [getTime\(\)](#)

Ritorna il timestamp della data – il numero di millisecondi trascorsi dal 1 Gennaio 1970 in UTC+0.

## [getTimezoneOffset\(\)](#)

Ritorna la differenza tra UTC e l'orario locale, in minuti:

```
// se sei in una timezone UTC-1, ritorna 60
// se sei in una timezone UTC+3, ritorna -180
alert(new Date().getTimezoneOffset());
```

## Impostare i componenti di date

I seguenti metodi consentono di impostare i componenti data/tempo:

- `setFullYear(year, [month], [date])`

- `setMonth(month, [date])` ↗
- `setDate(date)` ↗
- `setHours(hour, [min], [sec], [ms])` ↗
- `setMinutes(min, [sec], [ms])` ↗
- `setSeconds(sec, [ms])` ↗
- `setMilliseconds(ms)` ↗
- `setTime(milliseconds)` ↗ (imposta la data in millisecondi dal 01.01.1970 UTC)

Ognuno dei metodi sopra, ad eccezione di `setTime()` possiedono la variante UTC, ad esempio: `setUTCHours()`.

Come possiamo vedere, alcuni metodi possono impostare più componenti in una volta sola, ad esempio `setHours`. I componenti che non vengono menzionati non verranno modificati.

Ad esempio:

```
let today = new Date();

today.setHours(0);
alert(today); // ancora oggi, ma l'ora è cambiata a 0

today.setHours(0, 0, 0, 0);
alert(today); // ancora oggi, ma ora è 00:00:00 preciso.
```

## Autocorrezione

L' *autocorrezione* è un caratteristica molto utile degli oggetti `Date`. Potremmo inserire valori fuori dagli intervalli, e questi verranno automaticamente aggiustati.

Ad esempio:

```
let date = new Date(2013, 0, 32); // 32 Gen 2013 ?!?
alert(date); // ...è il primo Feb 2013!
```

I componenti fuori dall'intervallo vengono distribuiti automaticamente.

Ipotizziamo di voler incrementare la data “28 Feb 2016” di 2 giorni. Potrebbe essere “2 Mar” o “1 Mar” nel caso di anno bisestile. Non abbiamo bisogno di pensarci. Semplicemente aggiungiamo 2 giorni. L'oggetto `Date` farà il resto:

```
let date = new Date(2016, 1, 28);
date.setDate(date.getDate() + 2);

alert(date); // 1 Mar 2016
```

Questa caratteristica viene utilizzata spesso per ottenere la data dopo un certo periodo di tempo. Ad esempio, proviamo ad ottenere la data di “70 secondi da adesso”:

```
let date = new Date();
date.setSeconds(date.getSeconds() + 70);

alert(date); // mostra la data corretta
```

Possiamo anche impostare zero o un valore negativo. Ad esempio:

```
let date = new Date(2016, 0, 2); // 2 Gen 2016

date.setDate(1); // imposta il primo giorno del mese
alert(date);

date.setDate(0); // il primo giorno del mese è 1, quindi viene impostato l'ultimo giorno del mes
alert(date); // 31 Dec 2015
```

## Da date a number, differenza di date

Quando un oggetto `Date` viene convertito a numero, diventa un timestamp come `date.getTime()`:

```
let date = new Date();
alert(+date); // il numero di millisecondi, uguale a date.getTime()
```

Un importante effetto collaterale: le date possono essere sottratte; il risultato è la loro differenza in millisecondi.

Questa caratteristica può essere utilizzata per effettuare misurazioni:

```
let start = new Date(); // inizia a misurare il tempo

// esegui le tue operazioni
for (let i = 0; i < 100000; i++) {
 let doSomething = i * i * i;
}

let end = new Date(); // misura il tempo

alert(`The loop took ${end - start} ms`);
```

## Date.now()

Se vogliamo solo misurare una differenza, non abbiamo bisogno di un oggetto `Date`.

Esiste uno speciale metodo `Date.now()` che ritorna il timestamp corrente.

E' equivalente a `new Date().getTime()`, ma evita di creare un oggetto `Date`. Quindi risulta più veloce e non produce spazzatura in memoria.

Viene spesso utilizzato per comodità o quando le prestazioni diventano fondamentali, come nei giochi o altre particolari applicazioni.

Quindi è meglio fare:

```

let start = Date.now(); // il conto dei millisecondi dal 1 Gen 1970

// esegui le tue operazioni
for (let i = 0; i < 100000; i++) {
 let doSomething = i * i * i;
}

let end = Date.now(); // done

alert(`The loop took ${end - start} ms`); // sottrai numeri, non date

```

## Benchmarking

Se volessimo un benchmark affidabile del consumo di CPU di una funzione, dovremmo prestare attenzione.

Ad esempio, proviamo a misurare due funzioni che calcolano la differenza tra due date: quale sarebbe più veloce?

Queste misurazioni di performance vengono spesso chiamate “benchmarks”.

```

// abbiamo date1 e date2, quale funzione ritorna più velocemente la loro differenza in ms?
function diffSubtract(date1, date2) {
 return date2 - date1;
}

// o
function diffGetTime(date1, date2) {
 return date2.getTime() - date1.getTime();
}

```

Queste due fanno esattamente la stessa cosa, ma una di loro usa esplicitamente `date.getTime()` per ottenere la data in millisecondi, mentre l'altra si appoggia alla conversione data-numero. Il risultato non cambia.

Quindi, quale delle due è più veloce?

Una prima idea potrebbe essere quella di eseguirle varie volte e misurare la differenza. Nel nostro caso, le funzioni sono molto semplici, quindi dovremmo eseguirle 100000 volte.

Proviamo a misurare:

```

function diffSubtract(date1, date2) {
 return date2 - date1;
}

function diffGetTime(date1, date2) {
 return date2.getTime() - date1.getTime();
}

function bench(f) {
 let date1 = new Date(0);
 let date2 = new Date();

 let start = Date.now();

```

```

 for (let i = 0; i < 100000; i++) f(date1, date2);
 return Date.now() - start;
}

alert('Time of diffSubtract: ' + bench(diffSubtract) + 'ms');
alert('Time of diffGetTime: ' + bench(diffGetTime) + 'ms');

```

Wow! L'utilizzo di `getTime()` è molto più veloce! Questo accade perché non c'è alcuna conversione di tipo, il che significa un'operazione più semplice da ottimizzare.

Okay, abbiamo qualcosa. Ma non è sufficiente.

Immaginiamo che al momento dell'esecuzione di `bench(diffSubtract)` le risorse della CPU siano occupate, e che allo stesso tempo `bench(diffGetTime)` sia già stato elaborato.

Uno scenario realistico per i processori moderni.

Quindi, il primo benchmark potrebbe richiedere meno risorse CPU del secondo. E ciò potrebbe portare a conclusioni errate.

**Per eseguire benchmark più affidabili, l'intero pacchetto di benchmark dovrebbe essere eseguito più volte.**

Qui un esempio:

```

function diffSubtract(date1, date2) {
 return date2 - date1;
}

function diffGetTime(date1, date2) {
 return date2.getTime() - date1.getTime();
}

function bench(f) {
 let date1 = new Date(0);
 let date2 = new Date();

 let start = Date.now();
 for (let i = 0; i < 100000; i++) f(date1, date2);
 return Date.now() - start;
}

let time1 = 0;
let time2 = 0;

// esegue alternativamente bench(diffSubtract) e bench(diffGetTime) per 10 volte
for (let i = 0; i < 10; i++) {
 time1 += bench(diffSubtract);
 time2 += bench(diffGetTime);
}

alert('Total time for diffSubtract: ' + time1);
alert('Total time for diffGetTime: ' + time2);

```

I motori JavaScript moderni iniziano ad applicare ottimizzazioni solamente a "pezzi" di codice eseguiti molte volte (non è necessario ottimizzare un codice eseguito di rado). Quindi,

nell'esempio sopra, la prima esecuzione non è ben ottimizzata. Vorremmo quindi poter forzare l'ottimizzazione:

```
// aggiunto per "riscaldare" prima del loop principale
bench(diffSubtract);
bench(diffGetTime);

// ora benchmark
for (let i = 0; i < 10; i++) {
 time1 += bench(diffSubtract);
 time2 += bench(diffGetTime);
}
```

### Prestate attenzione ai microbenchmarking

I moderni motori JavaScript applicano molte ottimizzazioni. Potrebbero quindi “truccare” i risultati di un “test artificiale” a differenza del “normale utilizzo”, specialmente se stiamo eseguendo benchmark molto piccoli. Quindi se l'intenzione è quella di studiare le prestazioni, vale la pena studiare come funziona il motore JavaScript. Probabilmente non avrete più bisogno dei microbenchmark.

Un buona libreria di articoli può essere trovata qui: <http://mrale.ph>.

## Date.parse da una stringa

Il metodo [Date.parse\(str\)](#) può leggere una data da una stringa.

Il formato della stringa dovrebbe essere: `YYYY-MM-DDTHH:mm:ss.sssZ`, dove:

- `YYYY-MM-DD` – è la data: anno-mese-giorno.
- Il carattere `"T"` viene utilizzato come delimitatore.
- `HH:mm:ss.sss` – è l'orario: ore, minuti, secondi e millisecondi.
- La parte opzionale `'Z'` indica il fuso orario nel formato `+-hh:mm`. La singola lettera `Z` rappresenta UTC+0.

Sono disponibili anche varianti più brevi, come `YYYY-MM-DD` o `YYYY-MM` o anche `YYYY`.

La chiamata a `Date.parse(str)` analizza la stringa e ritorna il timestamp (numero di millisecondi trascorsi dal 1 Gennaio 1970). Se il formato non è valido, viene ritornato `Nan`.

Ad esempio:

```
let ms = Date.parse('2012-01-26T13:51:50.417-07:00');

alert(ms); // 1327611110417 (timestamp)
```

Possiamo utilizzare questo metodo insieme a `new Date` per creare un oggetto dal timestamp:

```
let date = new Date(Date.parse('2012-01-26T13:51:50.417-07:00'));
```

```
alert(date);
```

## Riepilogo

- Le date e gli orari in JavaScript sono rappresentate dall'oggetto [Date ↗](#). Non possiamo creare “solo una data” o “solo un orario”: l'oggetto `Date` li gestisce entrambi.
- Il conteggio dei mesi parte da zero (Gennaio viene identificato dallo zero).
- Il conteggio dei giorni della settimana in `getDay()` inizia da zero (la Domenica).
- `Date` si auto-corregge quando inseriamo valori fuori dai limiti. Questa caratteristica è fondamentale per sommare/sottrarre giorni/mesi/ora.
- Le date possono essere sottratte, fornendo la loro differenza in millisecondi. Questo è possibile perché `Date` diventa un timestamp quando lo convertiamo al tipo numerico.
- Si utilizza `Date.now()` per ottenere più rapidamente il corrente timestamp.

Da notare che a differenza di molti altri sistemi, in JavaScript il timestamp viene espresso in millisecondi, non in secondi.

Inoltre, talvolta potremmo aver bisogno di misurazioni più precise. JavaScript non permette di gestire i microsecondi (1 milionesimo di secondo), ma molti altri ambienti forniscono questa possibilità. Ad esempio, i browser possiedono [performance.now\(\) ↗](#) che fornisce il numero di millisecondi a partire dall'inizio del caricamento della pagina con precisione al microsecondo (3 cifre dopo la virgola):

```
alert(`Loading started ${performance.now()}ms ago`);
// Qualcosa come: "Loading started 34731.2600000001ms ago"
// .26 è microsecondi (260 microsecondi)
// dopo il punto decimale i numeri che seguono il terzo sono errori di precisione, solo i primi
```

Node.js possiede un modulo `microtime` e altri metodi. Tecnicamente, la maggior parte degli ambienti forniscono un modo per gestire precisioni più elevate, questo non è però previsto dall'oggetto `Date`.

## ✓ Esercizi

### Creare un oggetto date

importanza: 5

Create un oggetto `Date` per la data: Febbraio 20, 2012, 3:12am. Con ora locale.

Mostratela con `alert`.

[Alla soluzione](#)

### Mostrare il giorno della settimana

importanza: 5

Scrivete una funzione `getWeekDay(date)` per mostrare il giorno della settimana nel formato breve: 'LUN', 'MAR', 'MER', 'GIO', 'VEN', 'SAB', 'DOM'.

Ad esempio:

```
let date = new Date(2012, 0, 3); // 3 Gen 2012
alert(getWeekDay(date)); // dovrebbe mostrare "MAR"
```

Apri una sandbox con i test. [↗](#)

[Alla soluzione](#)

---

## Giorno della settimana Europeo

importanza: 5

Nel continente europeo i giorni della settimana iniziano con Lunedì (numero 1); segue Martedì (numero 2) e così via fino a Domenica (numero 7). Scrivete una funzione `getLocalDay(date)` che ritorna il giorno della settimana nel formato europeo.

```
let date = new Date(2012, 0, 3); // 3 Gen 2012
alert(getLocalDay(date)); // martedì, dovrebbe mostrare 2
```

Apri una sandbox con i test. [↗](#)

[Alla soluzione](#)

---

## Quale giorno del mese era qualche giorno fa?

importanza: 4

Create una funzione `getDateAgo(date, days)` che ritorna il giorno del mese di `days` giorni fa, a partire da `date`.

Ad esempio, se oggi è il 20, allora da `getDateAgo(new Date(), 1)` dovrebbe risultare il 19, e `getDateAgo(new Date(), 2)` dovrebbe ritornare 18.

Dovrebbe funzionare in maniera affidabile anche con `days=365` (o maggiori):

```
let date = new Date(2015, 0, 2);

alert(getDateAgo(date, 1)); // 1, (1 Gen 2015)
alert(getDateAgo(date, 2)); // 31, (31 Dec 2014)
alert(getDateAgo(date, 365)); // 2, (2 Gen 2014)
```

P.S. La funzione non deve modificare l'oggetto `date`.

Apri una sandbox con i test. [↗](#)

[Alla soluzione](#)

## Ultimo giorno del mese?

importanza: 5

Scrivi una funzione `getLastDayOfMonth(year, month)` che ritorna l'ultimo giorno del mese. Potrebbe essere il 30, il 31 o anche 28/29.

Parametri:

- `year` – anno nel formato 4 cifre, ad esempio 2012.
- `month` – mese, da 0 a 11.

Ad esempio, `getLastDayOfMonth(2012, 1) = 29` (anno bisestile, Febbraio).

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

---

## Quanti giorni sono passati oggi?

importanza: 5

Scrivete una funzione `getSecondsToday()` che ritorna il numero di secondi trascorsi oggi.

Ad esempio, se ora sono le `10:00 am`, senza contare gli spostamenti dovuti all'ora legale, allora:

```
getSecondsToday() == 36000 // (3600 * 10)
```

La funzione dovrebbe funzionare qualsiasi giorno. In altre parole, non deve esserci alcuna forzatura sul valore del giorno.

[Alla soluzione](#)

---

## Quanti secondi mancano a domani?

importanza: 5

Create una funzione `getSecondsToTomorrow()` che ritorni il numero di secondi mancanti al giorno successivo.

Ad esempio, se ora sono le `23:00`, allora:

```
getSecondsToTomorrow() == 3600
```

P.S. Anche in questo caso la funzione non dovrebbe forzare il valore del giorno.

[Alla soluzione](#)

---

## Formattare la data

importanza: 4

Scrivete una funzione `formatDate(date)` che dovrebbe formattare `date` come segue:

- Se da `date` è passato meno di un secondo, allora ritorna `"right now"`.
- Altrimenti, se da `date` è passato meno di un minuto, allora ritorna `"n sec. ago"`.
- Altrimenti, se è passata meno di un'ora, ritorna `"m min. ago"`.
- Altrimenti, l'intera data nel formato `"DD.MM.YY HH:mm"`. Ovvero: `"day.month.year hours:minutes"`, tutto nel formato due cifre, ad esempio `31.12.16 10:00`.

Un esempio:

```
alert(formatDate(new Date(new Date - 1))); // "right now"
alert(formatDate(new Date(new Date - 30 * 1000))); // "30 sec. ago"
alert(formatDate(new Date(new Date - 5 * 60 * 1000))); // "5 min. ago"

// la data di ieri, come: 31.12.16 20:00
alert(formatDate(new Date(new Date - 86400 * 1000)));
```

Apri una sandbox con i test. ↗

Alla soluzione

## Metodi JSON, `toJSON`

Ipotizziamo di avere un oggetto complesso, che vogliamo convertire a stringa prima trasmetterlo in rete, o anche solo per mostrarlo sullo schermo.

Naturalmente, una stringa di questo tipo deve includere tutte le proprietà importanti.

Potremmo implementarla in questo modo:

```
let user = {
 name: "John",
 age: 30,

 toString() {
 return `name: ${this.name}, age: ${this.age}`;
 }
};

alert(user); // {name: "John", age: 30}
```

...Ma nel processo di sviluppo, potrebbero essere aggiunte/eliminate/rinominate nuove proprietà. Aggiornare costantemente la funzione `toString` non è una buona soluzione. Potremmo provare a iterare sulle proprietà dell'oggetto, ma cosa succederebbe se l'oggetto contenesse oggetti annidati? Dovremmo implementare anche questa conversione. E se dovessimo inviare l'oggetto in rete, dovremmo anche fornire il codice per poterlo "leggere".

Fortunatamente, non c'è bisogno di scrivere del codice per gestire questa situazione.

## JSON.stringify

JSON (JavaScript Object Notation) è un formato per rappresentare valori e oggetti. Viene descritto nello standard RFC 4627. Inizialmente fu creato per lavorare con JavaScript, ma molti altri linguaggi possiedono delle librerie per la sua gestione. Quindi JSON risulta semplice da usare per lo scambio di dati quando il client utilizza JavaScript e il server codifica in Ruby/PHP/Java/Altro.

JavaScript fornisce i metodi:

- `JSON.stringify` per convertire oggetti in JSON.
- `JSON.parse` per convertire JSON in oggetto.

Ad esempio, abbiamo `JSON.stringify` per uno studente:

```
let student = {
 name: 'John',
 age: 30,
 isAdmin: false,
 courses: ['html', 'css', 'js'],
 wife: null
};

let json = JSON.stringify(student);

alert(typeof json); //abbiamo una stringa!

alert(json);
/* JSON-encoded object:
{
 "name": "John",
 "age": 30,
 "isAdmin": false,
 "courses": ["html", "css", "js"],
 "wife": null
}
*/
```

Il metodo `JSON.stringify(student)` prende l'oggetto e lo converte in stringa.

La stringa `json` risultante viene chiamata *codifica in JSON* o *serializzata, stringificata*, o addirittura oggetto *caramellizzato*. Ora è pronto per essere inviato o memorizzato.

Da notare che un oggetto codificato in JSON possiede delle fondamentali differenze da un oggetto letterale:

- Le stringhe utilizzano doppie virgolette. In JSON non vengono utilizzare backtick o singole virgolette. Quindi `'John'` diventa `"John"`.
- Anche i nomi delle proprietà dell'oggetto vengono racchiusi tra doppie virgolette. Quindi `age:30` diventa `"age":30`.

`JSON.stringify` può anche essere applicato agli oggetti primitivi.

I tipi che supportano nativamente JSON sono:

- Objects `{ ... }`
- Arrays `[ ... ]`
- Primitives:
  - strings,
  - numbers,
  - valori boolean `true/false`,
  - `null`.

Ad esempio:

```
// un numero in JSON è solo un numero
alert(JSON.stringify(1)) // 1

// una stringa in JSON è ancora una stringa, ma con doppie virgolette

alert(JSON.stringify('test')) // "test"

alert(JSON.stringify(true)); // true

alert(JSON.stringify([1, 2, 3])); // [1,2,3]
```

JSON è un linguaggio specifico per i dati, quindi alcune proprietà specifiche di JavaScript vengono ignorate da `JSON.stringify`.

Tra cui:

- Funzioni (metodi).
- Proprietà di tipo symbol.
- Proprietà che contengono `undefined`.

```
let user = {
 sayHi() { // ignorato
 alert("Hello");
 },
 [Symbol("id")]: 123, // ignorato
 something: undefined // ignorato
};

alert(JSON.stringify(user)); // {} (oggetto vuoto)
```

Solitamente questo è ciò che vogliamo. Se invece abbiamo intenzioni diverse, molto presto vedremo come controllare il processo.

Un'ottima cosa è che anche gli oggetti annidati vengono automaticamente convertiti.

Ad esempio:

```
let meetup = {
 title: "Conference",
```

```

room: {
 number: 23,
 participants: ["john", "ann"]
};

alert(JSON.stringify(meetup));
/* Tutta la struttura viene serializzata

{
 "title": "Conference",
 "room": {"number": 23, "participants": ["john", "ann"]},
}
*/

```

Una limitazione importante: non ci devono essere riferimenti ciclici.

Ad esempio:

```

let room = {
 number: 23
};

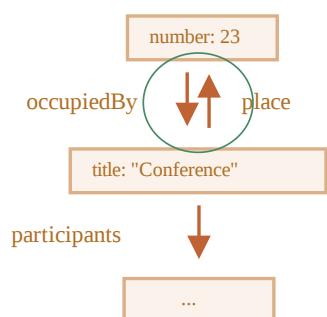
let meetup = {
 title: "Conference",
 participants: ["john", "ann"]
};

meetup.place = room; // meetup riferenzia room
room.occupiedBy = meetup; // room riferenzia meetup

JSON.stringify(meetup); // Errore: conversione di struttura circolare in JSON

```

In questo caso, la conversione fallisce a causa del riferimento circolare: `room.occupiedBy` fa riferimento a `meetup`, e `meetup.place` fa riferimento a `room`:



## Esclusione e rimpiazzo: replacer

La sintassi completa di `JSON.stringify` è:

```
let json = JSON.stringify(value[, replacer, space])
```

**value**

Valore da codificare.

### replacer

Array di proprietà da codificare o una funzione di mapping `function(key, value)`.

### space

Quantità di spazio da utilizzare per la formattazione

Nella maggior parte dei casi, `JSON.stringify` viene utilizzato specificando solamente il primo argomento. Ma se avessimo bisogno di gestire il processo di rimpiazzo, ad esempio filtrando i riferimenti ciclici, possiamo utilizzare il secondo argomento di `JSON.stringify`.

Se forniamo un array di proprietà, solamente quelle proprietà verranno codificate.

Ad esempio:

```
let room = {
 number: 23
};

let meetup = {
 title: "Conference",
 participants: [{name: "John"}, {name: "Alice"}],
 place: room // meetup referenzia room
};

room.occupiedBy = meetup; // room referenzia meetup

alert(JSON.stringify(meetup, ['title', 'participants']));
// {"title":"Conference","participants": [{"name": "John"}, {"name": "Alice"}]}
```

Qui, probabilmente, siamo stati troppo rigidi. La lista di proprietà viene applicata all'intera struttura dell'oggetto. Quindi `participants` risulta essere vuoto, perché `name` non è in lista.

Andiamo ad includere ogni proprietà ad eccezione di `room.occupiedBy`, che potrebbe causare un riferimento ciclico:

```
let room = {
 number: 23
};

let meetup = {
 title: "Conference",
 participants: [{name: "John"}, {name: "Alice"}],
 place: room // meetup referenzia room
};

room.occupiedBy = meetup; // room referenzia meetup

alert(JSON.stringify(meetup, ['title', 'participants', 'place', 'name', 'number']));
/*
{
 "title": "Conference",
 "participants": [{"name": "John"}, {"name": "Alice"}],
 "place": {"number": 23}
}
```

```
}\n*/
```

Ora tutto, ad eccezione di `occupiedBy`, viene serializzato. Ma la lista di proprietà è piuttosto lunga.

Fortunatamente, possiamo utilizzare come `replacer` una funzione piuttosto di un array.

La funzione verrà invocata per ogni coppia `(key, value)` e dovrebbe ritornare il valore sostitutivo, che verrà utilizzato al posto di quello originale.

Nel nostro caso, possiamo ritornare `value` (il valore stesso della proprietà) in tutti i casi, ad eccezione di `occupiedBy`. Per poter ignorare `occupiedBy`, il codice sotto ritorna `undefined`:

```
let room = {\n number: 23\n};\n\nlet meetup = {\n title: "Conference",\n participants: [{name: "John"}, {name: "Alice"}],\n place: room // meetup referenzia room\n};\n\nroom.occupiedBy = meetup; // room referenzia meetup\n\nalert(JSON.stringify(meetup, function replacer(key, value) {\n alert(`${key}: ${value}`);
 return (key == 'occupiedBy') ? undefined : value;
}));\n\n/* coppie key:value passate al replacer:\n : [object Object]
title: Conference
participants: [object Object],[object Object]
0: [object Object]
name: John
1: [object Object]
name: Alice
place: [object Object]
number: 23
occupiedBy: [object Object]
*/
```

Da notare che la funzione `replacer` ottiene tutte le coppie key/value, incluse quelle degli oggetti annidati e viene applicata ricorsivamente. Il valore di `this` all'interno di `replacer` è l'oggetto che contiene la proprietà corrente.

La prima chiamata è speciale. Viene effettuata utilizzando uno speciale “oggetto contenitore”: `{"": meetup}`. In altre parole, la prima coppia `(key, value)` possiede una chiave vuota, e il valore è l'oggetto stesso. Questo è il motivo per cui la prima riga dell'esempio sopra risulta essere `" : [object Object]"`.

L'idea è quella di fornire più potenza possibile a `replacer`: deve avere la possibilità di rimpiazzare/saltare l'oggetto stesso, se necessario.

## Formattazione: spacer

Il terzo argomento di `JSON.stringify(value, replacer, spaces)` è il numero di spazi da utilizzare per una corretta formattazione.

Tutti gli oggetti serializzati fino ad ora non possedevano una indentazione o spazi extra. Ci può andare bene se dobbiamo semplicemente inviare l'oggetto. L'argomento `spacer` viene utilizzato solo con lo scopo di abbellirlo.

In questo caso `spacer = 2` dice a JavaScript di mostrare gli oggetti annidati in diverse righe, con un indentazione di 2 spazi all'interno dell'oggetto:

```
let user = {
 name: "John",
 age: 25,
 roles: {
 isAdmin: false,
 isEditor: true
 }
};

alert(JSON.stringify(user, null, 2));
/* indentazione di due spazi:
{
 "name": "John",
 "age": 25,
 "roles": {
 "isAdmin": false,
 "isEditor": true
 }
}
*/
/* Per JSON.stringify(user, null, 4) il risultato sarebbe più indentato:
{
 "name": "John",
 "age": 25,
 "roles": {
 "isAdmin": false,
 "isEditor": true
 }
}
*/
```

Il parametro `spaces` viene utilizzato unicamente per procedure di logging o per abbellire l'output.

## Modificare “toJSON”

Come per la conversione `toString`, un oggetto può fornire un metodo `toJSON` per la conversione a JSON. Se questa è disponibile `JSON.stringify` la chiamerà automaticamente.

Ad esempio:

```
let room = {
```

```

 number: 23
};

let meetup = {
 title: "Conference",
 date: new Date(Date.UTC(2017, 0, 1)),
 room
};

alert(JSON.stringify(meetup));
/*
{
 "title": "Conference",
 "date": "2017-01-01T00:00:00.000Z", // (1)
 "room": {"number": 23} // (2)
}
*/

```

Qui possiamo vedere che `date` (1) diventa una stringa. Questo accade perché tutti gli oggetti di tipo `Date` possiedono un metodo `toJSON`.

Ora proviamo ad aggiungere un metodo `toJSON` personalizzato per il nostro oggetto `room` (2) :

```

let room = {
 number: 23,
 toJSON() {
 return this.number;
 }
};

let meetup = {
 title: "Conference",
 room
};

alert(JSON.stringify(room)); // 23

alert(JSON.stringify(meetup));
/*
{
 "title": "Conference",
 "room": 23
}
*/

```

Come possiamo vedere, `toJSON` viene utilizzato sia per le chiamate dirette a `JSON.stringify(room)`, sia per gli oggetti annidati.

## JSON.parse

Per decodificare una stringa in JSON, abbiamo bisogno del metodo `JSON.parse` ↗ .

La sintassi:

```
let value = JSON.parse(str, [reviver]);
```

## str

stringa JSON da decodificare.

## reviver

funzione(key, value) che verrà chiamata per ogni coppia (key, value) e che può rimpiazzare i valori.

Ad esempio:

```
// array serializzato
let numbers = "[0, 1, 2, 3]";

numbers = JSON.parse(numbers);

alert(numbers[1]); // 1
```

Nel caso di oggetti annidati:

```
let userData = '{ "name": "John", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';

let user = JSON.parse(userData);

alert(user.friends[1]); // 1
```

Il JSON può essere complesso quanto vogliamo, gli oggetti e array possono includere altri oggetti e array. Ma devono seguire il corretto formato.

Alcuni errori tipici di scrittura in JSON (in qualche caso lo utilizzeremo per scopi di debug):

```
let json = `{
 name: "John", // errore: proprietà name without quotes
 "surname": "Smith", // errore: singole virgolette in value (devono essere doppie)
 'isAdmin': false // errore: singole virgolette in key (devono essere doppie)
 "birthday": new Date(2000, 2, 3), // errore: "new" non è permesso
 "friends": [0,1,2,3] // qui tutto bene
};`;
```

Inoltre, JSON non supporta i commenti. Quindi l'aggiunta di un commento invaliderebbe il documento.

Esiste un altro formato chiamato [JSON5 ↗](#), che accetta chiavi senza virgolette, commenti etc. Ma consiste in una libreria a se stante, non fa parte della specifica del linguaggio.

Il JSON classico è così restrittivo non per pigrizia degli sviluppatori, ma per consentire una facile, affidabile e rapida implementazione degli algoritmi di analisi.

## Ritrasformare in oggetto

Immaginate di ricevere dal server un oggetto `meetup` serializzato.

Come:

```
// title: (meetup title), date: (meetup date)
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';
```

...Ora abbiamo bisogno di *deserializzarlo*, per ricreare l'oggetto.

Lo facciamo chiamando `JSON.parse`:

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str);

alert(meetup.date.getDate()); // Error!
```

Whoops! Errore!

Il valore di `meetup.date` è una stringa, non un oggetto di tipo `Date`. Come fa `JSON.parse` a sapere che dovrebbe trasformare quella stringa in un oggetto di tipo `Date`?

Passiamo a `JSON.parse` la funzione di riattivazione che ritorna tutti i valori per "come sono", quindi `date` diventerà `Date`:

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str, function(key, value) {
 if (key == 'date') return new Date(value);
 return value;
});

alert(meetup.date.getDate()); // now works!
```

Funziona anche per gli oggetti annidati:

```
let schedule = `{
 "meetups": [
 {"title":"Conference", "date":"2017-11-30T12:00:00.000Z"},
 {"title":"Birthday", "date":"2017-04-18T12:00:00.000Z"}
]
}`;

schedule = JSON.parse(schedule, function(key, value) {
 if (key == 'date') return new Date(value);
 return value;
});

alert(schedule.meetups[1].date.getDate()); // works!
```

## Riepilogo

- JSON è un formattatore di dati con i suoi standard; possiede molte librerie che gli consentono di lavorare con altrettanti linguaggi di programmazione.
- JSON supporta oggetti, array, stringhe, numeri, booleani, e `null`.
- JavaScript fornisce dei metodi: `JSON.stringify` per serializzare in JSON e `JSON.parse` per la lettura da JSON.
- Entrambi i metodi supportano funzioni di rimpiazzo per scritture/lettura “intelligenti”.
- Se un oggetto implementa `toJSON`, questa verrà automaticamente invocata da `JSON.stringify`.

## ✓ Esercizi

---

### Trasformare l'oggetto in JSON e vice versa

importanza: 5

Convertite `user` in JSON e successivamente riconvertitelo salvandolo in un'altra variabile.

```
let user = {
 name: "John Smith",
 age: 35
};
```

[Alla soluzione](#)

---

### Escludere contro-referenze

importanza: 5

In un semplice caso di riferimenti ciclici, possiamo escludere una proprietà dalla serializzazione tramite il suo nome.

Ma in certi casi possono esserci più contro-referenze. E i nomi potrebbero essere utilizzati sia per riferimenti ciclici che per normali proprietà.

Scrivete una funzione `replacer` che serializzi tutto, evitando la proprietà che fa riferimento a `meetup`:

```
let room = {
 number: 23
};

let meetup = {
 title: "Conference",
 occupiedBy: [{name: "John"}, {name: "Alice"}],
 place: room
};

// referenze circolari
room.occupiedBy = meetup;
meetup.self = meetup;

alert(JSON.stringify(meetup, function replacer(key, value) {
```

```

 /* il tuo codice */
});

/* il risultato dovrebbe essere:
{
 "title": "Conference",
 "occupiedBy": [{"name": "John"}, {"name": "Alice"}],
 "place": {"number": 23}
}
*/

```

[Alla soluzione](#)

## Gestione avanzata delle funzioni

### Ricorsione e pila

Torniamo alle funzioni per studiarle più in profondità.

Il nostro primo argomento riguarda la *ricorsione*.

Se non siete nuovi alla programmazione, potete tranquillamente saltare questo capitolo.

La ricorsione è uno modello di programmazione che diventa utile in situazioni in cui la risoluzione di un problema si presta ad essere suddivisa in altri piccoli sotto-problemi dello stesso tipo, ma più semplici. O anche nei casi in cui un problema può essere semplificato ad un semplice problema più una variante simile al problema stesso. O come vedremo presto, per lavorare con alcune strutture dati.

Quando una funzione risolve un problema, durante il processo di risoluzione può chiamare anche altre funzioni. Un caso particolare di questa situazione si ha quando la funzione chiama se stessa. Questa è la definizione di *ricorsione*.

### Due modi di pensare

Per iniziare con qualcosa di semplice – scriviamo una funzione `pow(x, n)` che eleva `x` ad una potenza naturale `n`. In altre parole, moltiplica `x` per se stessa `n` volte.

```

pow(2, 2) = 4
pow(2, 3) = 8
pow(2, 4) = 16

```

Ci sono due modi per implementarla.

1. Pensiero iterativo: il ciclo `for`:

```

function pow(x, n) {
 let result = 1;

 // multiply result by x n times in the loop
 for (let i = 0; i < n; i++) {
 result *= x;
 }
}

```

```

 }

 return result;
}

alert(pow(2, 3)); // 8

```

2. Pensiero ricorsivo: semplificare il problema e richiamare la funzione:

```

function pow(x, n) {
 if (n == 1) {
 return x;
 } else {
 return x * pow(x, n - 1);
 }
}

alert(pow(2, 3)); // 8

```

Da notare come la versione ricorsiva sia completamente differente.

Quando `pow(x, n)` viene chiamata, l'esecuzione si spezza in due rami:

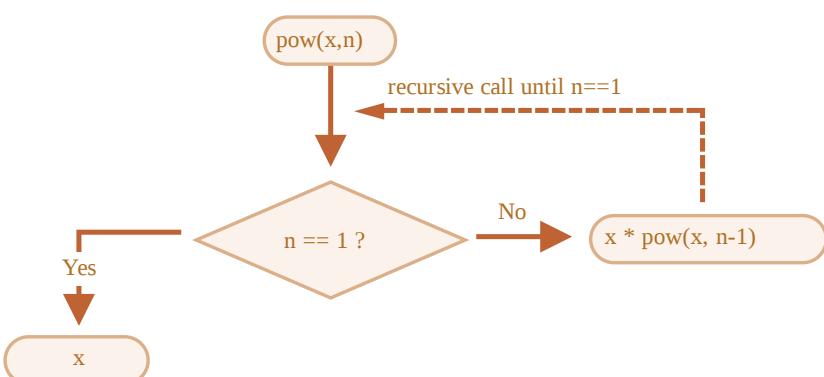
```

if n==1 = x
/
pow(x, n) =
 \
else = x * pow(x, n - 1)

```

1. Se `n == 1`, allora è banale. Viene chiamato il *caso base* della ricorsione, poiché produce immediatamente il risultato ovvio: `pow(x, 1)` uguale a `x`.
2. Altrimenti, possiamo rappresentare `pow(x, n)` come `x * pow(x, n - 1)`. In matematica, si potrebbe scrivere  $x^n = x * x^{n-1}$ . Questo viene chiamato il *passo ricorsivo*: trasformiamo il problema in un sotto-problema più semplice (moltiplicazione per `x`) e chiamiamo la stessa funzione con il sotto-problema più semplice (`pow` con una minore `n`). Il prossimo passo semplificherà ulteriormente finché `n` sarà `1`.

Possiamo anche dire che `pow` chiama ricorsivamente se stessa finché non vale `n == 1`.



Ad esempio, per calcolare `pow(2, 4)` la variante ricorsiva esegue:

1. `pow(2, 4) = 2 * pow(2, 3)`
2. `pow(2, 3) = 2 * pow(2, 2)`
3. `pow(2, 2) = 2 * pow(2, 1)`
4. `pow(2, 1) = 2`

Quindi, la ricorsione riduce una chiamata a funzione ad una più semplice, e successivamente – ad una ancora più semplice, e così via, finché il risultato diventa ovvio.

### La ricorsione è spesso più breve

Spesso una soluzione ricorsiva risulta più breve di una iterativa.

In questo caso possiamo riscrivere lo stesso codice utilizzando l'operatore ternario `?` piuttosto di un `if` per rendere `pow(x, n)` più breve e leggibile:

```
function pow(x, n) {
 return (n == 1) ? x : (x * pow(x, n - 1));
}
```

Il massimo numero di chiamate annidate (inclusa la prima) viene chiamato *profondità di ricorsione*. Nel nostro caso, sarà esattamente `n`.

La massima profondità di ricorsione viene limitata dal motore JavaScript. Possiamo farne all'incirca 10000, alcuni motori ne consentono un numero maggiore, ma 100000 probabilmente è al di fuori del limite di qualsiasi motore. Ci sono delle ottimizzazioni automatiche (“ottimizzazione della chiamate in coda”), ma non sono ancora supportate da tutti e funzionano solo in casi semplici.

Questo fattore limita le possibili applicazioni della ricorsione, che rimangono comunque molte. Ci sono molte attività che possono essere semplificati tramite la ricorsione, rendendo i programmi più mantenibili.

## Il contesto e la pila d'esecuzione

Ora vediamo come funzionano le chiamate ricorsive. Per farlo analizzeremo bene le funzioni.

L'informazione riguardo una funzione in esecuzione viene memorizzata nel suo *contesto di esecuzione*.

Il [contesto di esecuzione ↗](#) è una struttura dati interna che contiene i dettagli riguardo l'esecuzione di una funzione: dove si trova il flusso, le variabili, il valore di `this` (che non useremo in questo caso) e un paio di altri dettagli.

Una chiamata a funzione possiede esattamente un contesto di esecuzione associato.

Quando una funzione chiama una funzione annidata, succede quanto segue:

- La funzione attuale viene messa in pausa.
- Il contesto di esecuzione associato viene spostato in una struttura dati chiamata *pila dei contesti di esecuzione*.
- Viene eseguita la chiamata annidata.

- Al termine, viene ripristinato il vecchio contesto di esecuzione prelevandolo dalla pila, e la funzione esterna riprende da dove si era interrotta.

Vediamo cosa accade durante la chiamata `pow(2, 3)`.

### **pow(2, 3)**

Inizialmente con la chiamata `pow(2, 3)` il contesto d'esecuzione memorizza le variabili: `x = 2, n = 3`, mentre il flusso si trova alla riga `1` della funzione.

Che possiamo abbozzare:

- **Context: { x: 2, n: 3, at line 1 }** call: `pow(2, 3)`

Quello è ciò che accade quando la funzione inizia ad eseguire. La condizione `n == 1` è false, quindi il flusso continua nel secondo ramo della condizione `if`:

```
function pow(x, n) {
 if (n == 1) {
 return x;
 } else {
 return x * pow(x, n - 1);
 }
}

alert(pow(2, 3));
```

Le variabili sono le stesse, ma cambia la riga, quindi il contesto ora vale:

- **Context: { x: 2, n: 3, at line 5 }** call: `pow(2, 3)`

Per calcolare `x * pow(x, n - 1)`, dobbiamo eseguire una sotto-chiamata di `pow` con nuovi argomenti `pow(2, 2)`.

### **pow(2, 2)**

Per eseguire chiamate annidate, JavaScript memorizza il contesto di esecuzione nella *pila dei contesti d'esecuzione*.

Eseguiamo la chiamata della stessa funzione `pow`, ma non ha importanza. Il processo è lo stesso per tutte le funzioni:

1. Il contesto d'esecuzione viene "memorizzato" in cima alla pila.
2. Un nuovo contesto viene generato per la sotto-chiamata.
3. Quando la sotto-chiamata è conclusa – il precedente contesto viene ripristinato e rimosso dalla pila, e l'esecuzione procede.

Questo è il contesto d'esecuzione quando entriamo nella sotto-chiamata `pow(2, 2)`:

- **Context: { x: 2, n: 2, at line 1 }** call: `pow(2, 2)`
- **Context: { x: 2, n: 3, at line 5 }** call: `pow(2, 3)`

Il nuovo contesto d'esecuzione è in cima (in grassetto), e quelli precedenti sono sotto.

Quando abbiamo terminato la sotto-chiamata – è facile ripristinare il precedente contesto, poiché questo tiene traccia del punto d'arresto e delle variabili al momento dell'interruzione.

## pow(2, 1)

Il processo si ripete: una nuova sotto-chiamata viene eseguita alla riga 5, con gli argomenti x=2, n=1.

Un nuovo contesto d'esecuzione viene creato, quello precedente viene posto in cima alla pila:

- **Context: { x: 2, n: 1, at line 1 }** call: pow(2, 1)
- **Context: { x: 2, n: 2, at line 5 }** call: pow(2, 2)
- **Context: { x: 2, n: 3, at line 5 }** call: pow(2, 3)

Ora ci sono 2 vecchi contesti d'esecuzione e 1 in che sta eseguendo `pow(2, 1)`.

## L'uscita

Durante l'esecuzione di `pow(2, 1)`, a differenza delle precedenti esecuzioni, la condizione `n == 1` è vera, quindi viene preso il primo ramo `if`:

```
function pow(x, n) {
 if (n == 1) {
 return x;
 } else {
 return x * pow(x, n - 1);
 }
}
```

Non ci sono ulteriori chiamate annidate, quindi la funzione si conclude, ritornando 2.

Quando la funzione ha terminato, il suo contesto d'esecuzione non è più necessario, quindi viene rimosso dalla memoria. Viene ripristinato quello precedente, prelevandolo dalla cima della pila:

- **Context: { x: 2, n: 2, at line 5 }** call: pow(2, 2)
- **Context: { x: 2, n: 3, at line 5 }** call: pow(2, 3)

L'esecuzione di `pow(2, 2)` viene ripristinata. Ora però possiede il risultato ricevuto dalla chiamata `pow(2, 1)`, quindi può concludere il calcolo `x * pow(x, n - 1)`, ritornando 4.

Successivamente il precedente contesto viene ripristinato:

- **Context: { x: 2, n: 3, at line 5 }** call: pow(2, 3)

Quando si conclude, abbiamo il risultato di `pow(2, 3) = 8`.

La profondità d'esecuzione in questo caso è: 3.

Dalle figure viste sopra, possiamo notare che la profondità di ricorsione è uguale al massimo numero di contesti nella pila.

Da notare i requisiti di memoria. I contesti sfruttano la memoria. Nel nostro caso, la crescita della potenza `n` richiede un numero `n` di contesti.

Un algoritmo basato sui cicli risparmia più memoria:

```
function pow(x, n) {
 let result = 1;

 for (let i = 0; i < n; i++) {
 result *= x;
 }

 return result;
}
```

La forma iterativa di `pow` utilizza un solo contesto d'esecuzione, modificando `i` e `result` durante il calcolo. I suoi requisiti di memoria sono inferiori, fissati e non dipendono da `n`.

**Qualsiasi ricorsione può essere riscritta come un ciclo. La variante che utilizza un ciclo spesso può essere più efficace.**

...Qualche volta la traduzione potrebbe non essere banale, specialmente quando la funzione utilizza diverse sotto-chiamate ricorsive in base al verificarsi di certe condizioni, fonde i risultati delle diverse sotto-chiamate oppure quando le diramazioni diventano più complesse. In questi casi l'ottimizzazione potrebbe non essere necessaria o non valerne lo sforzo.

La ricorsione fornisce un codice più breve, più facile da capire e dimostrare. L'ottimizzazione non è sempre richiesta, spesso è meglio avere un buon codice, per questo viene molto utilizzata la ricorsione.

## Ricorsione trasversale

Un'altra grande applicazione della ricorsione è la ricorsione trasversale.

Immaginiamo di avere un'azienda. La struttura dello staff può essere rappresentata tramite un oggetto:

```
let company = {
 sales: [
 {
 name: 'John',
 salary: 1000
 },
 {
 name: 'Alice',
 salary: 1600
 }
],

 development: {
 sites: [
 {
 name: 'Peter',
 salary: 2000
 },
 {
 name: 'Alex',
 salary: 1800
 }
],
 internals: [
 {
 name: 'Jack',
 salary: 1300
 }
]
}
```

```
 }]
}
};
```

In altre parole, un'azienda ha dei dipartimenti.

- Un dipartimento può avere un array di staff. Ad esempio il dipartimento `sales` ("vendite") ha due impiegati: John e Alice.
- Oppure un dipartimento può essere suddiviso in due sotto-dipartimenti, come `development` che ha due rami: `sites` e `internals`. Ognuno di questi ha il proprio staff.
- E' anche possibile che un sotto-dipartimento cresca, dividendosi in sotto-sotto-dipartimenti (o `team`).

Ad esempio, il dipartimento `sites` in futuro potrebbe dividersi in due team dedicati a `siteA` e `siteB`. E questi, potenzialmente, potrebbero dividersi ulteriormente. Anche se nel nostro esempio non è così, va comunque tenuta in mente come possibilità.

Ora ipotizziamo di volere una funzione per ottenere la somma di tutti i salari. Come possiamo farlo?

Un approccio iterativo potrebbe non essere così semplice, poiché la struttura stessa non è semplice. La prima idea potrebbe essere quella di utilizzare un ciclo `for` su `company` con un sotto-ciclo annidato sul primo livello annidato dei dipartimenti. Ma ora abbiamo bisogno di ulteriori sotto-cicli annidati per poter iterare su un livello ulteriormente inferiore di staff, come ad esempio `sites`. ... E poi un ulteriore sotto-ciclo per il successivo livello di annidamento che potrebbe potenzialmente apparire in futuro. Potrebbero però esserci ulteriori livelli di annidamento, quindi inserire una serie di cicli annidati darebbe come risultato un pessimo codice.

Proviamo con la ricorsione.

Come possiamo vedere, quando la nostra funzione richiede la somma dei salari di un dipartimento, ci sono due casi possibili:

1. Siamo in caso "semplice" in cui il dipartimento contiene solamente *array di persone* – allora possiamo semplicemente sommare i salari con un ciclo.
2. Siamo nel caso *un oggetto con N sotto-dipartimenti* – allora possiamo eseguire `N` chiamate ricorsive per ottenere la somma dei vari sotto-dipartimenti e combinarle per ottenere il risultato finale.

Il caso base è (1), è banale.

Il passo ricorsivo è (2). Un problema complesso può essere diviso in sotto-problemi composti da dipartimenti. Questi potrebbero essere ulteriormente divisi, ma prima o poi ci troveremo nel caso base (1).

L'algoritmo probabilmente è più intuitivo leggendo il codice:

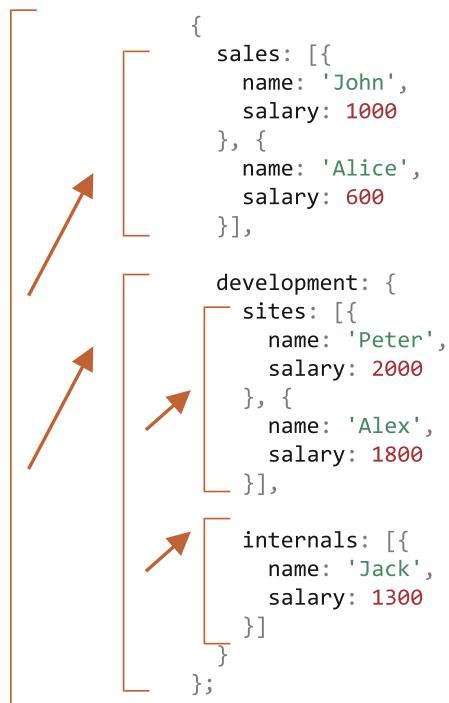
```
let company = { // the same object, compressed for brevity
 sales: [{name: 'John', salary: 1000}, {name: 'Alice', salary: 1600}],
 development: {
 sites: [{name: 'Peter', salary: 2000}, {name: 'Alex', salary: 1800}],
 internals: [{name: 'Jack', salary: 1300}]
 }
};
```

```
// The function to do the job
function sumSalaries(department) {
 if (Array.isArray(department)) { // case (1)
 return department.reduce((prev, current) => prev + current.salary, 0); // sum the array
 } else { // case (2)
 let sum = 0;
 for (let subdep of Object.values(department)) {
 sum += sumSalaries(subdep); // recursively call for subdepartments, sum the results
 }
 return sum;
 }
}

alert(sumSalaries(company)); // 7700
```

Il codice è più breve e facile da capire. Questo è il potere della ricorsione. Questa funzione continuerebbe a funzionare con qualsiasi livello di sotto-dipartimento.

Vediamo un diagramma delle chiamate:



Possiamo vedere il principio di base: per un oggetto `{ ... }` vengono effettuate le sotto-chiamate, mentre un array `[ ... ]` fornisce direttamente un risultato.

Da notare che il codice utilizza alcune caratteristiche interessanti che abbiamo già studiato:

- Il metodo `arr.reduce` spiegato nel capitolo [Metodi per gli array](#) per ottenere la somma dell'array.
- Il ciclo `for(val of Object.values(obj))` per iterare sui valori di un oggetto: `Object.values` che ritorna un array che li contiene.

## Strutture ricorsive

Una struttura ricorsiva (definita ricorsivamente) è una struttura che replica una parte di se stessa.

Abbiamo appena visto un esempio di una possibile strutturazione di un'azienda.

Un *dipartimento* di un'azienda è:

- o un array di persone.
- oppure un oggetto con *dipartimenti*.

Per gli sviluppatori web ci sono degli esempi molto più comuni: i documenti HTML e XML.

Nei documenti HTML, un *tag HTML* può contenere una lista di:

- Testo.
- Commenti HTML.
- Altri *tag HTML* (che a loro volta possono contenere testo/commenti oppure altri tag).

Questa è una definizione ricorsiva.

Per capire meglio questo concetto, studieremo una struttura dati ricorsiva chiamata “*Linked list*”, che in alcuni casi si rivela essere un'ottima sostituta agli array.

## Linked list

Immaginiamo di voler memorizzare una lista ordinata di oggetti.

La scelta naturale potrebbe ricadere su un array:

```
let arr = [obj1, obj2, obj3];
```

...Ma sorge un problema con gli array. Le operazioni di “*delete*” e “*insert*” (rispettivamente “cancellazione” e “inserimento”) sono costose. Ad esempio, `arr.unshift(obj)` deve rinumerare tutti gli elementi per creare spazio al nuovo `obj`, e se l'array fosse grande, potrebbe volerci del tempo. Lo stesso vale per `arr.shift()`.

Le uniche operazioni sulla struttura di un array che non richiedono una renumerazione di massa, sono quelle eseguite in coda all'array: `arr.push/pop`. Quindi un array può risultare piuttosto lento per certe operazioni.

In alternativa, se la situazione richiede rapidità nelle operazioni di inserimento/rimozione, possiamo optare per una struttura dati chiamata [linked list ↗](#).

Gli *elementi della linked list* vengono definiti ricorsivamente come un oggetto con:

- `value`.
- `next` proprietà che contiene un riferimento al prossimo *elemento della linked list* oppure `null` se siamo alla fine.

Ad esempio:

```
let list = {
 value: 1,
 next: {
 value: 2,
 next: {
 value: 3,
 next: {
```

```

 value: 4,
 next: null
 }
}
};


```

La rappresentazione grafica della linked list:



Un codice alternativo per la creazione:

```

let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };
list.next.next.next = { value: 4 };
list.next.next.next.next = null;

```

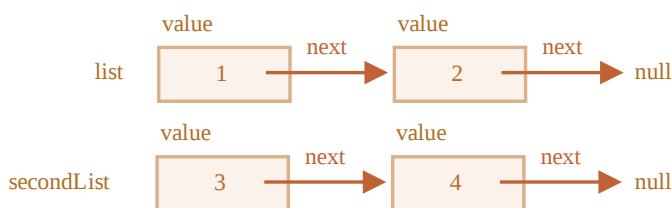
Qui possiamo vedere ancora più chiaramente che ci sono più oggetti, ognuno possiede gli attributi `value` e `next` che fa riferimento al vicino. La variabile `list` contiene il primo elemento della lista, segue il puntatore `next` tramite cui possiamo accedere a qualsiasi elemento.

La lista può essere divisa in più parti e ricomposta più avanti:

```

let secondList = list.next.next;
list.next.next = null;

```



Per ricomporre la lista:

```

list.next.next = secondList;

```

E ovviamente possiamo inserire o rimuovere elementi in qualsiasi posizione.

Ad esempio, per inserire un elemento all'inizio, è sufficiente aggiornare la testa della lista:

```

let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };

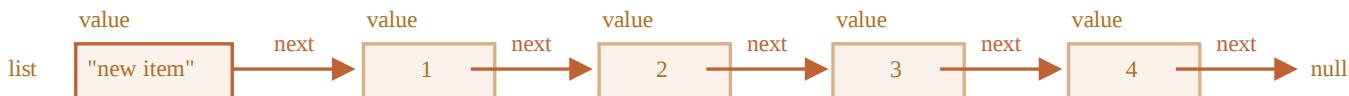
```

```

list.next.next.next = { value: 4 };

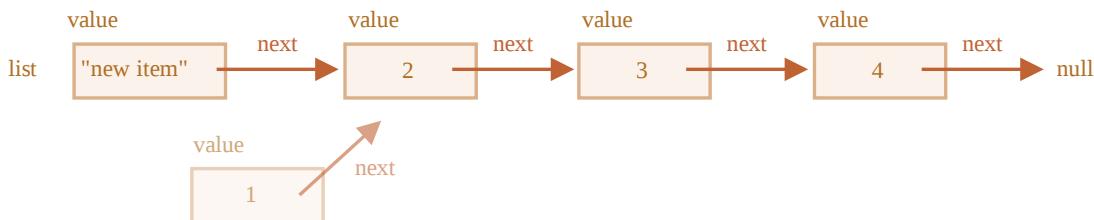
// prepend the new value to the list
list = { value: "new item", next: list };

```



Per rimuovere un elemento al centro, modifichiamo il campo `next` di quello precedente:

```
list.next = list.next.next;
```



Abbiamo modificato `list.next` da `1` a `2`. Il valore `1` è ora escluso dalla lista. Se non è stato memorizzato in nessun'altra parte del codice, questo verrà automaticamente rimosso dalla memoria.

A differenza degli array, non c'è alcuna reenumerazione di massa, possiamo riorganizzare gli elementi molto rapidamente.

Naturalmente, le liste non sono sempre la scelta migliore. Altrimenti verrebbero utilizzate solamente liste.

Il principale difetto è l'impossibilità di accedere direttamente ad un elemento tramite il numero. In un array è semplice: `arr[n]` è un riferimento diretto. Nelle liste è necessario partire dal primo elemento e scorrere `next`  $N$  volte per arrivare all' $n$ -esimo elemento.

...Non sempre abbiamo bisogno di queste operazioni. Ad esempio, potremmo utilizzare una queue oppure una [deque ↗](#) – una struttura dati ordinata che consente operazioni di inserimento/rimozione molto rapide sia in testa che in coda.

Talvolta vale la pena aggiungere un ulteriore variabile denominata `tail` per tenere traccia dell'ultimo elemento della lista (e aggiornarla ad ogni inserimento/rimozione in coda). Per grandi insiemi di elementi la differenza di velocità in confronto agli array è grande.

## Riepilogo

Terminologia:

- *Ricorsione* è un termine della programmazione che rappresenta una funzione che esegue “chiamate a se stessa”. Queste funzioni possono essere utilizzate per una risoluzione più elegante di determinati problemi.

Quando una funzione chiama se stessa, si indica questa azione come *passo ricorsivo*. La base della ricorsione sono degli argomenti che rendono la risoluzione del problema banale e immediata.

- Una struttura dati [definita ricorsivamente ↗](#) è una struttura che si definisce utilizzando se stessa.

Ad esempio, la linked list può essere definita come una struttura dati che consiste di un valore e un puntatore al successivo nodo (oppure null).

```
list = { value, next -> list }
```

Gli elementi HTML o la definizione di dipartimento sono definizioni ricorsive: ogni ramo può avere altri rami.

Si possono utilizzare funzioni ricorsive per attraversare questo tipo di oggetti, come abbiamo visto nell'esempio `sumSalary`.

Qualsiasi funzione ricorsiva può essere riscritta come iterativa. A volte è richiesta questa conversione, per ottimizzare le prestazioni. Ma molti problemi sono più semplici da risolvere tramite la ricorsione.

## ✓ Esercizi

### Sommare tutti i numeri fino a quello dato

importanza: 5

Scrivete una funzione `sumTo(n)` che calcola la somma dei numeri `1 + 2 + ... + n`.

Ad esempio:

```
sumTo(1) = 1
sumTo(2) = 2 + 1 = 3
sumTo(3) = 3 + 2 + 1 = 6
sumTo(4) = 4 + 3 + 2 + 1 = 10
...
sumTo(100) = 100 + 99 + ... + 2 + 1 = 5050
```

Scrivete 3 diverse varianti della soluzione:

1. Utilizzando un ciclo `for`.
2. Utilizzando la ricorsione, poiché `sumTo(n) = n + sumTo(n-1)` per `n > 1`.
3. Utilizzate la formula della [progressione aritmetica ↗](#).

Un esempio:

```
function sumTo(n) { /*... your code ... */ }

alert(sumTo(100)); // 5050
```

P.S. Quale soluzione risulta essere la più rapida? La più lenta? Perché?

P.P.S. Possiamo utilizzare la ricorsione per calcolare `sumTo(100000)`?

[Alla soluzione](#)

---

## Calcolare il fattoriale

importanza: 4

Il [fattoriale](#) di un numero naturale è il numero moltiplicato per "numero meno uno", poi per "numero meno due", e così via fino a 1. Il fattoriale di n si indica con n!

Possiamo definire il fattoriale come:

```
n! = n * (n - 1) * (n - 2) * ... * 1
```

Esempi:

```
1! = 1
2! = 2 * 1 = 2
3! = 3 * 2 * 1 = 6
4! = 4 * 3 * 2 * 1 = 24
5! = 5 * 4 * 3 * 2 * 1 = 120
```

Si richiede di scrivere una funzione `factorial(n)` che calcola n! utilizzando chiamate ricorsive.

```
alert(factorial(5)); // 120
```

P.S. Aiuto: n! può essere riscritto come n \* (n-1)!. Ad esempio: 3! = 3\*2! = 3\*2\*1! = 6

[Alla soluzione](#)

---

## Successione di Fibonacci

importanza: 5

La successione di [Fibonacci](#) la cui formula è  $F_n = F_{n-1} + F_{n-2}$ . In altre parole, il numero successivo è la somma dei due risultati precedenti.

I primi due numeri sono 1, poi 2(1+1), 3(1+2), 5(2+3) e così via: 1, 1, 2, 3, 5, 8, 13, 21....

La successione di Fibonacci è in relazione con il [rapporto aureo](#) e con molti altri fenomeni naturali.

Scrivete una funzione `fib(n)` che ritorna l' n-esimo numero della successione di Fibonacci.

Un esempio:

```
function fib(n) { /* your code */ }

alert(fib(3)); // 2
alert(fib(7)); // 13
alert(fib(77)); // 5527939700884757
```

P.S. La funzione dovrebbe essere rapida. La chiamata `fib(77)` non dovrebbe richiedere più di una frazione di secondo.

[Alla soluzione](#)

## Stampare una single-linked list

importanza: 5

Ipotizziamo di avere una single-linked list (descritta nel capitolo [Ricorsione e pila](#)):

```
let list = {
 value: 1,
 next: {
 value: 2,
 next: {
 value: 3,
 next: {
 value: 4,
 next: null
 }
 }
 }
};
```

Scrivete una funzione `printList(list)` che ritorna gli elementi uno ad uno.

Create due varianti della soluzione: iterativa e ricorsiva.

Qual'è la migliore: ricorsione o senza?

[Alla soluzione](#)

## Stampare una single-linked list in ordine inverso

importanza: 5

Stampate una single-linked list dell'esercizio precedente [Stampare una single-linked list](#) in ordine inverso.

Scrivete due soluzioni: iterativa e ricorsiva.

[Alla soluzione](#)

## Parametri resto e operatore di espansione

Molte funzioni integrate in JavaScript supportano un numero arbitrario di argomenti.

Ad esempio:

- `Math.max(arg1, arg2, ..., argN)` – ritorna il maggiore degli argomenti.
- `Object.assign(dest, src1, ..., srcN)` – copia le proprietà da `src1..N` in `dest`.
- ...e molto altro.

In questo capitolo impareremo come farlo. Ma soprattutto, impareremo come utilizzare al meglio questo tipo di funzioni.

## Parametri resto ...

Una funzione può essere invocata con un qualsiasi numero di argomenti, non ha importanza come sono definiti.

Come qui:

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

alert(sum(1, 2, 3, 4, 5));
```

In questo caso non ci saranno errori dovuti “all’eccesso” di argomenti. Ma ovviamente il risultato terrà conto solamente dei primi due.

I parametri restanti possono essere menzionati nella definizione di una funzione con i tre punti .... Che significano letteralmente “raccogli gli altri parametri in un array”.

Ad esempio, per raccogliere tutti gli argomenti in un array `args`:

```
function sumAll(...args) { // args is the name for the array
 let sum = 0;

 for (let arg of args) sum += arg;

 return sum;
}

alert(sumAll(1)); // 1
alert(sumAll(1, 2)); // 3
alert(sumAll(1, 2, 3)); // 6
```

Possiamo anche decidere di prendere i primi parametri e memorizzarli in variabili, e i parametri avanzati metterli in un array.

In questo caso i primi due argomenti vengono memorizzati in variabili i restanti finiscono nell’array `titles`:

```
function showName(firstName, lastName, ...titles) {
```

```

 alert(firstName + ' ' + lastName); // Julius Caesar

 // the rest go into titles array
 // i.e. titles = ["Consul", "Imperator"]
 alert(titles[0]); // Consul
 alert(titles[1]); // Imperator
 alert(titles.length); // 2
}

showName("Julius", "Caesar", "Consul", "Imperator");

```

### I parametri resto devono apparire alla fine

I parametri resto raccolgono tutti gli argomenti che avanzano, quindi non avrebbe senso fare:

```

function f(arg1, ...rest, arg2) { // arg2 after ...rest ?!
 // error
}

```

L'array `...rest` deve sempre apparire come ultimo.

## La variabile “arguments”

Esiste anche un oggetto simil-array denominato `arguments` che contiene tutti gli argomenti per indice.

Ad esempio:

```

function showName() {
 alert(arguments.length);
 alert(arguments[0]);
 alert(arguments[1]);

 // it's iterable
 // for(let arg of arguments) alert(arg);
}

// shows: 2, Julius, Caesar
showName("Julius", "Caesar");

// shows: 1, Ilya, undefined (no second argument)
showName("Ilya");

```

Agli inizi, i parametri resto non esistevano nel linguaggio, e si utilizzava `arguments` per ottenere tutti gli argomenti di una funzione.

Questa funzionalità ovviamente è ancora presente, possiamo quindi utilizzarla.

Il lato negativo è che `arguments` è un oggetto simil-array iterabile, non è un array puro. Non supporta quindi i metodi dedicati agli array, come ad esempio `arguments.map(...)`.

Inoltre, questo conterrà sempre tutti gli elementi. Non possiamo raccoglierli parzialmente, come abbiamo fatto con i parametri resto.

Quindi, in queste situazioni, si preferisce utilizzare i parametri resto.

### **i** Le funzioni freccia non possiedono "argomenti"

Se provassimo ad accedere all'oggetto `arguments` all'interno di una funzione freccia, questo preleverebbe le variabili dal contesto esterno.

Un esempio:

```
function f() {
 let showArg = () => alert(arguments[0]);
 showArg();
}

f(1); // 1
```

In sostanza, le funzioni freccia non hanno un proprio `this`. Ora sappiamo anche che non possiedono nemmeno l'oggetto `arguments`.

## Operatore di espansione

Abbiamo appena visto come ottenere un array da una lista di parametri.

In certe situazioni abbiamo bisogno di fare esattamente il contrario.

Ad esempio, esiste una funzione integrata `Math.max` ↗ che ritorna il numero maggiore di una lista:

```
alert(Math.max(3, 5, 1)); // 5
```

Ora ipotizziamo di avere un array `[3, 5, 1]`. Come invochiamo `Math.max` su un array?

Il semplice passaggio “così com’è” non funzionerebbe, perché `Math.max` si aspetta di ricevere una lista di argomenti numerici, non un singolo array:

```
let arr = [3, 5, 1];

alert(Math.max(arr)); // NaN
```

E ovviamente non possiamo nemmeno elencare manualmente tutti gli elementi in questo modo: `Math.max(arr[0], arr[1], arr[2])`, poiché il numero di elementi contenuti nell’array potrebbe non essere noto. In ogni caso non sarebbe nemmeno elegante.

L’*operatore di espansione* ci aiuta in questo! La sintassi è simile a quella dei parametri resto, utilizza `...`, ma fa esattamente l’opposto.

Quando si utilizza `...arr` in una chiamata a funzione, l’array `arr` verrà “espanso” in una lista di argomenti.

Nel caso `Math.max`:

```
let arr = [3, 5, 1];
```

```
alert(Math.max(...arr)); // 5 (spread turns array into a list of arguments)
```

Possiamo anche fornire più oggetti iterabili in questo modo:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert(Math.max(...arr1, ...arr2)); // 8
```

Possiamo anche combinare l'operatore di espansione con valori “normali”:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert(Math.max(1, ...arr1, 2, ...arr2, 25)); // 25
```

Inoltre, l'operatore di spread può essere utilizzato anche per fondere array:

```
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15 (0, then arr, then 2, then arr2)
```

Negli esempi sopra abbiamo utilizzato un array per dimostrare l'operatore di espansione, ma funziona correttamente con qualsiasi oggetto iterabile.

Ad esempio, in questo esempio utilizziamo l'operatore di espansione per convertire la stringa in un array di caratteri:

```
let str = "Hello";

alert([...str]); // H,e,l,l,o
```

L'operatore di spread internamente sfrutta gli iteratori per ottenere gli elementi, proprio come `for..of`.

Quindi, per una stringa, `for..of` ritorna dei caratteri e `...str` diventa `"H", "e", "l", "l", "o"`. La lista di caratteri viene passata per inizializzare un array `[...str]`.

Per quest'attività in particolare potremmo anche utilizzare `Array.from`, poiché converte un oggetto iterabile (come una stringa) in un array:

```
let str = "Hello";

// Array.from converts an iterable into an array
alert(Array.from(str)); // H,e,l,l,o
```

Il risultato è lo stesso ottenuto con `[...str]`.

C'è però una sottile differenza tra `Array.from(obj)` e `[...obj]`:

- `Array.from` funziona sia con array che con oggetti iterabili.
- L'operatore di espansione opera solamente su oggetti iterabili.

Quindi, per convertire qualcosa in array, la scelta migliore è `Array.from`.

## Copiare un array/oggetto

Ricordate quando abbiamo parlato del metodo `Object.assign()`?

E' possibile fare la stessa cosa con l'operatore di espansione (spread).

```
let arr = [1, 2, 3];
let arrCopy = [...arr]; // espande l'array in una lista di parametri
 // successivamente inserisce il risultato in un nuovo array

// l'array ha gli stessi contenuti?
alert(JSON.stringify(arr) === JSON.stringify(arrCopy)); // true

// gli array sono uguali?
alert(arr === arrCopy); // false (non contengono lo stesso riferimento)

// la modifica dell'array iniziale non modifica la copia:
arr.push(4);
alert(arr); // 1, 2, 3, 4
alert(arrCopy); // 1, 2, 3
```

Da notare che è possibile fare la stessa cosa per copiare un oggetto:

```
let obj = { a: 1, b: 2, c: 3 };
let objCopy = { ...obj }; // espande l'oggetto in una lista di parametri
 // successivamente inserisce il risultato in un oggetto

// l'oggetto ha gli stessi contenuti?
alert(JSON.stringify(obj) === JSON.stringify(objCopy)); // true

// gli oggetti sono uguali?
alert(obj === objCopy); // false (non contengono lo stesso riferimento)

// modificando l'oggetto iniziale non viene modificata la copia:
obj.d = 4;
alert(JSON.stringify(obj)); // {"a":1,"b":2,"c":3,"d":4}
alert(JSON.stringify(objCopy)); // {"a":1,"b":2,"c":3}
```

Questa modalità di copia di un oggetto è molto più rapida rispetto a `let objCopy = Object.assign({}, obj);` o per un array `let arrCopy = Object.assign([], arr);` per questo preferiamo utilizzarla quando possibile.

## Riepilogo

Quando nel codice incontriamo: " . . . ", potrebbe essere sia l'operatore di resto dei parametri che l'operatore di espansione.

Un modo semplice per distinguere i due casi:

- Quando . . . si trova alla fine della lista dei parametri della funzione, allora è l'operatore "resto dei parametri", il quale raccoglie tutti i parametri forniti (sotto forma di array) alla funzione che non trovano spazio nelle variabili.
- Quando . . . si trova in una chiamata a funzione o situazioni simili, viene chiamato "operatore operatore di espansione", che espande un array in una lista.

Casi d'uso:

- L'operatore parametri di resto viene utilizzato per creare una funzione che accetta un qualsiasi numero di argomenti.
- L'operatore di espansione viene utilizzato per passare un array ad una funzione che richiede una lista di argomenti.

Insieme questi due operatori consentono di lavorare facilmente con le funzioni e i parametri passati.

Tutti gli argomenti di una funzione sono accessibili anche con il metodo "vecchio stile" arguments : un oggetto simil-array.

## Variable scope, closure

JavaScript è un linguaggio fortemente orientato alle funzioni. Fornisce molta libertà. Una funzione può essere creata in qualsiasi momento, copiata su una variabile o passata come argomento ad un'altra funzione e richiamata da qualsiasi punto del codice.

Sappiamo che una funzione può accedere alle variabili esterne, questa caratteristica viene spesso utilizzata.

Cosa accade quando una variabile esterna cambia? La funzione utilizza il valore più recente o quello presente al momento della creazione della funzione?

Inoltre, cosa accade quando una funzione viene spostata in un altro punto del codice e viene richiamata – avrebbe accesso alle variabili esterne della nuova posizione?

In questa situazione linguaggi diversi si comportano in maniera diversa, in questo capitolo ci occuperemo del comportamento di JavaScript.

### Qui parleremo delle variabili let/const

In JavaScript, ci esistono 3 modi per dichiarare una variabile: let , const (metodologie più moderne), e var (metodo utilizzato in passato).

- In questo articolo utilizzeremo let .
- Variabili, dichiarate tramite const , quindi in questo articolo parleremo anche di const .
- Il vecchio var ha alcune differenze importanti, di cui parleremo nell'articolo [Il vecchio "var"](#).

## Blocchi di codice

Se una variabile viene dichiarata all'interno di un blocco di codice `{ . . . }`, questa sarà visibile solamente all'interno di quel blocco di codice.

Ad esempio:

```
{
 // facciamo alcune operazioni con variabili locali che non dovrebbero essere visibili all'esterno

 let message = "Hello"; // visibile solamente all'interno di questo blocco

 alert(message); // Hello
}

alert(message); // Error: message is not defined
```

Possiamo utilizzare i blocchi di codice per isolare pezzi di codice, definendo delle variabili che gli appartengono:

```
{
 // mostra message
 let message = "Hello";
 alert(message);
}

{
 // show another message
 let message = "Goodbye";
 alert(message);
}
```

### **i Ci sarebbe un errore senza blocchi**

Da notare, senza blocchi separati ci sarebbe un errore, nel caso in cui usassimo `let` con un nome di variabile già esistente:

```
// mostra message
let message = "Hello";
alert(message);

// mostra un altro message
let message = "Goodbye"; // Error: variable already declared
alert(message);
```

Per `if`, `for`, `while` e così via, le variabili dichiarate all'interno di `{ . . . }` sono visibili solo al suo interno:

```
if (true) {
 let phrase = "Hello!";
```

```
 alert(phrase); // Hello!
}

alert(phrase); // Error, no such variable!
```

Qui, quando termina `if`, l'espressione `alert` non avrà accesso a `phrase`, quindi verrà emesso un errore.

Questo è ottimo, poiché ci consente di creare variabili locali al blocco di codice, specifiche per un branch di `if`.

The similar thing holds true for `for` and `while` loops:

```
for (let i = 0; i < 3; i++) {
 // la variabile i è visibile solamente all'interno del for
 alert(i); // 0, then 1, then 2
}

alert(i); // Error, no such variable
```

Visually, `let i` is outside of `{...}`. But the `for` construct is special here: the variable, declared inside it, is considered a part of the block.

## Funzioni annidate

Una funzione si definisce “annidata” quando viene creata all'interno di un'altra funzione.

E' molto semplice farlo in JavaScript.

Possiamo utilizzare questo concetto per organizzare il codice, come in questo esempio:

```
function sayHiBye(firstName, lastName) {

 // helper nested function to use below
 function getFullName() {
 return firstName + " " + lastName;
 }

 alert("Hello, " + getFullName());
 alert("Bye, " + getFullName());

}
```

Qui la funzione *annidata* `getFullName()` è stata creata per comodità. Può accedere alle variabili esterne quindi può ritornarne il nome completo. Le funzioni annidate sono abbastanza comuni in JavaScript.

Un'altra cosa interessante, una funzione annidata può essere ritornata: sia come proprietà di un nuovo oggetto (se la funzione esterna crea un oggetto con dei metodi) o come risultato stesso. Può essere salvata e utilizzata da qualsiasi altra parte. Non ha importanza dove, avrà comunque accesso alle stesse variabili esterne.

Nell'esempio sotto, `makeCounter` crea una funzione “contatore” che ritorna il numero successivo ad ogni invocazione:

```

function makeCounter() {
 let count = 0;

 return function() {
 return count++;
 };
}

let counter = makeCounter();

alert(counter()); // 0
alert(counter()); // 1
alert(counter()); // 2

```

Nonostante siano semplici, varianti leggermente modificate di questo codice hanno usi pratici, ad esempio, come [generatore di numeri casuali ↗](#) per generare valori casuali per tests automatici. Quindi l'esempio non è così.

Come funziona? Se creiamo contatori multipli, saranno indipendenti? Come vengono gestite le variabili?

Conoscere queste cose è ottimo per una conoscenza generale di JavaScript è può essere utile nella gestione di scenari più complessi. Quindi proviamo ad entrare più nel dettaglio.

## Lexical Environment

### Here be dragons!

The in-depth technical explanation lies ahead.

As far as I'd like to avoid low-level language details, any understanding without them would be lacking and incomplete, so get ready.

For clarity, the explanation is split into multiple steps.

### Step 1. Variabili

In JavaScript, ogni funzione in esecuzione, blocco di codice `{ . . . }`, e lo script nella sua interezza possiedono un oggetto interno associato (nascosto), anche conosciuto come *Lexical Environment*.

The Lexical Environment object consists of two parts:

1. *Environment Record (Registro d'Ambiente)* – un oggetto che memorizza tutte le variabili locali e le sue proprietà (ed altre informazioni utili come il valore di `this`).
2. Un riferimento al *lexical environment esterno*, quello associato al codice esterno.

**Una variabile è solamente una proprietà di uno speciale oggetto interno, `Environment Record`. “Ottenere o modificare una variabile” significa “ottenere o modificare una proprietà di questo oggetto”.**

In questo semplice esempio senza funzioni, esiste solamente un Lexical Environment:

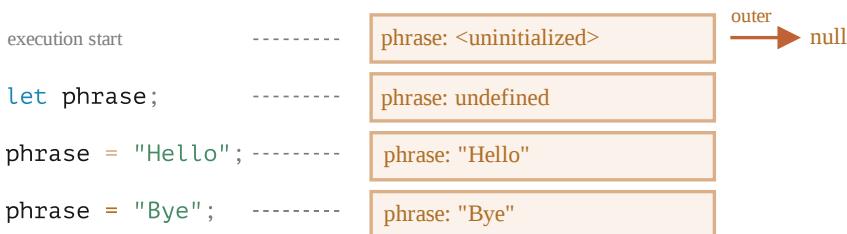


Questo è quello che viene chiamato Lexical Environment *globale*, associato all'interno script.

On the picture above, the rectangle means Environment Record (variable store) and the arrow means the outer reference. The global Lexical Environment has no outer reference, that's why the arrow points to `null`.

Mentre il codice inizia la sua esecuzione e procede, il Lexical Environment cambia.

Qui un codice leggermente più complesso:



I rettangoli nella parte destra dimostrano come il Lexical Environment globale cambia durante l'esecuzione: Rectangles on the right-hand side demonstrate how the global Lexical Environment changes during the execution:

1. Quando il codice inizia la sua esecuzione, il Lexical Environment viene popolato con tutte le variabili dichiarate.
  - Inizialmente, queste sono in uno stato “non inizializzato”. Questo è uno speciale stato interno, significa che JavaScript è a conoscenza dell'esistenza della variabile, ma ci si può fare riferimento fino a quando questa non viene dichiarata con `let`. E' equivalente a dire che la variabile non esiste.
2. Successivamente appare la dichiarazione `let phrase`. Non si ha ancora nessuna assegnazione, quindi il suo valore è `undefined`. Da questo momento in poi possiamo utilizzare la variabile.
3. A `phrase` viene assegnato un valore.
4. Il valore di `phrase` viene modificato.

Per ora tutto sembra semplice, vero?

- Una variabile è una proprietà di uno speciale oggetto interno, associato al blocco/funzione/script in esecuzione.
- Lavorare con le variabili significa concretamente lavorare con le proprietà di un oggetto.

### **i Lexical Environment è un oggetto definito dalla specifica**

“Lexical Environment” è un oggetto definito dalla specifica: esiste solamente in forma “teorica” nella [specificità del linguaggio ↗](#) per descrivere come le cose funzionano. Non abbiamo modo di ottenere questo oggetto nel nostro codice e manipolarlo direttamente.

JavaScript engines also may optimize it, discard variables that are unused to save memory and perform other internal tricks, as long as the visible behavior remains as described.

## Step 2. Dichiarazione di funzioni

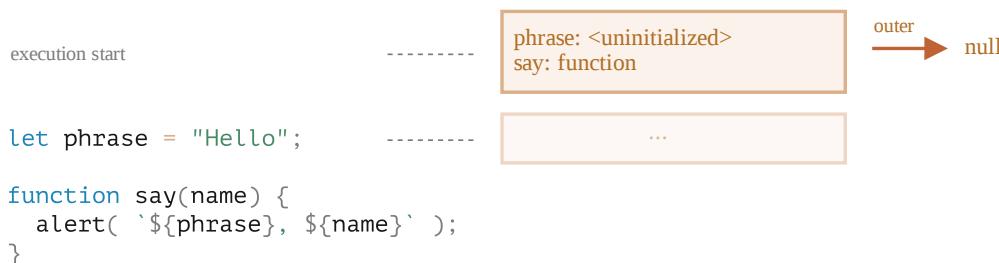
Anche una funzione è un valore, come una variabile.

**La differenza è che la dichiarazione di funzione viene inizializzata istantaneamente.**

Quando viene creato il Lexical Environment, un dichiarazione di funzione diventa immediatamente una funzione pronta per essere utilizzata (a differenza di `let`, che rimane inutilizzabile fino alla sua dichiarazione).

Questo è il motivo per cui possiamo utilizzare una funzione, ancora prima della sua dichiarazione.

Ad esempio, qui vediamo lo stato iniziale del Lexical Environment globale, quando aggiungiamo una funzione:

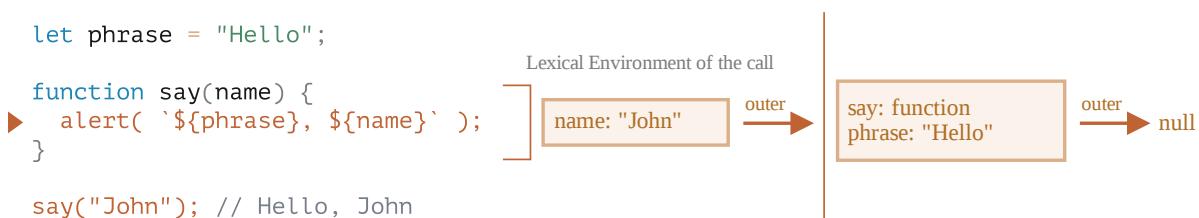


Naturalmente, questo comportamento si applica solamente alle dichiarazioni di funzione, non vale per le espressioni di funzione, dove assegniamo una funzione ad una variabile, come ad esempio `let say = function(name) ...;`

## Step 3. Lexical Environment interno ed esterno

Quando una funzione sta eseguendo, all'inizio della chiamata, viene creato un nuovo Lexical Environment per memorizzare le variabili locali e i parametri della chiamata.

Ad esempio, per `say("John")`, funzionerebbe in questo modo:



Durante l'esecuzione della funzione abbiamo due Lexical Environments: quello interno (utilizzato dalla funzione) e quello esterno (globale):

- Il Lexical Environment interno corrisponde all'esecuzione di `say`. Possiede una sola proprietà: `name`, l'argomento della funzione. Abbiamo invocato `say("John")`, quindi il

valore di `name` è "John".

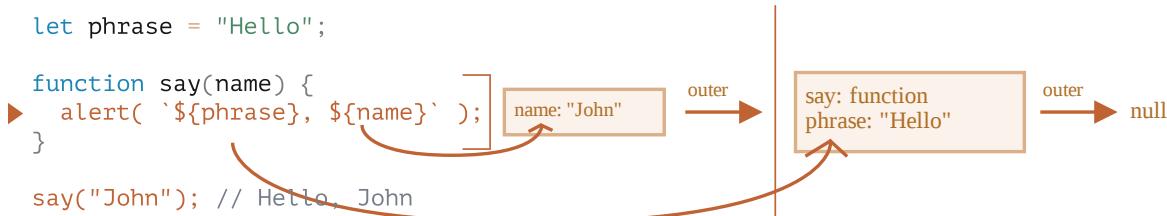
- Il Lexical Environment esterno è quello globale. Possiede la variabile `phrase` e la funzione stessa.

**Quando il codice vuole accedere ad una variabile – questa viene ricercata prima nel Lexical Environment interno, poi in quello esterno, poi quello ancora più esterno e così via fino ad arrivare a quello globale.**

Se una variabile non viene trovata, allora verrà lanciato un errore in strict mode (senza `use strict`, un assegnazione ad una variabile non esistente creerà una nuova variabile globale).

In questo esempio la ricerca procede:

- La variabile `name`, utilizzata da `alert` all'interno di `say`, viene trovata immediatamente nel Lexical Environment interno.
- Quando vuole accedere a `phrase`, non sarà in grado di trovare alcuna variabile `phrase` localmente, quindi seguirà il riferimento verso il Lexical Environment esterno.



#### Step 4. Ritornare una funzione

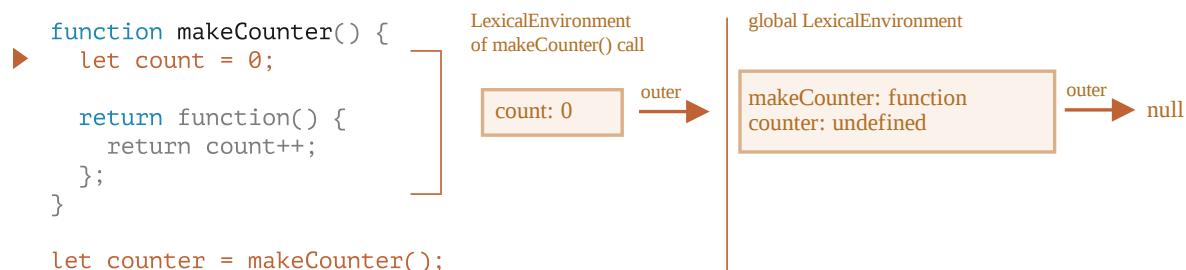
Torniamo all'esempio di `makeCounter`.

```
function makeCounter() {
 let count = 0;

 return function() {
 return count++;
 };
}

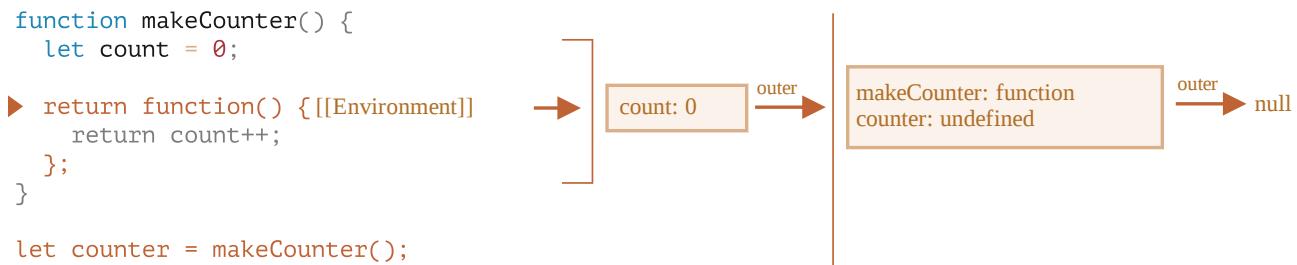
let counter = makeCounter();
```

All'inizio di ogni chiamata a `makeCounter()`, viene creato un nuovo Lexical Environment, dove memorizzare le variabili necessarie all'esecuzione di `makeCounter`.



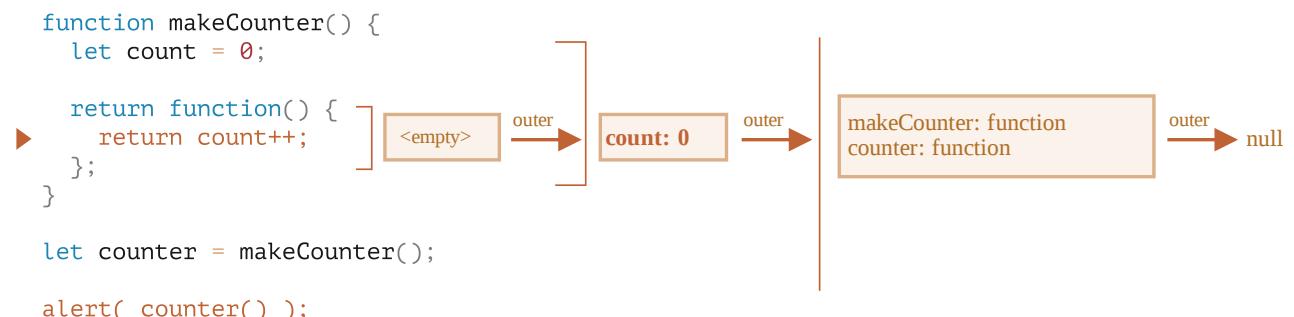
La differenza è che durante l'esecuzione di `makeCounter()`, viene creata una piccola funzione annidata: `return count++`. Non viene eseguita subito, viene solamente creata.

Tutte le funzioni ricordano il Lexical Environment in cui vengono create. Tecnicamente, non c'è nulla di magico: tutte le funzione possiedono la proprietà nascosta `[[Environment]]`, che memorizza il riferimento al Lexical Environment in cui la funzione è stata creata:



Quindi, `counter.[[Environment]]` possiede il riferimento al Lexical Environment `{count: 0}`. Questo è il modo in cui le funzioni memorizzano il contesto in cui sono state create, non ha importanza il posto in cui verranno chiamate. Il riferimento `[[Environment]]` viene impostato a tempo di creazione della funzione e non viene più modificato.

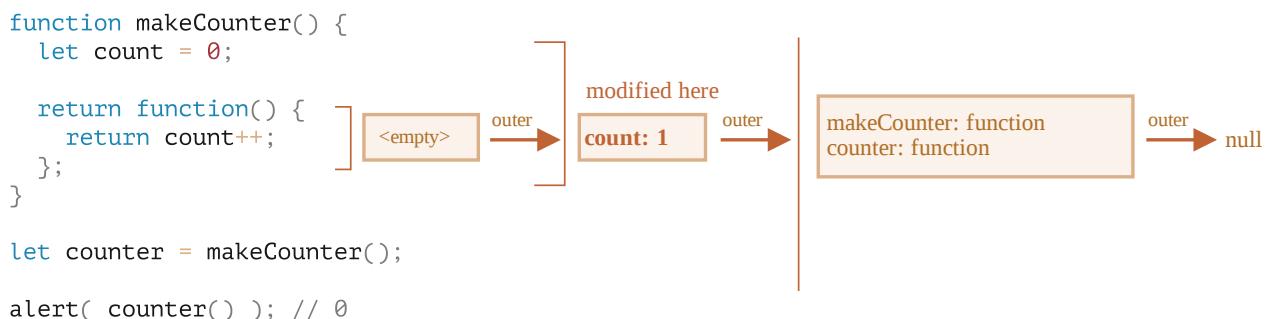
Più tardi, quando viene chiamato `counter()`, verrà creato un nuovo Lexical Environment locale, in cui verrà memorizzato il riferimento al Lexical Environment esterno `counter.[[Environment]]`:



Quindi, quando il codice all'interno di `counter()` cercherà la variabile `count` nel suo Lexical Environment (vuoto, poiché non possiede variabili locali), poi cercherà nel Lexical Environment esterno, quindi quello della chiamata `makeCounter()`, dove riuscirà a trovare la variabile e potrà modificarla.

### Una variabile viene aggiornata nel Lexical Environment in cui si trova.

Qui vediamo lo stato dopo l'esecuzione:



If we call `counter()` multiple times, the `count` variable will be increased to 2, 3 and so on, at the same place.

## Closure

Esiste un termine generale in programmazione, "closure", che gli sviluppatori dovrebbero conoscere.

Una [closure](#) è una funzione che ricorda le sue variabili esterne ed è in grado di accedervi. In alcuni linguaggi questo non è possibile, oppure è richiesto che la funzione venga scritta in un determinato modo. Ma come spiegato sopra, in JavaScript, tutte le funzioni sono closure di natura (esiste una sola eccezione, che verrà trattata nel capitolo [La sintassi "new Function"](#)).

That is: they automatically remember where they were created using a hidden `[[Environment]]` property, and then their code can access outer variables.

When on an interview, a frontend developer gets a question about "what's a closure?", a valid answer would be a definition of the closure and an explanation that all functions in JavaScript are closures, and maybe a few more words about technical details: the `[[Environment]]` property and how Lexical Environments work.

## Garbage collection

Solitamente, un Lexical Environment viene rimosso dalla memoria insieme a tutte le sue variabili dopo che la funzione ha completato la sua esecuzione. Questo avviene perché non si hanno più riferimenti ad essa. Come ogni altro oggetto in JavaScript, viene mantenuto in memoria solamente finché risulta essere raggiungibile.

Tuttavia, se una funzione annidata risulta essere ancora raggiungibile, allora avremmo una proprietà `[[Environment]]` che fa riferimento al Lexical Environment.

In questo caso il Lexical Environment risulta essere ancora raggiungibile dopo aver completato l'esecuzione, quindi rimane in memoria.

Ad esempio: `return function() { alert(value); } }`

`let g = f(); // g.[[Environment]] memorizza un riferimento al Lexical Environment // della corrispondente chiamata a f()`

Da notare che se `f()` viene chiamata più volte, e la funzione risultata viene memorizzata, allo

- Da notare che se `f()` viene invocata più volte, e vengono memorizzate delle funzioni, allora

```
```js
function f() {
    let value = Math.random();

    return function() { alert(value); };
}

// 3 functions in array, every one of them links to Lexical Environment
// from the corresponding f() run
//           LE   LE   LE
let arr = [f(), f(), f()];
````
```

Nel codice sotto, dopo aver rimosso la funzione annidata, il Lexical Environment interno (e anch

```
```js
```

```
function f() {
  let value = 123;

  return function() {
    alert(value);
  }
}

let g = f(); // finché g esiste, il valore rimane in memoria

g = null; // ...and now the memory is cleaned up
```

```

### ### Ottimizzazioni nel mondo reale

Come abbiamo visto, in teoria fino a che una funzione rimane viva, anche tutte le variabili este

Ma nella pratica, i motori JavaScript cercano di ottimizzare questa situazione. Monitorano l'uti

\*\*Un importante effetto collaterale in V8 (Chrome, Opera) è che queste variabili non sono dispon

Provate ad eseguire il codice d'esempio qui sotto in Chrome con aperta la finestra degli strumen

Quando si arresta, scrivete nella console `alert(value)`.

```
```js run
function f() {
  let value = Math.random();

  function g() {
    debugger; // in console: type alert(value); No such variable!
  }

  return g;
}

let g = f();
g();
```

Come avrete notato – non esiste questa variabile! In teoria, dovrebbe essere accessibile, ma il motore la ha rimossa.

Questo può portare a divertenti (soprattutto se avete poco tempo a disposizione) problemi in fase di debugging. Uno di questi – potremmo visualizzare una variabile esterna che ha lo stesso nome, piuttosto di quella desiderata:

```
let value = "Surprise!";

function f() {
  let value = "the closest value";

  function g() {
    debugger; // in console: type alert(value); Surprise!
  }

  return g;
}
```

```
let g = f();
g();
```

⚠ Ci incontriamo!

Questa caratteristica di V8 va tenuta a mente. Se state facendo debugging con Chrome/Opera, presto o tardi vi ci imbatterete.

Questo non è un problema del debugger, ma piuttosto una caratteristica di V8. In futuro potrebbe essere risolta. Potrete sempre testarlo provando ad eseguire il codice sopra.

✓ Esercizi

Does a function pickup latest changes?

importanza: 5

The function `sayHi` uses an external variable name. When the function runs, which value is it going to use?

```
let name = "John";

function sayHi() {
  alert("Hi, " + name);
}

name = "Pete";

sayHi(); // what will it show: "John" or "Pete"?
```

Such situations are common both in browser and server-side development. A function may be scheduled to execute later than it is created, for instance after a user action or a network request.

So, the question is: does it pick up the latest changes?

[Alla soluzione](#)

Which variables are available?

importanza: 5

The function `makeWorker` below makes another function and returns it. That new function can be called from somewhere else.

Will it have access to the outer variables from its creation place, or the invocation place, or both?

```
function makeWorker() {
  let name = "Pete";

  return function() {
    alert(name);
  };
}
```

```
let name = "John";

// create a function
let work = makeWorker();

// call it
work(); // what will it show?
```

Which value it will show? “Pete” or “John”?

[Alla soluzione](#)

Sono indipendenti i contatori?

importanza: 5

Qui costruiamo due contatori: `counter` e `counter2` utilizzando la stessa funzione `makeCounter`.

Sono indipendenti? Cosa mostrerà il secondo contatore? `0, 1` oppure `2, 3` o qualcosa altro?

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();
let counter2 = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1

alert( counter2() ); // ?
alert( counter2() ); // ?
```

[Alla soluzione](#)

Oggetto contatore

importanza: 5

Qui l’oggetto contatore viene creato con il costruttore.

Funziona? Cosa mostra?

```
function Counter() {
  let count = 0;

  this.up = function() {
    return ++count;
};
```

```
this.down = function() {
  return --count;
};

let counter = new Counter();

alert( counter.up() ); // ?
alert( counter.up() ); // ?
alert( counter.down() ); // ?
```

[Alla soluzione](#)

Funzione interna ad if

Guardate il codice. Quale sarà il risultato della chiamata all'ultima riga?

```
let phrase = "Hello";

if (true) {
  let user = "John";

  function sayHi() {
    alert(`#${phrase}, ${user}`);
  }
}

sayHi();
```

[Alla soluzione](#)

Somma con le closure

importanza: 4

Scrivete la funzione `sum` che funziona in questo modo: `sum(a)(b) = a+b`.

Esattamente, due parentesi tonde (non è un errore).

Ad esempio:

```
sum(1)(2) = 3
sum(5)(-1) = 4
```

[Alla soluzione](#)

Is variable visible?

importanza: 4

What will be the result of this code?

```
let x = 1;
```

```
function func() {  
    console.log(x); // ?  
  
    let x = 2;  
}  
  
func();
```

P.S. There's a pitfall in this task. The solution is not obvious.

[Alla soluzione](#)

Filter su funzioni

importanza: 5

Abbiamo a disposizione un metodo integrato `arr.filter(f)` per gli array. Questo filtra tutti gli elementi attraverso la funzione `f`. Se ritorna `true`, allora quell'elemento viene ritornato.

Create un insieme di filtri “pronti all’uso”:

- `inBetween(a, b)` – tra `a` e `b` o uguale.
- `inArray([...])` – contenuto nell’array.

Il loro utilizzo dovrebbe essere:

- `arr.filter(inBetween(3, 6))` – seleziona solo i valori compresi tra 3 e 6.
- `arr.filter(inArray([1, 2, 3]))` – seleziona solo gli elementi che corrispondono a `[1, 2, 3]`.

Ad esempio:

```
/* .. your code for inBetween and inArray */  
let arr = [1, 2, 3, 4, 5, 6, 7];  
  
alert( arr.filter(inBetween(3, 6)) ); // 3,4,5,6  
  
alert( arr.filter(inArray([1, 2, 10])) ); // 1,2
```

Apri una sandbox con i test. [↗](#)

[Alla soluzione](#)

Ordinare per campo

importanza: 5

Abbiamo un array di oggetti da ordinare:

```
let users = [  
    { name: "John", age: 20, surname: "Johnson" },  
    { name: "Pete", age: 18, surname: "Peterson" },  
    { name: "Mike", age: 22, surname: "Miller" }]
```

```
{ name: "Ann", age: 19, surname: "Hathaway" }  
];
```

Il modo più classico per farlo sarebbe:

```
// by name (Ann, John, Pete)  
users.sort((a, b) => a.name > b.name ? 1 : -1);  
  
// by age (Pete, Ann, John)  
users.sort((a, b) => a.age > b.age ? 1 : -1);
```

Possiamo renderlo anche più breve, ad esempio:

```
users.sort(byField('name'));  
users.sort(byField('age'));
```

Quindi, piuttosto che scrivere una funzione, utilizziamo `byField(fieldName)`.

Scrivete la funzione `byField`.

Apri una sandbox con i test. ↗

[Alla soluzione](#)

Funzione crea eserciti

importanza: 5

Il seguente codice crea un array di `shooters`.

Ogni funzione è pensata per ritornare il numero, Ma qualcosa non va...

```
function makeArmy() {  
  let shooters = [];  
  
  let i = 0;  
  while (i < 10) {  
    let shooter = function() { // shooter function  
      alert( i ); // should show its number  
    };  
    shooters.push(shooter);  
    i++;  
  }  
  
  return shooters;  
}  
  
let army = makeArmy();  
  
army[0](); // the shooter number 0 shows 10  
army[5](); // and number 5 also outputs 10...  
// ... all shooters show 10 instead of their 0, 1, 2, 3...
```

Perché tutti gli eserciti possiedono lo stesso numero di militari? Modificate il codice in modo tale che funzioni correttamente.

Apri una sandbox con i test. ↗

Alla soluzione

Il vecchio "var"

💡 Questo articolo è utile per la comprensione dei vecchi script

Le informazioni contenute in questo articolo sono utili per la comprensione dei vecchi script.

Non è il modo corretto di scrivere il codice oggi.

Nei primi capitoli in cui abbiamo parlato di [variabili](#), abbiamo menzionato tre diversi tipi di dichiarazione:

1. `let`
2. `const`
3. `var`

La dichiarazione `var` è molto simile a `let`. La maggior parte delle volte possiamo sostituire `let` con `var` o vice-versa, e lo script continuerebbe a funzionare senza problemi:

```
var message = "Hi";
alert(message); // Hi
```

But internally `var` is a very different beast, that originates from very old times. It's generally not used in modern scripts, but still lurks in the old ones.

If you don't plan on meeting such scripts you may even skip this chapter or postpone it.

E' però importante capire le differenze durante la migrazione dei vecchi script da `var` a `let`, per evitare errori.

"var" non ha uno scope di blocco

Le variabili dichiarate tramite `var` possono essere: locali alla funzione oppure globali.

Ad esempio:

```
if (true) {
  var test = true; // utilizziamo "var" piuttosto di "let"
}

alert(test); // vero, la variabile vive dopo if
```

Se avessimo utilizzato `let test` invece di `var test`, allora la variabile sarebbe stata visibile solo all'interno dell'`if`.

```
if (true) {  
  let test = true; // use "let"  
}  
  
alert(test); // ReferenceError: test is not defined
```

La stessa cosa accade con i cicli, `var` non può essere locale ad un blocco/ciclo:

```
for (var i = 0; i < 10; i++) {  
  var one = 1;  
  // ...  
}  
  
alert(i); // 10, "i" è visibile anche dopo il ciclo, è una variabile globale  
alert(one); // 1, "one" è visibile anche dopo il ciclo, è una variabile globale
```

Se un blocco di codice si trova all'interno di una funzione, allora `var` diventa una variabile a livello di funzione:

```
function sayHi() {  
  if (true) {  
    var phrase = "Hello";  
  }  
  
  alert(phrase); // funziona  
}  
  
sayHi();  
alert(phrase); // Errore: phrase non è definito
```

Come possiamo vedere, `var` passa attraverso `if`, `for` o altri blocchi di codice. Questo accade perché molto tempo fa i blocchi JavaScript non possedevano un Lexical Environments. E `var` ne è un ricordo.

“var” tollera dichiarazioni multiple

Se proviamo a ri-dichiarare la stessa variabile con `let` nello stesso scope, avremmo un errore:

```
let user;  
let user; // SyntaxError: 'user' has already been declared
```

Con `var`, possiamo ri-dichiarare una variabile quante volte vogliamo. Se proviamo ad utilizzare `var` con una variabile già dichiarata, esso verrà semplicemente ignorato e la variabile verrà normalmente riassegnata:

```
var user = "Pete";  
  
var user = "John"; // qui "var" non fa nulla (già dichiarata)  
// ...non emetterà nessun errore  
  
alert(user); // John
```

Le variabili “var” possono essere dichiarate dopo il loro utilizzo

Le dichiarazioni con `var` vengono processate quando la funzione inizia (o lo script, nel caso delle variabili globali).

In altre parole, le variabili `var` sono definite dall'inizio della funzione, non ha importanza dove vengano definite (ovviamente non vale nel caso di funzioni annidate).

Guardate questo esempio:

```
function sayHi() {  
    phrase = "Hello";  
  
    alert(phrase);  
  
    var phrase;  
}  
sayHi();
```

...E' tecnicamente la stessa cosa di (spostando `var phrase`):

```
function sayHi() {  
    var phrase;  
  
    phrase = "Hello";  
  
    alert(phrase);  
}  
sayHi();
```

...O anche di questa (ricordate, i blocchi di codice vengono attraversati dallo scope della variabile):

```
function sayHi() {  
    phrase = "Hello"; // (*)  
  
    if (false) {  
        var phrase;  
    }  
  
    alert(phrase);  
}  
sayHi();
```

Questo comportamento viene chiamato “sollevamento”, perché tutte `var` vengono “sollevate” fino all’inizio della funzione.

Quindi nell’esempio sopra, `if (false)` il ramo non eseguirà mai, ma non ha importanza. La `var` all’interno viene processata all’inizio della funzione, quindi quando ci troviamo in `(*)` la variabile esiste.

Le dichiarazioni vengono sollevate, le assegnazioni no.

Lo dimostriamo con un esempio, come quello seguente:

```
function sayHi() {  
    alert(phrase);  
  
    var phrase = "Hello";  
}  
  
sayHi();
```

La riga `var phrase = "Hello"` può essere suddivisa in due:

1. Dichiarazione della variabile `var`
2. Assegnazione della variabile con `=`.

La dichiarazione viene processata all’inizio della funzione (“sollevata”), l’assegnazione invece ha luogo sempre nel posto in cui appare. Quindi il codice funziona in questo modo:

```
function sayHi() {  
    var phrase; // la dichiarazione viene processata...  
  
    alert(phrase); // undefined  
  
    phrase = "Hello"; // ...assegnazione - quando viene raggiunta dal flusso d'esecuzione.  
}  
  
sayHi();
```

Il fatto che la dichiarazione di `var` venga processata all’inizio della funzione, ci consente di farne riferimento in qualsiasi punto. Ma la variabile rimane `undefined` fino all’assegnazione.

In entrambi gli esempi sopra `alert` esegue senza errori, poiché la variabile `phrase` esiste. Il suo valore però non gli è ancora stato assegnato, quindi viene mostrato `undefined`.

IIFE

In passato, poiché esisteva solo `var`, che non consentiva di definire variabili con visibilità a livello di blocco, i programmatore hanno inventato un modo per emulare questa situazione. Quello che facevano fu chiamato “immediately-invoked function expressions” (espressioni di funzioni invocate immediatamente, abbreviato come IIFE).

E’ qualcosa che dovremmo evitare oggi, ma è possibile trovare questo trucco nei vecchi script.

Una IIFE viene scritta in questo modo:

```
(function() {  
  var message = "Hello";  
  
  alert(message); // Hello  
})();
```

Qui, un'espressione di funzione viene creata ed immediatamente chiamata. Quindi il codice esegue nel modo giusto, e possiede le sue variabili private.

L'espressione di funzione è avvolta dalle parentesi `(function { . . . })`, poiché quando JavaScript incontra `"function"` nel flusso principale del codice, lo interpreta come l'inizio di una dichiarazione di funzione. Ma una dichiarazione di funzione deve avere un nome, quindi questo tipo di codice darebbe un errore:

```
// Proviamo a dichiarare ed invocare immediatamente una funzione  
function() { // <-- Errore di sintassi: La dichiarazione di funzione richiede un nome  
  
  var message = "Hello";  
  
  alert(message); // Hello  
}();
```

Anche se dovessimo pensare di aggiungere un nome, questo codice non funzionerebbe, poiché JavaScript non consente di invocare immediatamente le funzioni dichiarate:

```
// errore di sintassi a causa delle parentesi ()  
function go() {  
  
}(); // <-- non è possibile invocare una dichiarazione di funzione immediatamente
```

Quindi, le parentesi intorno alla funzione sono un trucco per mostrare a JavaScript che la funzione viene creata in un altro contesto, e quindi è un'espressione di funzione: la quale non richiede nome e può essere invocata immediatamente.

Esistono altri modi oltre alle parentesi per dire a JavaScript che intendiamo definire un'espressione di funzione:

```
// Altri modi per creare una IIFE  
  
(function() {  
  alert("Parentheses around the function");  
})();  
  
(function() {  
  alert("Parentheses around the whole thing");  
}());  
  
!function() {  
  alert("Bitwise NOT operator starts the expression");  
}();
```

```
+function() {
  alert("Unary plus starts the expression");
}();
```

In tutti gli esempi illustrati stiamo dichiarando un'espressione di funzione invocandola immediatamente. Lasciatemelo ripetere: al giorno d'oggi non c'è alcun motivo di scrivere codice del genere.

Riepilogo

Ci sono due principali differenze tra `var` e `let/const`:

1. `var` non hanno uno scope locale al blocco, sono infatti visibili a livello di funzione.
2. La dichiarazione di `var` viene processata all'inizio della funzione.

C'è un ulteriore differenza di minore importanza legata all'oggetto globale, che andremo ad analizzare nel prossimo capitolo.

L'insieme di queste differenze fa sì che `var` venga considerato uno svantaggio. Come prima cosa, non possiamo creare delle variabili locali al blocco. Il "sollevamento" genera solamente confusione. Quindi, negli script più recenti `var` viene utilizzato solamente in casi eccezionali.

Oggetto globale

L'oggetto globale fornisce variabili e funzioni che sono accessibili in qualsiasi punto. Principalmente quelle integrate dal linguaggio o fornite dall'ambiente.

In un browser l'ambiente si chiama `window`, per Node.js viene detto `global`, negli altri ambienti si usano diversi termini.

Recentemente, è stato aggiunto al linguaggio `globalThis`, come nome standart per l'oggetto globale, il quale dovrebbe essere supportato da tutti gli ambienti. In alcuni browser, ad esempio Edge, `globalThis` non è ancora supportato.

Tutte le proprietà dell'oggetto globale possono essere raggiunte direttamente:

```
alert("Hello");

// la stessa cosa
window.alert("Hello");
```

In un browser le variabili globali dichiarate con `var` diventano proprietà dell'oggetto globale:

```
var gVar = 5;

alert(window.gVar); // 5 (diventa una proprietà dell'oggetto globale)
```

Non affidatevi a questo! Questo comportamento esiste solamente per retrocompatibilità. Gli script moderni utilizzano i moduli JavaScript, che si comportano in maniera differente. Li studieremo più avanti nel capitolo [moduli JavaScript](#).

Inoltre, la dichiarazione di variabili in stile moderno, tramite `let` e `const` non hanno questo tipo di comportamento:

```
let gLet = 5;  
  
alert(window.gLet); // undefined (non diventa una proprietà dell'oggetto globale)
```

If a value is so important that you'd like to make it available globally, write it directly as a property:

```
// rendiamo globali le informazioni dell'utente corrente, per rendere accessibili in qualsiasi punto  
window.currentUser = {  
    name: "John"  
};  
  
// in un qualsiasi altro punto del codice  
alert(currentUser.name); // John  
  
// oppure, se abbiamo una variabile locale denominata "currentUser"  
// la preleviamo da window esplicitamente (più sicuro!)  
alert(window.currentUser.name); // John
```

Detto ciò, l'utilizzo di variabili globali è da evitare. Dovrebbero esserci sempre il minor numero di variabili globali possibili. Il design del codice in cui una funzione richiede un `input` e ritorna un `output` risulta essere molto più chiaro, e molto meno propenso ad errori.

Utilizzo di polyfill

Utilizziamo l'oggetto globale per testare il supporto delle funzionalità introdotte da linguaggio.

Ad esempio, potremmo testare se l'oggetto integrato `Promise` esiste (nei vecchi browser non lo troverete):

```
if (!window.Promise) {  
    alert("Your browser is really old!");  
}
```

Se non è presente (ipotizziamo di trovarci in un vecchio browser), possiamo creare un “polyfill” (contenitore): che consiste nell'aggiungere funzionalità moderne del linguaggio, che non sono supportate.

```
if (!window.Promise) {  
    window.Promise = ... // implementazione della caratteristica mancante  
}
```

Riepilogo

- L'oggetto globale contiene le variabili che dovrebbero essere accessibili ovunque.

Incluse quelle integrate in JavaScript, come `Array` e valori specifici dell'ambiente, come `window.innerHeight` – l'altezza della finestra nei browser.

- L'oggetto globale ha un nome universale `globalThis`.

...Ma è più facile trovarne riferimenti alla “vecchia maniera”, quindi con nomi specifici dell'ambiente, come `window` (browser) e `global` (Node.js). Poiché `globalThis` è un aggiornamento recente, non è ancora supportato da Edge (ma può essere aggiunto con la tecnica polyfill).

- Dovremmo memorizzare valori nell'oggetto globale solamente se questi hanno realmente uno scopo globale.
- In ambiente browser, senza l'utilizzo dei `moduli`, una variabile globale dichiarata tramite `var` diventa una proprietà dell'oggetto globale.

Per rendere il codice più semplice da interpretare e aggiornare, dovremmo accedere all'oggetto globale come `window.x`.

Oggetto funzione, NFE

Come già sappiamo, in JavaScript le funzioni sono valori.

Inoltre ogni valore ha un tipo. Di che tipo è una funzione?

In JavaScript, le funzioni sono oggetti.

Un ottimo modo per pensare alle funzioni è quello di immaginarle come “oggetti attivi” (che compiono azioni). Oltre a invocarli, possiamo trattarli come veri e propri oggetti: aggiungendo/rimuovendo proprietà, passarli per riferimento.

La proprietà “name”

Gli oggetti funzione contengono alcune proprietà utili.

Ad esempio, il nome di una funzione è accessibile tramite la proprietà “name”:

```
function sayHi() {  
  alert("Hi");  
}  
  
alert(sayHi.name); // sayHi
```

Inoltre l'assegnazione della proprietà `name` è intelligente. Funziona anche se dichiariamo la funzione per assegnazione ad una variabile:

```
let sayHi = function() {  
  alert("Hi");  
};  
  
alert(sayHi.name); // sayHi (funziona!)
```

Funziona anche nel caso in cui l'assegnazione viene effettuata come valore di default:

```
function f(sayHi = function() {}) {
  alert(sayHi.name); // sayHi (funziona!)
}

f();
```

Nelle specifiche, questa caratteristica viene chiamata “contextual name” (“nome prelevato dal contesto”). Se la funzione non ne fornisce uno, allora durante l’assegnazione questo viene ricavato dal contesto.

Anche i metodi dell’oggetto possiedono la proprietà `name`:

```
let user = {

  sayHi() {
    // ...
  },

  sayBye: function() {
    // ...
  }
}

alert(user.sayHi.name); // sayHi
alert(user.sayBye.name); // sayBye
```

Non sta accadendo nulla di magico. Anche se ci sono dei casi in cui non c’è alcun modo di ricavare il nome dal contesto. In questi casi, la proprietà `name` sarà vuota, come nell’esempio:

```
// funzione creata all'interno dell'array
let arr = [function() {}];

alert( arr[0].name ); // <stringa vuota>
// il motore non ha alcun modo di ricavare il nome corretto, per questo sarà vuoto
```

Nella pratica però, la maggior parte delle funzioni possiedono un nome.

La proprietà “length”

Esiste un’altra proprietà molto utile, “length” che ritorna il numero di parametri della funzione, ad esempio:

```
function f1(a) {}
function f2(a, b) {}
function many(a, b, ...more) {}

alert(f1.length); // 1
alert(f2.length); // 2
alert(many.length); // 2
```

Da questo esempio possiamo notare che i parametri di resto non vengono contati.

La proprietà `length` viene spesso utilizzata per ispezionare un funzione che opera su altre funzioni.

Ad esempio, nel codice sotto la funzione `ask` accetta una `question` da porre, e un numero arbitrario di `handler` (gestori) da invocare.

Una volta che l'utente ha fornito la risposta, la funzione invoca gli handlers. Possiamo fornire due tipi di handlers:

- Una funzione con zero argomenti, che viene in invocata solamente nel caso in cui l'utente fornisca una risposta positiva.
- Una funzione con argomenti, che viene invocata in entrambi i casi e ritorna una risposta.

L'idea è quella di avere un semplice handles senza argomenti, per gestire i casi positivi (la variante più frequente), ma siamo comunque in grado di fornire un gestore universale.

Per invocare `handlers` nel modo corretto, esaminiamo la proprietà `length`:

```
function ask(question, ...handlers) {
  let isYes = confirm(question);

  for(let handler of handlers) {
    if (handler.length == 0) {
      if (isYes) handler();
    } else {
      handler(isYes);
    }
  }

}

// per risposte positive, entrambi gli handler vengono chiamati
// per le risposte negative, solamente la seconda
ask("Question?", () => alert('You said yes'), result => alert(result));
```

Questo è un caso particolare di quello che viene chiamato [polimorfismo ↗](#) – trattare gli argomenti in maniera differente in base al loro tipo, o nel nostro caso in base a `length`. Quest'idea ha delle utili applicazioni nelle librerie JavaScript.

Proprietà aggiuntive

Possiamo anche aggiungere delle proprietà.

In questo esempio aggiungiamo la proprietà `counter` per tenere traccia delle chiamate totali:

```
function sayHi() {
  alert("Hi");

  // andiamo a contare quante volte verrà invocata
  sayHi.counter++;
}

sayHi.counter = 0; // valore iniziale
```

```
sayHi(); // Hi  
sayHi(); // Hi  
  
alert(`Called ${sayHi.counter} times`); // Chiamata 2 volte
```

⚠ Una proprietà non è una variabile

Una proprietà assegnata ad una funzione, come `sayHi.counter = 0` non definisce una variabile locale `counter`. In altre parole, una proprietà `counter` ed una variabile `let counter` sono due cose separate.

Possiamo quindi trattare una funzione come un oggetto, memorizzare proprietà, ma non avranno alcun effetto sull'esecuzione. Le variabili non utilizzano mai le proprietà della funzione e vice versa. Sono come due mondi paralleli.

Le proprietà delle funzioni possono rimpiazzare le closure in alcun casi. Ad esempio, possiamo riscrivere la funzione contatore del capitolo [Variable scope, closure](#) sfruttando una proprietà della funzione:

```
function makeCounter() {  
  // piuttosto di:  
  // let count = 0  
  
  function counter() {  
    return counter.count++;  
  };  
  
  counter.count = 0;  
  
  return counter;  
}  
  
let counter = makeCounter();  
alert(counter()); // 0  
alert(counter()); // 1
```

Ora `count` viene memorizzato direttamente nella funzione, non nel Lexical Environment esterno.

Questa soluzione è migliore o peggiore rispetto ad una closure?

La principale differenza è che se il valore di `count` sta su una variabile esterna, allora il codice al suo esterno non vi può accedere. Solamente le funzioni annidate possono modificarla. Se invece questa è legata ad una funzione, possono accadere cose simili:

```
function makeCounter() {  
  
  function counter() {  
    return counter.count++;  
  };  
  
  counter.count = 0;  
  
  return counter;
```

```
}
```



```
let counter = makeCounter();
```



```
counter.count = 10;
```



```
alert( counter() ); // 10
```

Quindi non c'è alcuna scelta migliore, ogni caso va analizzato.

Named Function Expression

Named Function Expression (Espressione di Funzione con Nome), o NFE, è un termine per riferirsi alle espressioni di funzioni che hanno un nome.

Ad esempio, prendiamo una normale espressione di funzione:

```
let sayHi = function(who) {
```



```
  alert(`Hello, ${who}`);
```



```
};
```

E diamogli un nome:

```
let sayHi = function func(who) {
```



```
  alert(`Hello, ${who}`);
```



```
};
```

Abbiamo ottenuto qualcosa? Qual'è lo scopo di aggiungere il nome "func"?

Innanzitutto vale la pena notare che continua ad essere un'espressione di funzione. Aggiungere il nome "func" dopo `function` non la rende una dichiarazione di funzione, perché viene comunque creata come una parte di un'assegnazione.

Quindi aggiungere un nome non provoca assolutamente nessuno danno.

La funzione rimane comunque disponibile come `sayHi()`:

```
let sayHi = function func(who) {
```



```
  alert(`Hello, ${who}`);
```



```
};
```



```
sayHi("John"); // Hello, John
```

Ci sono due cose che rendono speciale il nome `func`:

1. Consente alla funzione di riferirsi a se stessa internamente.
2. Non è visibile all'esterno della funzione.

Ad esempio, la funzione `sayHi` qui sotto, chiama nuovamente se stessa con "Guest" se non viene fornito alcun `who`:

```

let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // utilizza func per richiamare se stessa
  }
};

sayHi(); // Hello, Guest

// Questo non funziona:
func(); // Errore, func non è definita (non è visibile all'esterno)

```

Perché utilizziamo `func`? Forse potremmo semplicemente chiamare `sayHi` per le chiamate annidate?

In realtà, in molti è possibile:

```

let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest");
  }
};

```

Il problema con questo codice è che il valore di `sayHi` potrebbe cambiare. La funzione potrebbe essere trasferita su un'altra variabile, e il codice diventerebbe errato:

```

let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest"); // Errore: sayHi non è una funzione
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Errore, la chiamata annidata a sayHi non è più in funzione!

```

Questo accade perché la funzione prende `sayHi` dal suo lexical environment esterno. Non c'è alcun `sayHi` locale, quindi viene utilizzata la variabile esterna. Nell'esempio sopra al momento della chiamata il valore di `sayHi` è `null`.

La possibilità di aggiungere un nome ad un'espressione di funzione è pensato proprio per risolvere questo tipo di problemi.

Sfruttiamo questa caratteristica per sistemare il codice:

```

let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  }
};

```

```

    } else {
      func("Guest"); // Ora è tutto okay
    }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Hello, Guest (la chiamata annidata funziona correttamente)

```

Ora funziona, perché il nome "func" è locale alla funzione. Non viene prelevato dall'esterno. Le specifiche garantiscono che in questo modo si avrà sempre un riferimento alla funzione corrente.

Il codice esterno continuerà ad utilizzare la variabile `sayHi` o `welcome`. E `func` servirà da "nome interno" della funzione.

Tutto questo non vale per la dichiarazione di funzione

La caratteristica del "nome interno" è disponibile solamente per le espressioni di funzione, non per le dichiarazioni di funzione. Per le dichiarazioni di funzione, non c'è alcun modo per aggiungere un ulteriore "nome interno".

Talvolta, nel caso in cui avessimo bisogno di un nome interno, potrebbe essere sensato riscrivere la dichiarazione di funzione come espressione di funzione.

Riepilogo

Le funzioni sono oggetti.

Qui abbiamo visto le proprietà:

- `name` – il nome della funzione. Non esiste solamente quando viene fornito nella definizione della funzione, ma anche nel caso di assegnazioni o proprietà di un oggetto.
- `length` – il numero di argomenti nella definizione della funzione. I parametri di resto non vengono contati.

Se una funzione viene dichiarata come espressione di funzione (non nel principale flusso di codice), e possiede un nome, questa viene definita una Named Function Expression. Il nome può essere utilizzato internamente per auto-riferimenti, per chiamate ricorsive e altri contesti simili.

Inoltre, una funzione può possedere diverse proprietà aggiuntive. Molte librerie JavaScript fanno largo uso di questa caratteristica.

Queste creano una funzione "principale" e ci attaccano molte altre funzioni di "supporto". Ad esempio la libreria [jquery](#) definisce una funzione chiamata `$`. La libreria [lodash](#) definisce una funzione `_`. E ci aggiunge `_.clone`, `_.keyBy` e altre proprietà (vedi la [documentazione](#)). In realtà, lo fanno anche per diminuire la sporcizia nello spazio globale, in questo modo una libreria fornisce una sola variabile globale. Questo riduce la probabilità di conflitti tra nomi.

Quindi una funzione, oltre ad essere utile, può fornire un insieme di altre funzionalità grazie alle proprietà.

Esercizi

Impostare e decrementare il contatore

importanza: 5

Modificate il codice di `makeCounter()` in modo tale che il counter possa essere anche decrementato e reimpostato:

- `counter()` dovrebbe ritornare il prossimo numero (come già fa).
- `counter.set(value)` dovrebbe impostare il contatore a `value`.
- `counter.decrease()` dovrebbe decrementare il contatore di 1.

Vedi il codice in sandbox per un esempio completo.

P.S. Potete usare sia una closure che una proprietà di funzione. O scrivere entrambe le varianti.

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

Sommare con un numero arbitrario di parentesi

importanza: 2

Scrivete una funzione `sum` che funzioni in questo modo:

```
sum(1)(2) == 3; // 1 + 2
sum(1)(2)(3) == 6; // 1 + 2 + 3
sum(5)(-1)(2) == 6
sum(6)(-1)(-2)(-3) == 0
sum(0)(1)(2)(3)(4)(5) == 15
```

P.S. Aiuto: potresti impostare una conversione “toPrimitive” del tuo oggetto.

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

La sintassi "new Function"

Esiste un ulteriore modo per creare una funzione. È raramente utilizzato, ma a volte è l'unica alternativa.

Sintassi

La sintassi per la creazione di una funzione con questo metodo è la seguente:

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

La funzione viene creata con gli argomenti `arg1...argN` ed il corpo `functionBody`.

E' più semplice da comprendere guardando un esempio. Nel seguente abbiamo una funzione con due argomenti:

```
let sum = new Function('a', 'b', 'return a + b');

alert( sum(1, 2) ); // 3
```

In quest'altro esempio, invece, non abbiamo argomenti, c'è solo il corpo della funzione:

```
let sayHi = new Function('alert("Hello")');

sayHi(); // Hello
```

La differenza dalle altre modalità di creazione delle funzioni che abbiamo visto, è che qui la funzione viene creata letteralmente da una stringa, che viene passata in fase di esecuzione.

Tutte le dichiarazioni di funzione precedenti richiedevano di scrivere il codice della funzione nello script.

Ma `new Function` ci permette di trasformare qualsiasi stringa in una funzione. Per esempio potremmo ricevere una nuova funzione da un server e quindi eseguirla:

```
let str = ... riceviamo il codice della funzione dinamicamente, da un server ...

let func = new Function(str);
func();
```

Viene utilizzato in casi molto specifici, come quando riceviamo del codice da un server, o per compilare dinamicamente una funzione da un modello, in applicazioni web complesse.

Closure (chiusura)

Di solito, una funzione memorizza dove è nata nella proprietà speciale `[[Environment]]`. Questa fa riferimento al Lexical Environment in cui è stata creata (lo abbiamo trattato nel capitolo [Variable scope, closure](#)).

Ma quando una funzione viene creata con `new Function`, il suo `[[Environment]]` non fa riferimento all'attuale Lexical Environment, ma a quello globale.

Quindi, tale funzione non ha accesso alle variabili esterne, ma solo a quelle globali.

```
function getFunc() {
  let value = "test";

  let func = new Function('alert(value)');
  return func;
}
```

```
getFunc(); // error: value is not defined
```

Confrontiamolo con il normale comportamento:

```
function getFunc() {
  let value = "test";

  let func = function() { alert(value); };

  return func;
}

getFunc(); // "test", dal Lexical Environment di getFunc
```

Questa caratteristica speciale di `new Function` sembra strana, ma si rivela molto utile nella pratica.

Immaginiamo di dover creare una funzione da una stringa. Il codice di questa funzione è sconosciuto nel momento in cui scriviamo lo script (per questo non usiamo i normali metodi), ma lo conosceremo durante l'esecuzione. Potremmo riceverlo dal server o da un'altra fonte.

La nostra nuova funzione ha bisogno di interagire con lo script principale.

E se potesse accedere alle variabili esterne?

Il problema è che, prima che JavaScript venga messo in produzione, viene spesso compresso utilizzando un *minifier*, ossia un programma speciale che riduce il codice rimuovendo commenti, spazi e, cosa importante, rinominando le variabili locali utilizzando nomi più brevi.

Ad esempio, se una funzione contiene `let userName`, il minifier lo sostituisce con `let a` (o con un'altra lettera se questa è già occupata), e lo fa ovunque. Solitamente è una procedura sicura: poiché la variabile è locale, nulla al di fuori della funzione può accedervi. Mentre all'interno della funzione il minifier sostituisce ogni sua menzione. I minifiers sono intelligenti, analizzano la struttura del codice e non rompono nulla. Non sono degli stupidi trova-e-sostituisci.

Quindi se `new Function` avesse accesso alle variabili esterne, non sarebbe in grado di trovare la variabile `userName` rinominata.

Se `new Function` avesse accesso alle variabili esterne, ci sarebbero problemi con i minifiers.

Inoltre, tale codice sarebbe pessimo dal punto di vista architettonale e soggetto ad errori.

Per passare qualcosa a una funzione, creata con `new Function`, dovremmo usare i suoi argomenti.

Riepilogo

La sintassi:

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

Per ragioni storiche, gli argomenti possono anche essere passati come elenco separato da virgolette.

Queste tre dichiarazioni hanno lo stesso significato:

```
new Function('a', 'b', 'return a + b'); // sintassi base  
new Function('a,b', 'return a + b'); // elenco separato da virgola  
new Function('a , b', 'return a + b'); // elenco separato da virgola e spazio
```

Nelle funzioni create con `new Function`, `[[Environment]]` fa riferimento al Lexical Environment globale, non a quello esterno. Quindi queste funzioni non possono utilizzare variabili esterne. In realtà ciò è un bene perché ci mette al riparo da errori. Passare i parametri in modo esplicito è un metodo migliore dal punto di vista architettonico e non causa problemi con i minifiers.

Pianificazione: `setTimeout` e `setInterval`

Potremmo decidere di non eseguire subito una funzione, ma dopo un certo lasso di tempo. Questo è detto “pianificare una chiamata” (“scheduling a call”).

Ci sono due metodi per farlo:

- `setTimeout` permette di eseguire una volta la funzione dopo l'intervallo prescelto.
- `setInterval` permette di eseguire regolarmente la funzione lasciando scorrere l'intervallo di tempo prescelto tra una chiamata e l'altra.

Questi metodi non fanno parte delle specifiche di JavaScript. Ma la maggior parte degli ambienti hanno un pianificatore interno e forniscono questi metodi. In particolare, sono supportati in tutti i browser e in Node.js.

`setTimeout`

La sintassi:

```
let timerId = setTimeout(func|codice, [ritardo], [arg1], [arg2], ...)
```

Parametri:

`func|codice`

Funzione o stringa (string) di codice da eseguire. Di solito è una funzione. Per ragioni storiche, si può passare una stringa (string) di codice, ma è sconsigliato.

`ritardo`

Il ritardo in millisecondi (1000 ms = 1 secondo) prima dell'esecuzione, di base 0.

`arg1, arg2 ...`

Gli argomenti della funzione (non supportati in IE9-)

Per esempio, questo codice esegue `saluta()` dopo un secondo:

```
function saluta() {  
  alert('Ciao');  
}  
  
setTimeout(saluta, 1000);
```

Con gli argomenti:

```
function saluta(frase, chi) {  
  alert( frase + ', ' + chi );  
}  
  
setTimeout(saluta, 1000, "Ciao", "Giovanni"); // Ciao, Giovanni
```

Se il primo argomento è una stringa (string), JavaScript crea una funzione dallo stesso.

Quindi funzionerà anche così:

```
setTimeout("alert('Ciao')", 1000);
```

Ma l'utilizzo delle stringhe (string) è sconsigliato, usiamo piuttosto una funzione come questa:

```
setTimeout(() => alert('Ciao'), 1000);
```

Passa una funzione, ma non la esegue

Gli sviluppatori alle prime armi talvolta fanno l'errore di aggiungere le parentesi `()` dopo la funzione:

```
// sbagliato!  
setTimeout(saluta(), 1000);
```

Non funziona, perché `setTimeout` si aspetta un richiamo a una funzione. Qui `saluta()` esegue la funzione e viene passato a `setTimeout` il *risultato della sua esecuzione*. Nel nostro caso il risultato di `saluta()` è `undefined` (la funzione non restituisce nulla), quindi non viene pianificato niente.

Annnullare con `clearTimeout`

Una chiamata a `setTimeout` restituisce un “identificatore del timer” (timer identifier) `timerId` che possiamo usare per disattivare l'esecuzione.

La sintassi per annullare:

```
let timerId = setTimeout(...);  
clearTimeout(timerId);
```

Nel codice qui sotto, pianifichiamo la funzione e poi la annulliamo (abbiamo cambiato idea). Ne risulta che non accade niente:

```
let timerId = setTimeout(() => alert("non accade niente"), 1000);
alert(timerId); // identificatore del timer

clearTimeout(timerId);
alert(timerId); // stesso identificatore (non diventa null dopo la disattivazione)
```

Come possiamo vedere dall'output dell'`alert`, in un browser l'identificatore del timer è un numero. In altri ambienti, potrebbe essere qualcos'altro. Per esempio, Node.js restituisce un oggetto timer con metodi addizionali.

Anche qui, non ci sono specifiche universali per questi metodi, quindi va bene così.

Per i browser, i timer sono descritti nella [sezione Timers ↗](#) di HTML5 standard.

setInterval

Il metodo `setInterval` ha la stessa sintassi di `setTimeout`:

```
let timerId = setInterval(func|codice, [ritardo], [arg1], [arg2], ...)
```

Tutti gli argomenti hanno lo stesso significato. Ma a differenza di `setTimeout` non esegue la funzione solo una volta, ma in modo regolare dopo un dato intervallo di tempo.

Per evitare ulteriori chiamate, dobbiamo eseguire `clearInterval(timerId)`.

L'esempio che segue mostrerà un messaggio ogni 2 secondi. Dopo 5 secondi, l'output verrà fermato:

```
// ripete con un intervallo di 2 secondi
let timerId = setInterval(() => alert('tic'), 2000);

// dopo 5 secondi si ferma
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Il tempo passa mentre viene mostrato l'`alert`

Nella maggior parte dei browser, inclusi Chrome e Firefox, il timer interno continua a "ticchettare" mentre viene mostrato `alert/confirm/prompt`.

Quindi, se eseguiamo il codice qui sopra e non chiudiamo la finestra dell'`alert` per qualche istante, l'`alert` successivo verrà mostrato immediatamente e l'intervallo tra i due avvisi sarà più breve di 2 secondi.

setTimeout annidati

Ci sono due modi per eseguire qualcosa regolarmente.

Uno è `setInterval`. L'altro è un `setTimeout` ricorsivo, come questo:

```
/** invece di:  
let timerId = setInterval(() => alert('tic'), 2000);  
*/  
  
let timerId = setTimeout(function tic() {  
  alert('tic');  
  timerId = setTimeout(tic, 2000); // (*)  
}, 2000);
```

Il `setTimeout` qui sopra pianifica la prossima chiamata subito alla fine di quella attuale `(*)`.

Il `setTimeout` ricorsivo è un metodo più flessibile di `setInterval`. In tal modo la chiamata successiva può essere pianificata in modo diverso, a seconda del risultato di quella attuale.

Per esempio, dobbiamo scrivere un servizio che mandi ogni 5 secondi una richiesta al server chiedendo dati, ma, in caso il server sia sovraccarico, dovrebbe aumentare l'intervallo di 10, 20, 40 secondi...

Qui lo pseudocodice:

```
let ritardo = 5000;  
  
let timerId = setTimeout(function richiesta() {  
  ...manda la richiesta...  
  
  if (la richiesta fallisce a causa di sovraccarico del server) {  
    // aumenta l'intervallo per la prossima esecuzione  
    ritardo *= 2;  
  }  
  
  timerId = setTimeout(richiesta, ritardo);  
}, ritardo);
```

Inoltre, se le funzioni che stiamo pianificando sono avide di CPU, possiamo misurare il tempo richiesto dall'esecuzione e pianificare la chiamata successiva prima o dopo.

Il `setTimeout` ricorsivo permette di impostare un ritardo tra le esecuzioni in modo più preciso di `setInterval`.

Paragoniamo due frammenti di codice. Il primo usa `setInterval`:

```
let i = 1;  
setInterval(function() {  
  func(i++);  
}, 100);
```

Il secondo usa il `setTimeout` ricorsivo:

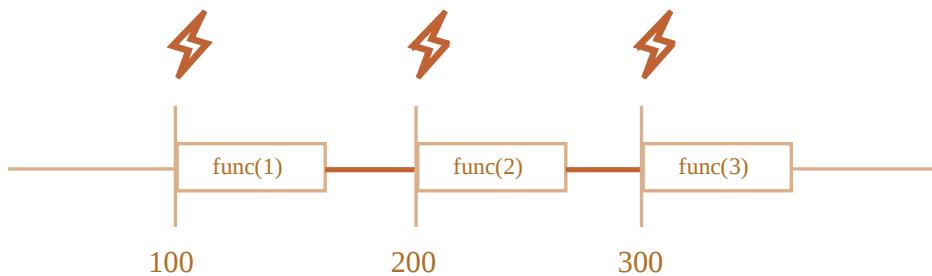
```
let i = 1;  
setTimeout(function avvia() {
```

```

func(i++);
setTimeout(avvia, 100);
}, 100);

```

Per `setInterval` la pianificazione interna eseguirà `func(i++)` ogni 100ms:



Avete notato?

Il ritardo reale tra la chiamata `func` di `setInterval` è inferiore a quello del codice!

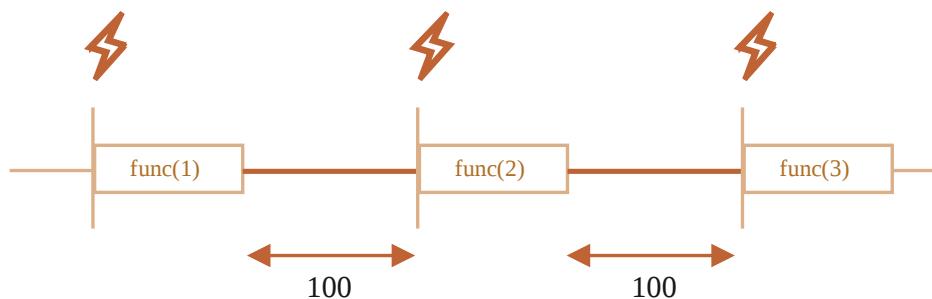
È normale, perché il tempo che occorre all'esecuzione di `func` "consuma" una parte dell'intervallo.

È possibile che l'esecuzione di `func` sia più lunga del previsto e richieda più di 100ms.

In tal caso la macchina attende che `func` sia completa, poi verifica la pianificazione e se il tempo è terminato, la esegue di nuovo *immediatamente*.

In casi limite, se la funzione viene eseguita sempre dopo gli ms di `ritardo`, le chiamate avverranno senza alcuna pausa.

Qui c'è l'immagine per il `setTimeout` ricorsivo:



Il `setTimeout` ricorsivo garantisce il ritardo fissato (qui 100ms).

Questo perché una nuova chiamata è pianificata solo alla fine della precedente.

i La Garbage Collection (letteralmente ‘raccolta dei rifiuti’) e il callback in setInterval/setTimeout

Quando una funzione viene passata in `setInterval/setTimeout`, viene creata e salvata nella pianificazione una referenza interna per la funzione stessa. Questo evita che la funzione venga eliminata anche se non ci sono altre referenze.

```
// la funzione resta in memoria finché la pianificazione la esegue  
setTimeout(function() {...}, 100);
```

Per `setInterval` la funzione resta in memoria fino a quando viene eseguito `clearInterval`.

C’è un effetto collaterale. Una funzione si riferisce all’ambiente lessicale esterno, quindi, finché vive, vivono anche le variabili esterne. Queste possono richiedere molta più memoria della funzione stessa. Ne consegue che quando non ci serve più la funzione pianificata, è meglio cancellarla, anche se è molto piccola.

setTimeout con zero-delay (ritardo zero)

C’è un caso speciale: `setTimeout(func, 0)` o semplicemente `setTimeout(func)`.

In questo caso l’esecuzione della `func` viene pianificata quanto prima possibile, ma la pianificazione la esegue solo dopo che il codice corrente è completo.

Quindi la funzione viene pianificata per avviarsi “subito dopo” il codice corrente.

Per esempio, questo produce “Ciao” quindi, immediatamente, “Mondo”:

```
setTimeout(() => alert("Mondo"));  
  
alert("Ciao");
```

La prima linea “mette in calendario” la chiamata dopo 0ms, ma la pianificazione “verifica il calendario” solo dopo che il codice corrente è completo, quindi “Ciao” viene per primo, seguito da “Mondo”.

Ci sono anche casi di utilizzo avanzato relativi ai browser del timeout zero-delay, li discuteremo nel capitolo [Event loop: microtasks e macrotasks](#).

i Zero-delay in effetti non è zero (in un browser)

In un browser c'è un limite a quanto spesso possono essere avviati i timer nidificati. L'[HTML5 standard](#) dice: "dopo cinque timer nidificati, l'intervallo è costretto a essere di almeno 4 millisecondi".

Vediamo cosa significa con l'esempio qui sotto. La chiamata `setTimeout` riprogramma se stessa con zero-delay. Ogni chiamata ricorda il tempo reale dall'esecuzione precedente nell'array `tempi`. Come sono realmente i ritardi? Vediamo:

```
let partenza = Date.now();
let tempi = [];

setTimeout(function avvia() {
    tempi.push(Date.now() - partenza); // ricorda il ritardo dalla chiamata precedente

    if (partenza + 100 < Date.now()) alert(tempi); // mostra i ritardi dopo 100ms
    else setTimeout(avvia); // allora riprogramma
});

// un esempio del risultato:
// 1,1,1,1,9,15,20,24,30,35,40,45,50,55,59,64,70,75,80,85,90,95,100
```

I primi timer vengono eseguiti immediatamente (come scritto nelle specifiche), poi vediamo 9, 15, 20, 24... Entra in gioco il ritardo obbligatorio di 4+ ms fra le esecuzioni.

Una cosa simile accade se usiamo `setInterval` invece di `setTimeout`:

`setInterval(f)` esegue `f` alcune volte con zero-delay, dopo di che con 4+ ms di ritardo.

Questo limite viene da tempi remoti e molti script vi si affidano, quindi esiste per ragioni storiche.

Per JavaScript lato server, questo limite non esiste e ci sono altri metodi per pianificare un lavoro asincrono immediato, come [setImmediate](#) per Node.js. Quindi questa nota è specifica per i browser.

Riepilogo

- I metodi `setInterval(func, ritardo, ...arg)` e `setTimeout(func, ritardo, ...arg)` consentono di avviare la `func` regolarmente/una volta dopo `ritardo` millisecondi.
- Per disattivare l'esecuzione, dovremo chiamare `clearInterval/clearTimeout` con il valore restituito da `setInterval/setTimeout`.
- La chiamata nidificata di `setTimeout` è un'alternativa più flessibile a `setInterval`, permettendo di impostare in modo più preciso l'intervallo di tempo tra le esecuzioni.
- Zero-delay si pianifica con `setTimeout(func, 0)` (lo stesso di `setTimeout(func)`) ed è usato per pianificare la chiamata "quanto prima possibile, ma dopo che il codice corrente è completo".
- Il browser limita il ritardo minimo per cinque o più chiamate nidificate di `setTimeout` o `setInterval` (dopo la 5a chiamata) a 4ms. Ciò accade per ragioni storiche.

Da notare che tutti i metodi di pianificazione non *garantiscono* un ritardo preciso.

Per esempio, il timer nel browser può rallentare per molte ragioni:

- La CPU è sovraccarica.
- La scheda del browser è sullo sfondo.
- Il portatile ha la batteria scarica.

Tutto ciò può aumentare l'esecuzione minima del timer (il ritardo minimo) di 300ms o anche 1000ms a seconda del browser e le impostazioni di prestazione a livello dell'OS.

✓ Esercizi

Output ogni secondo

importanza: 5

Scrivi una funzione `stampaNumeri(da, a)` che produca un numero ogni secondo, partendo da `da` e finendo con `a`.

Crea due varianti della soluzione.

1. Usando `setInterval`.
2. Usando `setTimeout` ricorsivo.

[Alla soluzione](#)

Cosa mostrerà `setTimeout`?

importanza: 5

Nel codice qui sotto è pianificata una chiamata con `setTimeout`, poi viene eseguito un calcolo pesante, che richiede più di 100ms per essere completato.

Quando verrà eseguita la funzione pianificata?

1. Dopo il loop.
2. Prima del loop.
3. All'inizio del loop.

Cosa mostrerà l'`alert`?

```
let i = 0;

setTimeout(() => alert(i), 100); // ?

// ipotizza che il tempo necessario a eseguire questa funzione sia >100ms
for(let j = 0; j < 100000000; j++) {
    i++;
}
```

[Alla soluzione](#)

Decorators e forwarding, call/apply

JavaScript offre una flessibilità eccezionale quando si tratta di funzioni. Possono essere passate, usate come oggetti, ed ora vedremo come inoltrarle (*forward*) e decorarle (*decorate*).

Caching trasparente

Immaginiamo di avere una funzione `slow(x)` che richiede alla CPU molte risorse, ma i suoi risultati sono stabili. In altre parole, per lo stesso valore di `x` ritorna sempre il medesimo risultato.

Se la funzione viene chiamata spesso, potremmo voler memorizzare nella cache (ricordare) i risultati, per evitare di dedicare tempo extra nel ripetere gli stessi calcoli.

Ma invece di aggiungere quella funzionalità in `slow()`, andremo a creare una funzione *wrapper* (che incapsula o avvolge), che aggiunge il caching. Come vedremo, ci sono molti vantaggi in questo metodo.

Ecco il codice e la sua descrizione:

```
function slow(x) {
  // qui può esserci un duro lavoro per la CPU
  alert(`Called with ${x}`);
  return x;
}

function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) { // se questa chiave è già presente in cache
      return cache.get(x); // leggi il risultato
    }

    let result = func(x); // altrimenti chiama func

    cache.set(x, result); // e metti in cache (memorizza) il risultato
    return result;
  };
}

slow = cachingDecorator(slow);

alert( slow(1) ); // slow(1) viene messo in cache
alert( "Again: " + slow(1) ); // lo stesso

alert( slow(2) ); // slow(2) viene messo in cache
alert( "Again: " + slow(2) ); // lo stesso della linea precedente
```

Nel codice precedente `cachingDecorator` è un *decorator*: una speciale funzione che prende come argomento un'altra funzione e ne altera il comportamento.

L'idea è quella di poter chiamare `cachingDecorator` con qualsiasi funzione per applicare la funzionalità di *caching*. È fantastico, perché in questo modo possiamo avere molte funzioni che utilizzano tale caratteristica, e tutto ciò che dobbiamo fare è applicare ad esse `cachingDecorator`.

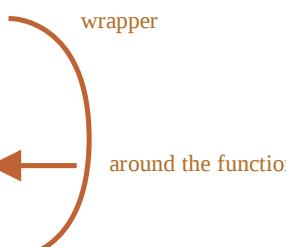
Separando la funzionalità di caching dalla funzione principale abbiamo anche il vantaggio di mantenere il codice semplice.

Il risultato di `cachingDecorator(func)` è un “involucro” (*wrapper*): `function(x)` che “incapsula” (*wraps*) la chiamata di `func(x)` nella logica di caching:

```
function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }

    let result = func(x); ← wrapper
    cache.set(x, result);
    return result;
  };
}
```



Per un codice esterno, la funzione “incapsulata” `slow` continua a fare la stessa cosa. Ma, in aggiunta al suo comportamento, ha ricevuto la funzionalità di caching.

Per riassumere, ci sono diversi vantaggi nell’usare separatamente `cachingDecorator` invece di modificare direttamente il codice di `slow`:

- Il *decorator* `cachingDecorator` è riutilizzabile, possiamo applicarlo ad altre funzioni.
- La logica di cache è separata, non aumenta la complessità della funzione `slow`.
- In caso di bisogno possiamo combinare *decorators* multipli (come vedremo più avanti).

Utilizzo di “`func.call`” per il contesto

Il *decorator* con funzione di caching menzionato prima, non è adatto a lavorare con i metodi degli oggetti.

Ad esempio, nel codice seguente, `worker.slow()` smette di funzionare dopo la *decoration*:

```
// prepariamo worker.slow per essere messo in cache
let worker = {
  someMethod() {
    return 1;
  },

  slow(x) {
    // compito terribilmente impegnativo per la CPU
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};

// stesso codice di prima
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
  };
}
```

```

    }
    let result = func(x); // (**)
    cache.set(x, result);
    return result;
};

}

alert( worker.slow(1) ); // il metodo originale funziona

worker.slow = cachingDecorator(worker.slow); // ora mettiamolo in cache

alert( worker.slow(2) ); // Errore! Error: Cannot read property 'someMethod' of undefined

```

L'errore avviene alla linea (*) la quale cerca di accedere a `this.someMethod`, ma fallisce. Riesci a capire il motivo?

Il motivo è che il *wrapper* chiama la funzione originale come `func(x)` alla linea (**). E, quando chiamata in questo modo, la funzione prende `this = undefined`.

Osserveremmo la stessa cosa se provassimo a eseguire:

```

let func = worker.slow;
func(2);

```

Quindi, il *wrapper* passa la chiamata al metodo originale, ma senza il contesto `this`. Da qui l'errore.

Correggiamolo.

C'è uno speciale metodo di funzione integrato `func.call(context, ...args)` ↗ che permette di chiamare una funzione impostando esplicitamente `this`.

La sintassi è:

```
func.call(context, arg1, arg2, ...)
```

Esegue `func` passando `this` come primo argomento ed i successivi come normali argomenti.

In poche parole, queste due chiamate fanno praticamente la stessa cosa:

```

func(1, 2, 3);
func.call(obj, 1, 2, 3)

```

Entrambe chiamano `func` con gli argomenti 1, 2 e 3. L'unica differenza è che `func.call` imposta anche `this` su `obj`.

Prendiamo il codice sottostante come esempio, chiamiamo `sayHi` usando il contesto di oggetti differenti: `sayHi.call(user)` invoca `sayHi` passando `this=user`, e nella linea seguente imposta `this=admin`:

```

function sayHi() {
  alert(this.name);
}

```

```

}

let user = { name: "John" };
let admin = { name: "Admin" };

// usiamo call per passare oggetti differenti come "this"
sayHi.call( user ); // John
sayHi.call( admin ); // Admin

```

Qui, invece, usiamo `call` per chiamare `say` passando il contesto e l'argomento frase:

```

function say(frase) {
  alert(this.name + ': ' + frase);
}

let user = { name: "John" };

// user diventa this e "Hello" diventa il primo argomento (frase)
say.call( user, "Hello" ); // John: Hello

```

Nel nostro caso possiamo usare `call` nel *wrapper* per passare il contesto alla funzione originale:

```

let worker = {
  someMethod() {
    return 1;
  },

  slow(x) {
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};

function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func.call(this, x); // ora "this" è passato nel modo corretto
    cache.set(x, result);
    return result;
  };
}

worker.slow = cachingDecorator(worker.slow); // ora abilitiamo il caching

alert( worker.slow(2) ); // funziona
alert( worker.slow(2) ); // funziona, non viene chiamato l'originale dalla cache

```

Ora funziona tutto correttamente.

Per renderlo ancora più chiaro, vediamo più approfonditamente come viene passato `this`:

1. Dopo la *decoration* `worker.slow` diventa il *wrapper* `function (x) { ... }`.

- Quindi quando viene eseguito `worker.slow(2)`, il `wrapper` prende `2` come argomento e `this=worker` (è l'oggetto prima del punto).
- All'interno del `wrapper`, assumendo che il risultato non sia stato ancora messo in cache, `func.call(this, x)` passa `this` (`=worker`) e l'argomento (`=2`) al metodo originale.

Passando argomenti multipli

Rendiamo `cachingDecorator` un po' più universale. Finora abbiamo lavorato solamente con funzioni con un solo argomento.

Come fare per gestire il caching del metodo con argomenti multipli `worker.slow`?

```
let worker = {
  slow(min, max) {
    return min + max; // il solito processo terribilmente assetato di CPU
  }
};

// dovrebbe ricordare le chiamate con lo stesso argomento
worker.slow = cachingDecorator(worker.slow);
```

Precedentemente, con un singolo argomento `x` potevamo usare `cache.set(x, result)` per salvare il risultato, e `cache.get(x)` per richiamarlo. Ma ora abbiamo bisogno di memorizzare il risultato per più combinazioni di argomenti `(min, max)`, e il comando `Map` prende un solo argomento come chiave.

Sono possibili diverse soluzioni:

- Implementare una nuova (o usarne una di terze parti) struttura simile a `map`, ma più versatile e che permetta chiavi multiple.
- Usare `maps annidate`: `cache.set(min)` sarà un `Map` che conterrà le coppie `(max, result)`. Quindi potremo avere `result` come `cache.get(min).get(max)`.
- Unire i due valori in uno. Nel nostro caso potremmo usare una semplice stringa `"min, max"` come chiave del `Map`. Per maggiore flessibilità potremmo dotare il decorator di una funzione di `hashing`, che sappia trasformare più valori in uno solo.

Per molte applicazioni pratiche, la terza soluzione è sufficiente, quindi ci atterremo ad essa.

Non abbiamo bisogno di passare solo `x`, ma anche gli altri argomenti in `func.call`. Ricordiamo che in una `function()` sono disponibili tutti i suoi argomenti tramite il `pseudo-array arguments`. Quindi `func.call(this, x)` può essere sostituito con `func.call(this, ...arguments)`.

Il seguente è `cachingDecorator` migliorato:

```
let worker = {
  slow(min, max) {
    alert(`Called with ${min}, ${max}`);
    return min + max;
  }
};
```

```

function cachingDecorator(func, hash) {
  let cache = new Map();
  return function() {
    let key = hash(arguments); // (*)
    if (cache.has(key)) {
      return cache.get(key);
    }
    let result = func.call(this, ...arguments); // (**)
    cache.set(key, result);
    return result;
  };
}

function hash(args) {
  return args[0] + ',' + args[1];
}

worker.slow = cachingDecorator(worker.slow, hash);

alert( worker.slow(3, 5) ); // funziona
alert( "Again " + worker.slow(3, 5) ); // anche qui funziona (dalla cache)

```

Ora funziona con qualsiasi numero di argomenti (anche la funzione hash dovrebbe essere sistemata per consentire un numero qualsiasi di argomenti. Un modo interessante per farlo sarà trattato di seguito).

Ci sono due cambiamenti:

- Nella linea (*) viene chiamato `hash` per creare una chiave unica da `arguments`. In questo caso abbiamo usato una semplice funzione di unione che trasforma gli argomenti (3, 5) nella chiave "3,5". Casi più complessi potrebbero richiedere approcci differenti per la funzione di *hashing*.
- Successivamente (**) usa `func.call(this, ...arguments)` per passare alla funzione originale sia il contesto che tutti gli argomenti ricevuti dal *wrapper*.

func.apply

Anziché `func.call(this, ...arguments)` potremmo usare `func.apply(this, arguments)`.

La sintassi del metodo `func.apply` ↗ è:

```
func.apply(context, args)
```

Questo esegue `func` impostando `this=context` ed usando l'oggetto (simil-array) `args` come lista di argomenti.

L'unica differenza di sintassi tra `call` e `apply` è che `call` si aspetta una lista di argomenti, mentre `apply` vuole un oggetto simil-array.

Queste due chiamate sono praticamente identiche:

```
func.call(context, ...args); // passa un array come lista, usando la sintassi spread  
func.apply(context, args); // è uguale all'uso di call
```

Eseguono la medesima chiamata a `func` con un dati contesto ed argomenti.

C'è solo una sottile differenza:

- La sintassi `...` permette di passare `args iterabili` come lista a `call`.
- `apply` accetta solo *simil-array* `args`.

Quindi, se ci aspettiamo un iterabile usiamo `call`, se invece ci aspettiamo un array, usiamo `apply`.

E per oggetti che sono sia iterabili che simil-array, come un vero array, possiamo usarne uno qualsiasi, ma `apply` sarà probabilmente più veloce, perché è meglio ottimizzato nella maggior parte dei motori JavaScript.

Il passaggio di tutti gli argomenti insieme al contesto a un'altra funzione è chiamato *call forwarding* (inoltro di chiamata).

Questa è la sua forma più semplice:

```
let wrapper = function() {  
  return func.apply(this, arguments);  
};
```

Quando un codice esterno chiama il `wrapper`, è indistinguibile dalla chiamata della funzione originale `func`.

Prendere in prestito un metodo

Ora facciamo un ulteriore piccolo miglioramento nella funzione di hashing:

```
function hash(args) {  
  return args[0] + ',' + args[1];  
}
```

Per ora funziona solo su due argomenti. Sarebbe meglio se potesse unire un numero qualsiasi di `args`.

La soluzione più immediata sarebbe usare il metodo `arr.join ↗`:

```
function hash(args) {  
  return args.join();  
}
```

...Sfortunatamente non funziona, perché stiamo chiamando `hash(arguments)`, e l'oggetto `arguments` è sia iterabile che simil-array, ma non è un vero array.

Quindi chiamare `join` su di esso darebbe errore, come possiamo vedere di seguito:

```
function hash() {
  alert( arguments.join() ); // Error: arguments.join is not a function
}

hash(1, 2);
```

Tuttavia, c'è un modo semplice per utilizzare il metodo `join`:

```
function hash() {
  alert( [].join.call(arguments) ); // 1,2
}

hash(1, 2);
```

Il trucco è chiamato *method borrowing*.

Prendiamo (in prestito) il metodo `join` da un normale array (`[].join`) ed usiamo `[].join.call` per eseguirlo nel contesto di `arguments`.

Perché funziona?

Perché l'algoritmo interno del metodo nativo `arr.join(glue)` è molto semplice.

Preso quasi letteralmente dalla specifica:

1. Imposta `glue` come primo argomento, o, se non ci sono argomenti, una virgola `", "`.
2. Imposta `result` come stringa vuota.
3. Aggiungi `this[0]` a `result`.
4. Aggiungi `glue` e `this[1]`.
5. Aggiungi `glue` e `this[2]`.
6. ...Continua fino a che `this.length` elementi sono stati "incollati".
7. Ritorna `result`.

Quindi, tecnicamente prende `this` ed unisce `this[0]`, `this[1]` ...ecc. E' scritto intenzionalmente in modo da permette l'uso di un simil-array come `this` (non è una coincidenza se molti metodi seguono questa pratica). E' per questo motivo che funziona anche con `this=arguments`.

Decorators e proprietà di funzione

In genere è sicuro sostituire una funzione o un metodo con una sua versione "decorata", tranne per una piccola cosa. Se la funzione originale aveva proprietà associate, come `func.calledCount` o qualsiasi altra cosa, allora quella decorata non le fornirà. Perché quello è un *wrapper*, quindi bisogna stare attenti a come lo si usa.

Es. nell'esempio sopra, se la funzione `slow` avesse avuto delle proprietà, allora `cachingDecorator(slow)` sarebbe stato un *wrapper* senza di esse.

Alcuni *decorators* possono fornire le proprie proprietà. Per esempio, un *decorator* può contare quante volte una funzione è stata invocata e quanto tempo ha impiegato, ed esporre queste informazioni tramite le proprietà del *wrapper*.

Esiste un modo per creare *decorators* che mantengono l'accesso alle proprietà della funzione, ma questo richiede l'uso di uno speciale oggetto `Proxy` per racchiudere una funzione. Ne parleremo più avanti nell'articolo [Proxy e Reflect](#).

Riepilogo

Decorator è un *wrapper* attorno a una funzione che ne altera il comportamento. Il compito principale è ancora svolto dalla funzione.

I *decorators* possono essere visti come “caratteristiche” o “aspetti” che possono essere aggiunti a una funzione. Possiamo aggiungerne uno o aggiungerne molti. E tutto questo senza cambiarne il codice!

Per implementare `cachingDecorator`, abbiamo studiato i metodi:

- `func.call(context, arg1, arg2...)` – chiama `func` con un dato contesto ed argomenti.
- `func.apply(context, args)` – chiama `func` passando `context` come `this` ed un simile array `args` come lista di argomenti.

Generalmente il *call forwarding* viene eseguito usando `apply`:

```
let wrapper = function() {
  return original.apply(this, arguments);
};
```

Abbiamo anche visto un esempio di *method borrowing*, quando prendiamo un metodo da un oggetto ed usiamo `call` per chiamarlo nel contesto di un altro oggetto. È abbastanza comune prendere metodi array e applicarli ad `argomenti`. L'alternativa è utilizzare l'oggetto parametri `...rest` che è un vero array.

Esistono molti usi dei *decorators*, vediamone alcuni risolvendo i tasks di questo capitolo.

✓ Esercizi

decorator spia

importanza: 5

Crea un decorator `spy(func)` che restituisca un wrapper che salva tutte le chiamate alla funzione nella sua proprietà `calls`.

Ogni chiamata viene salvata come un array di argomenti.

Ad esempio:

```
function work(a, b) {
  alert( a + b ); // work è una funzione o un metodo arbitrario
}

work = spy(work);

work(1, 2); // 3
```

```
work(4, 5); // 9

for (let args of work.calls) {
  alert('call:' + args.join()); // "call:1,2", "call:4,5"
}
```

P.S. Questo decoratore a volte è utile per fare *unit-testing*. La sua forma avanzata è `sinon.spy` nella libreria [Sinon.JS](#).

Apri una sandbox con i test. [↗](#)

[Alla soluzione](#)

decorator ritardante

importanza: 5

Crea il decoratore `delay(f, ms)` che ritarda ogni chiamata ad `f` di `ms` millisecondi.

Ad esempio:

```
function f(x) {
  alert(x);
}

// crea i wrappers
let f1000 = delay(f, 1000);
let f1500 = delay(f, 1500);

f1000("test"); // visualizza "test" dopo 1000ms
f1500("test"); // visualizza "test" dopo 1500ms
```

In altre parole, `delay(f, ms)` ritorna una variante di `f` ritardata di `ms`.

Nel codice sopra, `f` è una funzione con un solo argomento, ma la tua soluzione potrebbe passare molti argomenti ed il contesto `this`.

Apri una sandbox con i test. [↗](#)

[Alla soluzione](#)

Debounce decorator

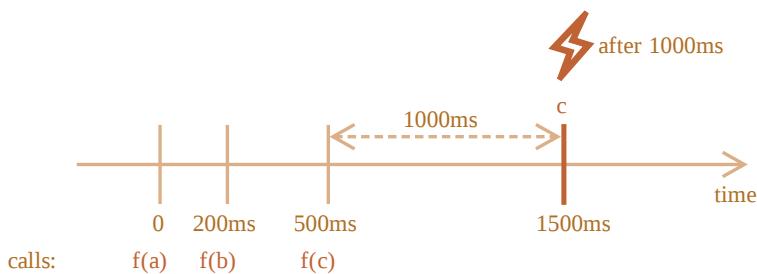
importanza: 5

Il risultato del decoratore `debounce(f, ms)` è un wrapper che sospende le chiamate a `f` finché non ci sono `ms` millisecondi di inattività (nessuna chiamata, “periodo di cooldown”), quindi invoca `f` una volta, con gli ultimi argomenti.

In altre parole, `debounce` è come una segretaria che riceve “telefonate” e aspetta finché non ci sono `ms` di silenzio. Solo allora trasferisce le informazioni sull’ultima chiamata al “capo” (chiama effettivamente `f`).

Ad esempio, avevamo una funzione `f` e l'abbiamo sostituita con `f = debounce(f, 1000)`.

Se la funzione incapsulata viene chiamata a 0 ms, 200 ms e 500 ms, e quindi non ci sono chiamate, la `f` effettiva verrà chiamata solo una volta, a 1500 ms. Cioè dopo il periodo di *cooldown* di 1000 ms dall'ultima chiamata.



... E riceverà gli argomenti dell'ultima chiamata, tutte le altre chiamate vengono ignorate.

Ecco il codice, (usa il *debounce* decorator dalla [Libreria Lodash ↗](#)):

```
let f = _.debounce(alert, 1000);

f("a");
setTimeout( () => f("b"), 200);
setTimeout( () => f("c"), 500);
// la funzione di debounced attende 1000ms dopo l'ultima chiamata, quindi esegue: alert("c")
```

Ora un esempio pratico. Diciamo che l'utente digita qualcosa, e vorremmo inviare una richiesta al server quando l'input è finito.

Non ha senso inviare la richiesta per ogni carattere digitato. Vorremmo invece aspettare, e poi elaborare l'intero risultato.

In un browser web, possiamo impostare un gestore di eventi – una funzione che viene chiamata ad ogni modifica di un campo di input. Normalmente, un gestore di eventi viene chiamato molto spesso, per ogni tasto digitato. Ma se utilizziamo un `debounce` di 1000 ms, verrà chiamato solo una volta, dopo 1000 ms dopo l'ultimo input.

Quindi, `debounce` è un ottimo modo per elaborare una sequenza di eventi: che si tratti di una sequenza di pressioni di tasti, movimenti del mouse o qualsiasi altra cosa.

Aspetta il tempo specificato dopo l'ultima chiamata, quindi esegue la sua funzione, che può elaborare il risultato.

Il compito è implementare il decorator `debounce`.

Suggerimento: sono solo poche righe se ci pensi :)

[Apri una sandbox con i test. ↗](#)

[Alla soluzione](#)

Throttle decorator

importanza: 5

Creare un “throttling” decorator `throttle(f, ms)` – che ritorna un wrapper.

Quando viene chiamato più volte, passa la chiamata a `f` al massimo una volta ogni `ms` millisecondi.

Rispetto al *debounce* decorator abbiamo un decorator completamente diverso:

- `debounce` esegue la funzione una volta, dopo il periodo di “cooldown”. Valido per processare il risultato finale.
- `throttle` la esegue non più spesso dell’intervallo di tempo `ms`. Valido per aggiornamenti regolari ma non troppo frequenti.

In altre parole, `throttle` è come una segretaria che accetta telefonate, ma le passa al capo (chiama `f`) non più di una volta ogni `ms` millisecondi.

Vediamo l’applicazione nella vita reale, per capire meglio tale esigenza e per vedere da dove nasce.

Ad esempio, vogliamo tenere traccia dei movimenti del mouse.

In un browser possiamo impostare una funzione da eseguire ad ogni movimento del mouse, e ottenere la posizione del puntatore mentre si sposta. Durante un utilizzo attivo del mouse, questa funzione di solito viene eseguita molto frequentemente, può essere qualcosa come 100 volte al secondo (ogni 10 ms).

Vorremmo aggiornare alcune informazioni sulla pagina web quando il puntatore si sposta.

... Ma l’aggiornamento della funzione `update()` è troppo pesante per farlo ad ogni micro-movimento. Inoltre, non ha senso aggiornare più spesso di una volta ogni 100 ms.

Quindi la andremo ad inserire nel decorator, usando `throttle(update, 100)` come funzione da eseguire ad ogni movimento del mouse, invece dell’originale `update()`. Il decorator verrà chiamato spesso, ma inoltrerà la chiamata a `update()` al massimo una volta ogni 100 ms.

Visivamente, sarà simile a questo:

1. Per il primo movimento del mouse la variante *decorata* passa immediatamente la chiamata ad `update`. Questo è importante, l’utente vede immediatamente una reazione al suo movimento.
2. Successivamente, per i movimenti del mouse entro lo scadere di `100ms`, non accade nulla. La variante *decorata* ignora le chiamate.
3. Allo scadere dei `100ms` viene chiamato un ulteriore `update` con le ultime coordinate.
4. Infine, il mouse si ferma da qualche parte. La variante *decorata* attende la scadenza dei `100ms` e poi esegue `update` con le ultime coordinate. Quindi, cosa abbastanza importante, vengono elaborate le coordinate finali del mouse.

Un esempio del codice:

```

function f(a) {
  console.log(a);
}

// f1000 passa ad f un massimo di una chiamata ogni 1000 ms
let f1000 = throttle(f, 1000);

f1000(1); // visualizza 1
f1000(2); // (throttling, 1000ms non ancora scaduti)
f1000(3); // (throttling, 1000ms non ancora scaduti)

// allo scadere dei 1000 ms...
// ...visualizza 3, il valore intermedio 2 viene ignorato

```

P.S. Gli argomenti e il contesto `this` passati a `f1000` dovrebbero essere passati alla funzione `f` originale.

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

Function binding

Quando passiamo i metodi di un oggetto tramite callback, ad esempio con `setTimeout`, c'è un problema ben noto: "la perdita di `this`".

In questo capitolo vedremo i modi per risolverlo.

“`this` perso”

Abbiamo già visto esempi di `this` perso. Quando un metodo viene passato separatamente dall'oggetto che lo contiene – `this` viene perso.

Ecco quello che accade con `setTimeout`:

```

let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(user.sayHi, 1000); // Hello, undefined!

```

Come possiamo vedere, l'output non mostra “John” come `this.firstName`, ma `undefined`!

Questo perché `setTimeout` ha ricevuto `user.sayHi` separatamente dall'oggetto. L'ultima riga può essere riscritta così:

```
let f = user.sayHi;
```

```
setTimeout(f, 1000); // il contesto user è stato perso
```

Nei browser, il metodo `setTimeout` è un po' speciale: per la chiamata della funzione imposta `this=window` (in Node.js, invece, `this` è l'oggetto timer, ma qui non ci interessa). Quindi per `this.firstName` prova a recuperare `window.firstName`, il quale non esiste. In altri casi simili, di solito `this` diventa semplicemente `undefined`.

Il problema è abbastanza tipico: vogliamo passare un metodo di un oggetto da qualche parte (qui, allo scheduler), dove verrà chiamato. Come assicurarsi che venga chiamato nel giusto contesto?

Soluzione 1: un wrapper

La soluzione più semplice consiste nell'usare una funzione wrapper:

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(function() {
  user.sayHi(); // Hello, John!
}, 1000);
```

Ora funziona, perché riceve `user` dal *lexical environment* esterno, e quindi chiama il metodo normalmente.

Lo stesso, ma più conciso:

```
setTimeout(() => user.sayHi(), 1000); // Hello, John!
```

Sembra ok, ma nella struttura del nostro codice appare una piccola vulnerabilità.

Cosa succederebbe se prima che `setTimeout` "scada" (c'è un secondo di ritardo!) `user` cambiasse valore? All'improvviso ci ritroveremmo con l'oggetto sbagliato!

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(() => user.sayHi(), 1000);

// ...il valore di user cambia entro 1 secondo
user = {
  sayHi() { alert("Another user in setTimeout!"); }
};
```

```
// Un altro user in setTimeout!
```

La prossima soluzione garantisce che questo genere di cose non accada.

Soluzione 2: bind

Function fornisce il metodo nativo [bind ↗](#) che permette di “fissare” `this`.

La sintassi di base è la seguente:

```
// più avanti vedremo una sintassi più complessa
let boundFunc = func.bind(context);
```

Il risultato di `func.bind(context)` è un “exotic object”, una particolare funzione richiamabile come una normale funzione e che passa in maniera trasparente la chiamata a `func` impostando `this=context`.

In altre parole, chiamare `boundFunc` è come chiamare `func` con `this` fisso.

Ad esempio, qui `funcUser` passa la chiamata a `func` con `this=user`:

```
let user = {
  firstName: "John"
};

function func() {
  alert(this.firstName);
}

let funcUser = func.bind(user);
funcUser(); // John
```

Qui `func.bind(user)` è come una “variante” di `func`, con `this=user` fisso.

Tutti gli argomenti vengono passati così come sono a `func` originale, ad esempio:

```
let user = {
  firstName: "John"
};

function func(phrase) {
  alert(phrase + ', ' + this.firstName);
}

// bind this a user
let funcUser = func.bind(user);

funcUser("Hello"); // Hello, John (l'argomento "Hello" viene passato, e this=user)
```

Ora proviamo con il metodo di un oggetto:

```

let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

let sayHi = user.sayHi.bind(user); // (*)

// puo' funzionare senza un oggetto
sayHi(); // Hello, John!

setTimeout(sayHi, 1000); // Hello, John!

// anche se il valore di user cambia entro 1 secondo
// sayHi utilizza il valore pre-associato che fa riferimento al vecchio oggetto user
user = {
  sayHi() { alert("Un utente differente in setTimeout!"); }
};

```

Alla linea (*) prendiamo il metodo `user.sayHi` e lo leghiamo a `user.sayHi` è una funzione “associata”, che può essere chiamata da sola, o passata a `setTimeout` – non importa, il contesto sarà sempre esatto.

Qui possiamo vedere che gli argomenti vengono passati “così come sono”, solo `this` viene fissato da `bind`:

```

let user = {
  firstName: "John",
  say(phrase) {
    alert(`${phrase}, ${this.firstName}`);
  }
};

let say = user.say.bind(user);

say("Hello"); // Hello, John (l'argomento "Hello" viene passato a say)
say("Bye"); // Bye, John ("Bye" viene passato a say)

```

i Un metodo comodo: bindAll

Se un oggetto possiede molti metodi che abbiamo bisogno di passare, allora potremmo eseguire un ciclo per usare `bind` su tutti:

```

for (let key in user) {
  if (typeof user[key] == 'function') {
    user[key] = user[key].bind(user);
  }
}

```

Alcune librerie JavaScript forniscono utili funzioni per il *binding* di massa, ad esempio `_.bindAll(object, methodNames)` ↗ in lodash.

Funzioni parziali

Finora abbiamo solo parlato di come legare `this`. Portiamo il concetto ad un livello successivo.

Possiamo legare non solo `this`, ma anche argomenti. Questo viene fatto raramente, ma a volte può rivelarsi utile.

La sintassi completa di `bind`:

```
let bound = func.bind(context, [arg1], [arg2], ...);
```

Questo permette di legare alla funzione un contesto come `this`, e degli argomenti di partenza.

Ad esempio, abbiamo la funzione di moltiplicazione `mul(a, b)`:

```
function mul(a, b) {
  return a * b;
}
```

Usiamo `bind` per creare una funzione `double` che si basi su di essa:

```
function mul(a, b) {
  return a * b;
}

let double = mul.bind(null, 2);

alert( double(3) ); // = mul(2, 3) = 6
alert( double(4) ); // = mul(2, 4) = 8
alert( double(5) ); // = mul(2, 5) = 10
```

La chiamata a `mul.bind(null, 2)` crea una nuova funzione `double` che passa la chiamata a `mul`, fissa `null` come contesto e `2` come primo argomento. I seguenti argomenti vengono passati “così come sono”.

Questa pratica è chiamata “applicazione parziale di funzione” [partial function application ↗](#) – creiamo una nuova funzione fissando alcuni parametri di quella esistente.

Nota che in realtà qui non usiamo `this`, ma `bind` lo richiede, quindi dobbiamo mettere al suo posto qualcosa tipo `null`.

La funzione `triple` nel codice che segue, triplica il valore:

```
function mul(a, b) {
  return a * b;
}

let triple = mul.bind(null, 3);

alert( triple(3) ); // = mul(3, 3) = 9
alert( triple(4) ); // = mul(3, 4) = 12
alert( triple(5) ); // = mul(3, 5) = 15
```

Perché dovremmo aver bisogno di una funzione parziale?

Il vantaggio è che possiamo creare una funzione indipendente con un nome leggibile (`double`, `triple`). Possiamo usarla senza fornire tutte le volte il primo argomento, poiché è stato associato con `bind`.

In altri casi, l'applicazione parziale è utile quando abbiamo una funzione molto generica e per comodità ne vogliamo una variante meno universale.

Ad esempio, abbiamo una funzione `send(from, to, text)`. Ma nell'oggetto `user` potremmo volerne usare una parziale variante: `sendTo(to, text)` che invia dallo user corrente.

Utilizzo parziale senza contesto

E se volessimo fissare alcuni argomenti, ma non il contesto `this`? Ad esempio, per un metodo in un oggetto.

Il `bind` nativo non lo permette. Non possiamo semplicemente omettere il contesto e saltare agli argomenti.

Fortunatamente una funzione `partial` per legare solo gli argomenti, può essere implementata con facilità.

Così:

```
function partial(func, ...argsBound) {
  return function(...args) { // (*)
    return func.call(this, ...argsBound, ...args);
  }
}

// Uso:
let user = {
  firstName: "John",
  say(time, phrase) {
    alert(`[${time}] ${this.firstName}: ${phrase}!`);
  }
};

// aggiunge un metodo parziale con un orario fisso
user.sayNow = partial(user.say, new Date().getHours() + ':' + new Date().getMinutes());

user.sayNow("Hello");
// Qualcosa tipo:
// [10:00] John: Hello!
```

Il risultato della chiamata di `partial(func[, arg1, arg2...])` è un wrapper (*) che chiama `func` con:

- Lo stesso `this` (per la chiamata di `user.sayNow` è `user`)
- quindi passa `...argsBound` – argomenti dalla chiamata di `partial ("10:00")`
- quindi passa `...args` – argomenti passati al wrapper (`"Hello"`)

E' molto facile da fare con la sintassi spread vero?

Esiste anche un'implementazione già pronta, [_.partial](#) ↗ dalla libreria lodash.

Riepilogo

Il metodo `func.bind(context, ...args)` ritorna una “variante associata” della funzione `func` con il contesto `this` fisso, ed i primi argomenti, se impostati.

Di solito usiamo `bind` per fissare `this` in un metodo di un oggetto, in modo da poterlo passare senza problemi. Ad esempio a `setTimeout`.

Quando leghiamo alcuni argomenti ad una funzione esistente, la funzione risultante (meno universale) è chiamata *parzialmente applicata* o *parziale*.

Le parziali sono utili quando non vogliamo ripetere lo stesso argomento più e più volte. Ad esempio, se abbiamo una funzione `send (from, to)`, e `from` dovrebbe essere sempre lo stesso.

✓ Esercizi

Funzione associata come metodo

importanza: 5

Quale sarà l'output?

```
function f() {
  alert( this ); // ?
}

let user = {
  g: f.bind(null)
};

user.g();
```

[Alla soluzione](#)

Secondo bind

importanza: 5

Possiamo cambiare `this` con una associazione addizionale?

Quale sarà l'output?

```
function f() {
  alert(this.name);
}

f = f.bind( {name: "John"} ).bind( {name: "Ann"} );

f();
```

[Alla soluzione](#)

Proprietà della funzione dopo il bind

importanza: 5

C'è un valore nella proprietà di una funzione. Cambierà dopo `bind`? Perché, o perché no?

```
function sayHi() {
  alert( this.name );
}

sayHi.test = 5;

let bound = sayHi.bind({
  name: "John"
});

alert( bound.test ); // quale sarà l'output? Perché?
```

[Alla soluzione](#)

Correggi una funzione che ha perso "this"

importanza: 5

La chiamata di `askPassword()` nel codice sottostante dovrebbe controllare la password e quindi chiamare `user.loginOk/loginFail` a seconda della risposta.

Ma porta a un errore. Perché?

Correggi la riga evidenziata affinché tutto funzioni correttamente (le altre righe non devono essere modificate).

```
function askPassword(ok, fail) {
  let password = prompt("Password?", '');
  if (password == "rockstar") ok();
  else fail();
}

let user = {
  name: 'John',

  loginOk() {
    alert(`#${this.name} logged in`);
  },
  loginFail() {
    alert(`#${this.name} failed to log in`);
  },
};

askPassword(user.loginOk, user.loginFail);
```

[Alla soluzione](#)

Applicazione parziale per login

importanza: 5

Il compito è una variante leggermente più complessa di [Correggi una funzione che ha perso "this"](#).

L'oggetto `user` è stato modificato. Ora al posto delle due funzioni `loginOk/loginFail`, ha una sola funzione `user.login(true/false)`.

Cosa dovremmo passare a `askPassword` nel codice qui sotto, in modo che chiami `user.login(true)` come `ok` e `user.login(false)` come `fail`?

```
function askPassword(ok, fail) {
  let password = prompt("Password?", '');
  if (password == "rockstar") ok();
  else fail();
}

let user = {
  name: 'John',

  login(result) {
    alert( this.name + (result ? ' logged in' : ' failed to log in') );
  }
};

askPassword(?); // ?
```

Le tue modifiche dovrebbero solo interessare la porzione di codice evidenziata.

[Alla soluzione](#)

Arrow functions rivisitate

Rivediamo l'argomento delle arrow functions.

Le arrow functions non sono semplicemente una “scorciatoia” per scrivere codice più breve. Infatti queste possiedono delle caratteristiche aggiuntive piuttosto utili.

Usando JavaScript ci troviamo spesso a dover scrivere delle funzioni molto semplici, che verranno però eseguite in un diverso punto del codice.

Ad esemmpio:

- `arr.forEach(func)` – `func` viene eseguita da `forEach` per ogni elemento dell'array.
- `setTimeout(func)` – `func` viene eseguita dallo scheduler integrato.
- ...ed esistono molti altri casi.

È nel vero spirito di JavaScript poter creare una funzione in un punto e passarla in qualsiasi altra parte del codice.

E in questo tipo di funzioni, generalmente, non vorremmo perdere il riferimento al context (il contesto in cui la funzione è stata definita). Queste sono le situazioni in cui le arrow functions ci vengono in soccorso.

Le arrow functions non possiedono un “this”

Come già abbiamo studiato nel capitolo [Metodi degli oggetti, "this"](#), le arrow functions non possiedono un `this`. Infatti, il valore di `this`, viene preso dal contesto esterno.

Ad esempio, possiamo utilizzarlo per le iterazioni all'interno del un metodo di un oggetto:

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],

  showList() {
    this.students.forEach(
      student => alert(this.title + ': ' + student)
    );
  }
};

group.showList();
```

Qui, all'interno del `forEach`, viene utilizzata una arrow function, quindi il valore di `this.title` è esattamente lo stesso che troviamo nel metodo esterno `showList`. Cioè: `group.title`.

Se avessimo utilizzato una funzione “regolare”, avremmo ottenuto un errore:

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],

  showList() {
    this.students.forEach(function(student) {
      // Error: Cannot read property 'title' of undefined
      alert(this.title + ': ' + student);
    });
  }
};

group.showList();
```

L'errore viene generato perché `forEach` esegue le funzioni con `this=undefined` di default, quindi il codice equivale a: `undefined.title`.

Questo non si verifica con le arrow functions, poiché queste non possiedono un `this`.

Arrow functions non funzionano con `new`

Il fatto di non possedere un `this` porta ad una naturale limitazione: le arrow functions non possono essere utilizzate come costruttori. Non possono essere invocate con la keyword `new`.

Arrow functions VS bind

Esiste una sottile differenza tra l'utilizzo di una arrow function `=>` ed una funzione regolare invocata con `.bind(this)`:

- `.bind(this)` crea una versione “bounded” (delimitata) della funzione.
- Le arrow functions, tramite `=>`, non creano alcun binding. La funzione semplicemente non avrà un `this`. La ricerca di `this` seguirà esattamente le stesse procedure della ricerca di una variabile: verrà cercata nel lexical environment esterno.

Le arrow functions non possiedono “arguments”

Le arrow functions non possiedono la variabile `arguments`.

Questo è fantastico per i decorators, in cui abbiamo bisogno di inoltrare una chiamata con il valori attuali di `this` e `arguments`.

Ad esempio, `defer(f, ms)` accetta una funzione come parametro e ritorna un wrapper della stessa, il quale ne ritarderà l'invocazione di `ms` millisecondi:

```
function defer(f, ms) {
  return function() {
    setTimeout(() => f.apply(this, arguments), ms);
  };
}

function sayHi(who) {
  alert('Hello, ' + who);
}

let sayHiDeferred = defer(sayHi, 2000);
sayHiDeferred("John"); // Hello, John dopo 2 secondi
```

La stessa cosa, senza una arrow function, sarebbe:

```
function defer(f, ms) {
  return function(...args) {
    let ctx = this;
    setTimeout(function() {
      return f.apply(ctx, args);
    }, ms);
  };
}
```

In questo caso abbiamo avuto bisogno di creare delle variabili aggiuntive come `args` e `ctx`, in modo che la funzione interna a `setTimeout` possa riceverle.

Riepilogo

Arrow functions:

- Non possiedono `this`
- Non possiedono `arguments`
- Non possono essere invocate con `new`
- Non possiedono `super`, ma non lo abbiamo ancora studiato. Lo faremo nel capitolo [Ereditarietà delle classi](#)

Questo perché sono pensate per piccole parti di codice che non possiedono un contesto proprio, ma piuttosto, lavorano nel contesto corrente. E sono veramente perfette per questi casi d'uso.

Configurazione delle proprietà dell'oggetto

Procederemo con lo studio degli oggetti e studieremo le loro proprietà più nel dettaglio.

Attributi e descrittori di proprietà

Come già sappiamo, gli oggetti possono memorizzare proprietà.

Fino ad ora, per noi, una proprietà è sempre stata una coppia “chiave-valore”. Ma in realtà, una proprietà è molto più potente e flessibile di così.

In questo capitolo studieremo ulteriori opzioni di configurazione, e nel prossimo vedremo come trasformarle in funzioni getter/setter.

Attributi di proprietà

Le proprietà degli oggetti, oltre ad un `valore`, possiedono tre attributi speciali (così detti “flags”, o “bandiere”):

- `writable` – se impostato a `true`, il valore può essere modificato, altrimenti è possibile accedervi in sola lettura.
- `enumerable` – se impostato a `true`, appare nei loop, altrimenti non verrà considerata.
- `configurable` – se impostato a `true`, la proprietà può essere cancellata e questi attributi possono essere modificati.

Non li abbiamo mai visti fino ad ora, perché generalmente non vengono mostrati. Quando creiamo una proprietà in “modo ordinario”, questi attributi vengono tutti impostati a `true`. Ma possiamo comunque modificarli in qualsiasi momento.

Come prima cosa, vediamo come poter accedere a questi attributi.

Il metodo [Object.getOwnPropertyDescriptor](#)  ritorna *tutte* le informazioni riguardo una proprietà.

La sintassi:

```
let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);
```

obj

L'oggetto da cui vogliamo ottenere le informazioni.

propertyName

Il nome della proprietà.

Il valore ritornato viene chiamato “descrittore di proprietà” dell’oggetto: contiene il valore della proprietà e tutti i suoi attributi.

Ad esempio:

```
let user = {  
    name: "John"  
};  
  
let descriptor = Object.getOwnPropertyDescriptor(user, 'name');  
  
alert( JSON.stringify(descriptor, null, 2 ) );  
/* descrittore di proprietà:  
{  
    "value": "John",  
    "writable": true,  
    "enumerable": true,  
    "configurable": true  
}  
*/
```

Per modificare gli attributi possiamo utilizzare [Object.defineProperty](#).

La sintassi:

```
Object.defineProperty(obj, propertyName, descriptor)
```

obj , propertyName

L’oggetto e la proprietà a cui applicare il descrittore.

descriptor

Oggetto *descriptor* da utilizzare.

Se la proprietà esiste, `defineProperty` aggiornerà l’attributo. Altrimenti, creerà la proprietà con il valore e gli attributi forniti; se un attributo non viene fornito, gli verrà assegnato il valore `false`.

Ad esempio, qui creiamo una proprietà `name` con tutti gli attributi `false`:

```
let user = {};
```

```

Object.defineProperty(user, "name", {
  value: "John"
});

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2) );
/*
{
  "value": "John",
  "writable": false,
  "enumerable": false,
  "configurable": false
}
*/

```

Confrontandola con la proprietà “creata normalmente” `user.name` vista sopra, ora tutti gli attributi sono `false`. Se questo non è ciò che vogliamo, allora dovremmo impostarli a `true` tramite il `descriptor`.

Ora analizziamo gli effetti degli attributi guardando alcuni esempi.

Non-writable

Vediamo come rendere `user.name` *non-writable* (la variabile non può essere riassegnata) modificando l’attributo `writable`:

```

let user = {
  name: "John"
};

Object.defineProperty(user, "name", {
  writable: false
});

user.name = "Pete"; // Error: Cannot assign to read only property 'name'

```

Ora nessuno potrà modificare il nome dell’utente, a meno che non vada a sovrascrivere il valore degli attributi con `defineProperty`.

Gli errori verranno mostrati solamente in strict mode

Se non siamo in “strict mode”, e tentiamo di sovrascrivere una proprietà non-writable, non verrà mostrato alcun errore. Nonostante non venga mostrato l’errore, l’operazione fallirà comunque. Quindi le violazioni di attributi fuori dalla strict mode verranno silenziosamente ignorate.

Qui vediamo lo stesso esempio, ma la proprietà viene creata dal nulla:

```

let user = { };

Object.defineProperty(user, "name", {

```

```

    value: "John",
    // per le nuove proprietà dobbiamo esplicitare quali attributi sono true
    enumerable: true,
    configurable: true
});

alert(user.name); // John
user.name = "Pete"; // Error

```

Non-enumerable

Ora proviamo ad aggiungere un metodo `toString` ad `user`.

Normalmente, la funzione *built-in* (integrata) `toString`, per gli oggetti è non-enumerabile, quindi non verrà mostrata nei cicli come `for .. in`. Ma se proviamo ad aggiungere una nostra definizione di `toString`, allora questa verrà mostrata nei cicli `for .. in`, come nell'esempio:

```

let user = {
  name: "John",
  toString() {
    return this.name;
  }
};

// Di default, entrambe le proprietà verranno elencate
for (let key in user) alert(key); // name, toString

```

Se non è ciò che ci aspettiamo, possiamo impostare l'attributo `enumerable:false`. In questo modo non verrà più mostrata nei cicli `for .. in`, proprio come la funzione già integrata (definita da Javascript):

```

let user = {
  name: "John",
  toString() {
    return this.name;
  }
};

Object.defineProperty(user, "toString", {
  enumerable: false
});

// In questo modo la nostra funzione toString sparirà
for (let key in user) alert(key); // name

```

Non-enumerable properties are also excluded from `Object.keys`:

```
alert(Object.keys(user)); // name
```

Non-configurable

L'attributo non-configurable (`configurable: false`) è talvolta preimpostato negli oggetti e nelle proprietà integrate.

Una proprietà *non-configurable* non può essere cancellata.

Ad esempio, `Math.PI` è *non-writable*, *non-enumerable* e *non-configurable*:

```
let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');

alert( JSON.stringify(descriptor, null, 2) );
/*
{
  "value": 3.141592653589793,
  "writable": false,
  "enumerable": false,
  "configurable": false
}
*/
```

Quindi, uno sviluppatore non sarà in grado di cambiare il valore `Math.PI` o di sovrascriverlo.

```
Math.PI = 3; // Error, because it has writable: false

// e nemmeno la cancellazione di Math.PI funzionerebbe
```

Rendere una proprietà *non-configurable* è una “strada a senso unico”. Non possiamo tornare indietro tramite `defineProperty`.

Per essere precisi, l'attributo non-configurable impone diverse restrizioni a `defineProperty`:

1. Non possiamo modificare l'attributo `configurable`.
2. Non possiamo modificare l'attributo `enumerable`.
3. Non possiamo modificare l'attributo da `writable: false` a `true` (possiamo invece modificarlo da `true` a `false`).
4. Non possiamo modificare le funzioni di accesso `get/set` (ma possiamo assegnarle nel caso non siano definite).

L'idea alla base di “configurable: false” è quella di prevenire la modifica e la rimozione degli attributi di una proprietà, permettendo comunque la modifica del suo valore.

In questo esempio `user.name` è *non-configurable*, ma possiamo comunque modificarlo (poiché è *writable*):

```
let user = {
  name: "John"
};

Object.defineProperty(user, "name", {
  configurable: false
});

user.name = "Pete"; // Funziona senza errori
delete user.name; // Error
```

Qui invece “sigilliamo” per sempre `user.name` rendendolo un valore costante:

```
let user = {
  name: "John"
};

Object.defineProperty(user, "name", {
  writable: false,
  configurable: false
});

// non saremo in grado di modificare user.name o i suoi attributi
// nessuna delle seguenti istruzioni funzionerà
user.name = "Pete";
delete user.name;
Object.defineProperty(user, "name", { value: "Pete" });
```

i The only attribute change possible: writable true → false

There's a minor exception about changing flags.

We can change `writable: true` to `false` for a non-configurable property, thus preventing its value modification (to add another layer of protection). Not the other way around though.

Object.defineProperties

Utilizzando il metodo `Object.defineProperties(obj, descriptors)` ↗ abbiamo la possibilità di definire più proprietà alla volta.

La sintassi è:

```
Object.defineProperties(obj, {
  prop1: descriptor1,
  prop2: descriptor2
  // ...
});
```

Ad esempio:

```
Object.defineProperties(user, {
  name: { value: "John", writable: false },
  surname: { value: "Smith", writable: false },
  // ...
});
```

In questo modo siamo in grado di impostare più proprietà in una volta sola.

Object.getOwnPropertyDescriptors

Per ottenere tutti i descrittori di una proprietà, possiamo utilizzare il metodo [Object.getOwnPropertyDescriptors\(obj\)](#).

Il metodo `Object.defineProperties` può essere utilizzato per clonare un oggetto mantenendo gli attributi delle sue proprietà:

```
let clone = Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));
```

Normalmente, quando cloniamo un oggetto, utilizziamo l'assegnazione per copiarne le proprietà, come nell'esempio:

```
for (let key in user) {
  clone[key] = user[key]
}
```

...Ma in questo modo non stiamo copiando gli attributi. Quindi per una clonazione più completa, l'utilizzo di `Object.defineProperties` è la scelta migliore.

Un'altra differenza è che `for .. in` ignora le proprietà di tipo `symbol`, mentre `Object.getOwnPropertyDescriptors` ritorna *tutti* i descrittori, inclusi quelli di tipo `symbol`.

Sigillare un oggetto globalmente

I descrittori di proprietà permettono di lavorare a livello di proprietà.

Esistono però diversi metodi in grado di limitare l'accesso *all'intero* oggetto:

[Object.preventExtensions\(obj\)](#)

Vieta di aggiungere nuove proprietà all'oggetto.

[Object.seal\(obj\)](#)

Vieta di aggiungere/rimuovere proprietà, ed imposta `configurable: false` su tutte le proprietà già esistenti dell'oggetto.

[Object.freeze\(obj\)](#)

Vieta di aggiungere/rimuovere/modificare le proprietà dell'oggetto. Imposta `configurable: false, writable: false` su tutte le proprietà già esistenti dell'oggetto.

Ed esistono anche dei metodi per verificare lo stato degli attributi di un oggetto:

[Object.isExtensible\(obj\)](#)

Ritorna `false` se è vietato aggiungere nuove proprietà, altrimenti ritorna `true`.

[Object.isSealed\(obj\)](#)

Ritorna `true` se è vietato aggiungere/rimuovere proprietà, e tutte le altre proprietà sono impostate a `configurable: false`.

[Object.isFrozen\(obj\)](#)

Ritorna `true` se è vietato aggiungere/rimuovere/modificare proprietà, e tutte le altre proprietà sono impostate a `configurable: false, writable: false`.

In pratica, tuttavia, questi metodi sono utilizzati molto raramente.

Proprietà getters e setters

Esistono due tipi di proprietà per gli oggetti.

Il primo tipo sono le *data properties* (proprietà di tipo “dato”). Sappiamo già come utilizzarle, poiché tutte le proprietà viste fino ad ora erano *data properties*.

Il secondo tipo di proprietà è qualcosa di nuovo. Sono definite *accessor properties* (proprietà accessorie). Sono essenzialmente funzioni che vengono eseguite quando viene letto o impostato un valore, ma al codice esterno appaiono come normali proprietà.

Getters e setters

Le *accessor properties* sono rappresentate dai metodi “*getter*” e “*setter*”. In un *object literal* vengono indicate da `get` e `set`:

```
let obj = {
  get propName() {
    // getter, il codice eseguito per ottenere obj.propName
  },
  set propName(value) {
    // setter, il codice eseguito per impostare il valore di obj.propName = value
  }
};
```

La proprietà *getter* viene eseguita quando `obj.propName` viene letto, la proprietà *setter*, invece, quando viene assegnato.

Ad esempio, abbiamo un oggetto `user` con le proprietà `name` e `surname`:

```
let user = {
  name: "John",
  surname: "Smith"
};
```

Ora vogliamo aggiungere una proprietà `fullName`, che dovrebbe valere `"John Smith"`. Ovviamente vorremmo evitare di copiare ed incollare informazioni già esistenti, quindi possiamo implementare questa funzionalità tramite un *accessor*:

```
let user = {
  name: "John",
  surname: "Smith",
```

```

    get fullName() {
        return `${this.name} ${this.surname}`;
    }
};

alert(user.fullName); // John Smith

```

Vista esternamente, una accessor *property* è del tutto simile ad una normale proprietà, è questa l'idea che sta dietro alle accessor *properties*. Non *invochiamo* `user.fullName` come una normale funzione, ma la *leggiamo* come una normale proprietà: in questo caso il *getter* sta lavorando per noi.

Per ora, `fullName` possiede un solo getter. Se provassimo ad assegnare `user.fullName=`, otterremo un errore:

```

let user = {
    get fullName() {
        return '...';
    }
};

user.fullName = "Test"; // Error (la proprietà possiede solo un getter)

```

Aggiungiamo quindi un *setter* per `user.fullName`:

```

let user = {
    name: "John",
    surname: "Smith",

    get fullName() {
        return `${this.name} ${this.surname}`;
    },

    set fullName(value) {
        [this.name, this.surname] = value.split(" ");
    }
};

// set fullName viene eseguito con i valori forniti
user.fullName = "Alice Cooper";

alert(user.name); // Alice
alert(user.surname); // Cooper

```

Come risultato finale, abbiamo un proprietà “virtuale” `fullName`. Che possiamo sia leggere che scrivere.

Descrittori degli accessors

I descrittori per le accessor *properties* sono diversi da quelli per le *data properties*.

Per le accessor *properties*, non ci sono `value` o `writable`, ma ci sono invece le funzioni `get` or `set`.

Un descrittore di accessor *properties* può possedere:

- `get` – una funzione che non accetta argomenti, che specifica come accedere in lettura ad una proprietà,
- `set` – una funzione con un solo argomento, che specifica come impostare il valore della proprietà,
- `enumerable` – stesso comportamento visto per le *data properties*,
- `configurable` – stesso comportamento visto per le *data properties*.

Ad esempio, possiamo creare un accessor `fullName` con `defineProperty`, passando un `descriptor` con `get` e `set`:

```
let user = {
  name: "John",
  surname: "Smith"
};

Object.defineProperty(user, 'fullName', {
  get() {
    return `${this.name} ${this.surname}`;
  },
  set(value) {
    [this.name, this.surname] = value.split(" ");
  }
});

alert(user.fullName); // John Smith

for(let key in user) alert(key); // name, surname
```

Da notare che una proprietà può essere o un accessor (con i metodi `get/set`) o una *data property* (con un `value`), ma non entrambe.

Se proviamo a fornire sia `get` che `value`, nello stesso `descriptor`, otterremo un errore:

```
// Error: Invalid property descriptor.
Object.defineProperty({}, 'prop', {
  get() {
    return 1
  },
  value: 2
});
```

Getters/setters intelligenti

Getters/setters possono essere utilizzati come *wrappers* (contenitori) per le proprietà “reali”, in questo modo avremo più controllo sulle operazioni di lettura/scrittura.

Ad esempio, potremmo vietare nomi troppo brevi per la proprietà `name`, possiamo definire un setter `name` e mantenere il valore in una proprietà diversa `_name`:

```

let user = {
  get name() {
    return this._name;
  },

  set name(value) {
    if (value.length < 4) {
      alert("Name is too short, need at least 4 characters");
      return;
    }
    this._name = value;
  }
};

user.name = "Pete";
alert(user.name); // Pete

user.name = ""; // Il nome è troppo corto...

```

Quindi, il nome viene memorizzato nella proprietà `_name`, e gli accessi vengono effettuati tramite *getter* e *setter*.

Tecnicamente, il codice all'esterno potrebbe accedere direttamente al nome utilizzando `user._name`. Ma esiste una concezione molto diffusa che specifica di non utilizzare direttamente le proprietà che iniziano con `"_"`.

Utilizzato per compatibilità

Uno dei principali vantaggi offerti dagli *accessors* è che permettono di migliorare il controllo di una normale *data property* rimpiazzandola con le proprietà *getter* e *setter* e lavorando sul loro comportamento.

Immaginiamo di iniziare ad implementare l'oggetto `user` con le proprietà `name` e `age` °

```

function User(name, age) {
  this.name = name;
  this.age = age;
}

let john = new User("John", 25);

alert(john.age); // 25

```

...Ma prima o poi, le cose potrebbero cambiare. Invece di `age` potremmo decidere di memorizzare `birthday`, poiché è più preciso e conveniente:

```

function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;
}

let john = new User("John", new Date(1992, 6, 1));

```

Ora come ci comportiamo con il codice “vecchio” che utilizza ancora la proprietà `age`?

Possiamo provare a cercare tutti i posti in cui viene utilizzata nel codice e sistemarlo, ma questo potrebbe richiedere tempo e potrebbe essere ancora più complesso se lo stesso codice viene utilizzato da altre persone. E in ogni caso, `age` è una proprietà utile da avere in `user`, giusto?

Quindi manteniamola.

Aggiungere un *getter* per `age` risolve il problema:

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;

  // age viene calcolata utilizzando la data attuale ed il compleanno
  Object.defineProperty(this, "age", {
    get() {
      let todayYear = new Date().getFullYear();
      return todayYear - this.birthday.getFullYear();
    }
  });
}

let john = new User("John", new Date(1992, 6, 1));

alert(john.birthday); // birthday è disponibile
alert(john.age);     // ...è lo è anche age
```

In questo modo il codice “vecchio” continua a funzionare e abbiamo anche guadagnato un’ottima proprietà aggiuntiva.

Prototypes, inheritance

Prototypal inheritance

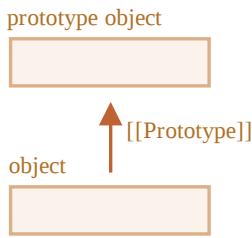
Nella programmazione, spesso vogliamo prendere qualcosa ed estenderla.

Ad esempio, potremmo avere un oggetto `user` con le sue proprietà e i suoi metodi, e voler definire gli oggetti `admin` e `guest` come sue varianti. Vorremmo però poter riutilizzare ciò che abbiamo nell’oggetto `user`, evitando di copiare e reimplementare nuovamente i suoi metodi, quindi vorremmo semplicemente definire un nuovo oggetto a partire da esso.

La *prototypal inheritance* (ereditarietà dei prototype) è una caratteristica del linguaggio che aiuta in questo senso.

[[Prototype]]

In JavaScript, gli oggetti possiedono una speciale proprietà nascosta `[[Prototype]]` (come definito nella specifica); questo può valere `null` oppure può contenere il riferimento ad un altro oggetto. Quell’oggetto viene definito “prototype” (prototipo):



Quando leggiamo una proprietà da `object`, e questa non esiste, JavaScript prova automaticamente a recuperarla dal suo prototype. In programmazione, questo comportamento viene definito “prototypal inheritance”. Presto vederemo diversi esempi di questo tipo di ereditarietà, e vedremo anche delle interessanti caratteristiche del linguaggio basate su di essa.

La proprietà `[[Prototype]]` è interna e nascosta, ma esistono diversi modi per poterla impostare.

Uno di questi è quello di utilizzare la nomenclatura speciale `__proto__`:

```

let animal = {
  eats: true
};

let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal; // imposta il prototype di rabbit,.[[Prototype]] = animal

```

Ora se proviamo a leggere una proprietà da `rabbit`, e questa risulta essere mancante, JavaScript andrà a prenderla automaticamente da `animal`.

Ad esempio:

```

let animal = {
  eats: true
};

let rabbit = {
  jumps: true
};

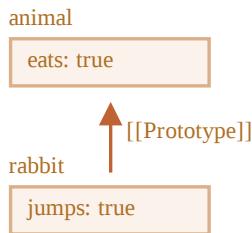
rabbit.__proto__ = animal; // (*)

// ora in rabbit possiamo trovare entrambe le proprietà
alert( rabbit.eats ); // true (**)
alert( rabbit.jumps ); // true

```

Nell'esempio la linea `(*)` imposta `animal` come prototype di `rabbit`.

Successivamente, quando `alert` proverà a leggere la proprietà `rabbit.eats` `(**)`, non la troverà in `rabbit`, quindi JavaScript seguirà il riferimento in `[[Prototype]]` e la troverà in `animal` (ricerca dal basso verso l'alto):



In questo caso possiamo dire che "animal" è il prototype di "rabbit" o, in alternativa, che "rabbit" *prototypically inherits* (eredità dal prototipo) da "animal".

Quindi se `animal` possiede molte proprietà e metodi utili, questi saranno automaticamente disponibili in `rabbit`. Queste proprietà vengono definite come "ereditate".

Se abbiamo un metodo in `animal`, possiamo invocarlo anche in `rabbit`:

```

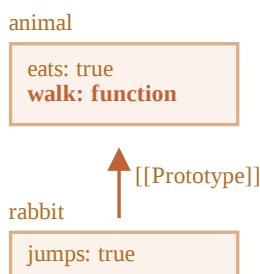
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// walk viene ereditato dal prototype
rabbit.walk(); // Animal walk

```

Il metodo viene preso automaticamente dal prototipo, in questo modo:



La catena dei prototype può esser anche più lunga:

```

let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

```

```

let longEar = {
  earLength: 10,
  __proto__: rabbit
};

// walk viene presa dalla catena di prototype
longEar.walk(); // Animal walk
alert(longEar.jumps); // true (da rabbit)

```

animal

eats: true
walk: function

\uparrow [[Prototype]]

rabbit

jumps: true

\uparrow [[Prototype]]

longEar

earLength: 10

Ora, se provassimo a leggere qualcosa da `longEar`, e non esistesse, JavaScript andrebbe a guardare prima in `rabbit`, e poi in `animal`.

Ci sono solamente due limitazioni:

1. Non possono esserci riferimenti circolari. JavaScript restituirebbe un errore se provassimo ad assegnare a `__proto__` un riferimento circolare.
2. Il valore di `__proto__` può essere o un oggetto o `null`. Gli altri valori vengono ignorati.

Inoltre, anche se dovrebbe essere già ovvio: può esserci solamente un `[[Prototype]]`. Un oggetto non può ereditare da più oggetti.

i `__proto__` è un getter/setter storico per `[[Prototype]]`

E' un errore comune tra i principianti quello di non conoscere la differenza tra questi due.

Da notare che `__proto__` non è *la stessa cosa* della proprietà `[[Prototype]]`. E' solamente un getter/setter per `[[Prototype]]`. Più avanti vedremo alcune situazioni in cui questa differenza avrà importanza, ma per ora tenetelo solo a mente.

La proprietà `__proto__` è leggermente datata. Esiste solamente per ragioni storiche, la versione attuale di JavaScript suggerisce di utilizzare le funzioni `Object.getPrototypeOf/Object.setPrototypeOf` per impostare il prototype. Vedremo meglio queste funzioni più avanti.

Secondo la specifica, `__proto__` deve essere supportato solamente dai browser. In realtà, tutti gli ambienti, inclusi quelli server-side, supportano `__proto__`, quindi il suo utilizzo è piuttosto sicuro.

Poiché la notazione `__proto__` risulta essere più intuitiva, la utilizzeremo nei nostri esempi.

La scrittura non utilizza prototype

Il prototype viene utilizzato solamente per la lettura delle proprietà.

Le operazioni di scrittura/rimozione utilizzano direttamente l'oggetto.

Nell'esempio che vediamo sotto, assegniamo un metodo `walk` a `rabbit`, che sarà solo suo:

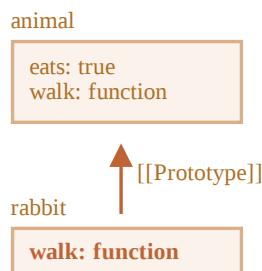
```
let animal = {
  eats: true,
  walk() {
    /* questo metodo non verrà utilizzato da rabbit */
  }
};

let rabbit = {
  __proto__: animal
};

rabbit.walk = function() {
  alert("Rabbit! Bounce-bounce!");
};

rabbit.walk(); // Rabbit! Bounce-bounce!
```

Da questo punto in poi, la chiamata `rabbit.walk()` troverà il metodo direttamente nell'oggetto e lo eseguirà, senza utilizzare il prototype:



Le proprietà di accesso sono delle eccezioni, poiché l'assegnazione viene gestita da un setter. Quindi scrivere su una proprietà di questo tipo equivale ad invocare una funzione.

Per questo motivo, `admin.fullName` funziona correttamente nel codice sotto:

```
let user = {
  name: "John",
  surname: "Smith",

  set fullName(value) {
    [this.name, this.surname] = value.split(" ");
  },

  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

let admin = {
```

```

__proto__: user,
isAdmin: true
};

alert(admin.fullName); // John Smith (*)

// il setter viene invocato!
admin.fullName = "Alice Cooper"; // (**)

alert(admin.fullName); // Alice Cooper, lo stato di admin è stato modificato
alert(user.fullName); // John Smith, lo stato di user è protetto

```

Nell'esempio, alla linea (*) la proprietà `admin.fullName` ha un getter nel prototype `user`, quindi viene invocato. E alla linea (**) la proprietà ha un setter nel prototype, che viene quindi invocato.

Il valore di “this”

Dall'esempio sopra potrebbe sorgere una domanda interessante: qual è il valore di `this` all'interno `set fullName(value)`? Dove vengono scritte le proprietà `this.name` e `this.surname`: in `user` o `admin`?

La risposta è semplice: `this` non viene influenzato dai prototype.

Non ha importanza dove viene trovato il metodo: nell'oggetto o in un suo prototype. Quando invochiamo un metodo, `this` fa sempre riferimento all'oggetto che precede il punto.

Quindi, l'invocazione del setter `admin.fullName=` utilizza `admin` come `this`, non `user`.

Questo è molto importante, poiché potremmo avere un oggetto molto grande con molti metodi, e avere diversi oggetti che ereditano da esso. Quando gli oggetti che ereditano eseguono un metodo ereditato, andranno a modificare solamente il loro stato, non quello dell'oggetto principale da cui ereditano.

Ad esempio, qui `animal` rappresenta un “archivio di metodi”, che `rabbit` utilizza.

La chiamata `rabbit.sleep()` imposta `this.isSleeping` nell'oggetto `rabbit`:

```

// animal possiede dei metodi
let animal = {
  walk() {
    if (!this.isSleeping) {
      alert(`I walk`);
    }
  },
  sleep() {
    this.isSleeping = true;
  }
};

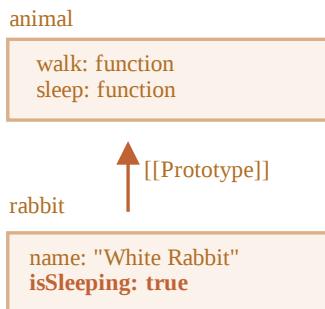
let rabbit = {
  name: "White Rabbit",
  __proto__: animal
};

```

```
// modifica rabbit.isSleeping
rabbit.sleep();

alert(rabbit.isSleeping); // true
alert(animal.isSleeping); // undefined (non esiste questa proprietà nel prototype)
```

Il risultato:



Se avessimo altri oggetti, come `bird`, `snake`, etc., che ereditano da `animal`, avrebbero a loro volta accesso ai metodi di `animal`. In ogni caso, `this` all'interno della chiamata farebbe riferimento all'oggetto corrispondente, che viene valutato al momento dell'invocazione (appena prima del punto), e non ad `animal`. Quindi quando scriviamo dati utilizzando `this`, questi verranno memorizzati nell'oggetto corrispondente.

Come risultato i metodi sono condivisi, mentre lo stato degli oggetti non lo è.

Il ciclo for...in

Il ciclo `for .. in` itera anche le proprietà ereditate.

Ad esempio:

```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// Object.keys ritorna solamente le chiavi
alert(Object.keys(rabbit)); // jumps

// il ciclo for..in itera sia le proprietà di rabbit, che quelle ereditate da animal
for(let prop in rabbit) alert(prop); // jumps, then eats
```

Se questo non è ciò che ci aspettiamo, e vogliamo escludere le proprietà ereditate, esiste un metodo integrato `obj.hasOwnProperty(key)`: ritorna `true` se `obj` possiede la propria proprietà `key` (non ereditata).

Quindi possiamo filtrare le proprietà ereditate (o farci qualcos'altro):

```
let animal = {
```

```

        eats: true
    };

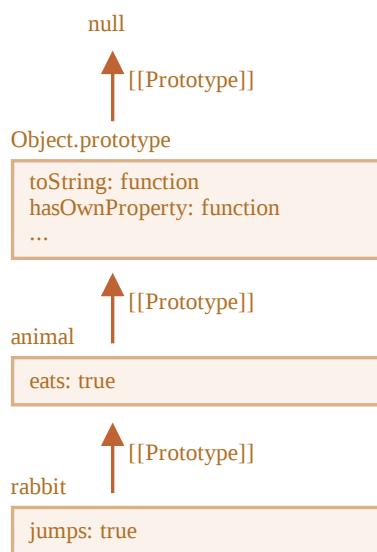
let rabbit = {
    jumps: true,
    __proto__: animal
};

for(let prop in rabbit) {
    let isOwn = rabbit.hasOwnProperty(prop);

    if (isOwn) {
        alert(`Our: ${prop}`); // Our: jumps
    } else {
        alert(`Inherited: ${prop}`); // Inherited: eats
    }
}

```

Qui abbiamo la seguente catena di ereditarietà: `rabbit` eredita da `animal`, che eredita da `Object.prototype` (poiché `animal` è un *literal objects* `{...}`), e infine `null`:



Da notare, c'è una cosa divertente. Da dove arriva il metodo `rabbit.hasOwnProperty`? Noi non lo abbiamo mai definito. Osservando la catena ci accorgiamo che il metodo viene fornito da `Object.prototype.hasOwnProperty`. In altre parole, è ereditato.

...Ma perché `hasOwnProperty` non appare nel ciclo `for..in` come `eats` e `jumps`, se `for..in` elenca tutte le proprietà ereditate?

La risposta è semplice: la proprietà è *non enumerable*. Come tutte le altre proprietà di `Object.prototype`, possiedono la flag `enumerable:false`. Quindi `for..in` elenca solamente le proprietà *enumerable*. Questo è il motivo per cui le proprietà di `Object.prototype` non vengono elencate.

i Quasi tutti gli altri metodi getter key-value ignorano le proprietà ereditate

Quasi tutti gli altri metodi getter key-value, come `Object.keys`, `Object.values` e così via, ignorano le proprietà ereditate.

Questi metodi lavorano solamente sull'oggetto stesso. Le proprietà di prototype *non* vengono prese in considerazione.

Riepilogo

- In JavaScript, tutti gli oggetti possiedono una proprietà nascosta `[[Prototype]]` che può essere il riferimento ad un altro oggetto, oppure `null`.
- Possiamo utilizzare `obj.__proto__` per accedervi (una proprietà getter/setter storica, ci sono altri modi che vederemo presto).
- L'oggetto a cui fa riferimento `[[Prototype]]` viene chiamato “prototype”.
- Se vogliamo leggere una proprietà di `obj` o invocare un metodo, ma questo non esiste, allora JavaScript andrà a cercarlo nel prototype.
- Le operazioni di scrittura/rimozione agiscono direttamente sull'oggetto, non utilizzano il prototype (assumendo che questa sia una proprietà e non un setter).
- Se invochiamo `obj.method()`, e il `method` viene prelevato dal prototype, `this` farà comunque riferimento a `obj`. Quindi i metodi lavoreranno sempre con l'oggetto corrente, anche se questi sono ereditati.
- Il ciclo `for..in` itera sia le proprietà dell'oggetto che quelle ereditate. Tutti gli altri metodi di tipo getter key/value operano solamente sull'oggetto stesso.

✓ Esercizi

Lavorare con prototype

importanza: 5

Il seguente codice crea due oggetti, e successivamente li modifica.

Quali valori vengono mostrati nel processo?

```
let animal = {  
    jumps: null  
};  
let rabbit = {  
    __proto__: animal,  
    jumps: true  
};  
  
alert( rabbit.jumps ); // ? (1)  
  
delete rabbit.jumps;  
  
alert( rabbit.jumps ); // ? (2)  
  
delete animal.jumps;
```

```
alert( rabbit.jumps ); // ? (3)
```

Dovrebbero esserci 3 risposte.

[Alla soluzione](#)

Algoritmo di ricerca

importanza: 5

Il task è suddiviso in due parti.

Dati i seguenti oggetti:

```
let head = {  
  glasses: 1  
};  
  
let table = {  
  pen: 3  
};  
  
let bed = {  
  sheet: 1,  
  pillow: 2  
};  
  
let pockets = {  
  money: 2000  
};
```

1. Utilizza `__proto__` per assegnare il prototypes in modo che la catena segua il percorso: `pockets → bed → table → head`. Ad esempio, `pockets.pen` dovrebbe essere `3` (in `table`), e `bed.glasses` dovrebbe essere `1` (in `head`).
2. Rispondi alla domanda: è più veloce ottenere `glasses` come `pockets.glasses` o come `head.glasses`? Esegui test se necessario.

[Alla soluzione](#)

Dove andrà a scrivere?

importanza: 5

Abbiamo un oggetto `rabbit` che eredita da `animal`.

Se invochiamo `rabbit.eat()`, quale oggetto riceverà la proprietà `full: animal` o `rabbit`?

```
let animal = {  
  eat() {  
    this.full = true;  
  }  
}
```

```
};

let rabbit = {
  __proto__: animal
};

rabbit.eat();
```

[Alla soluzione](#)

Perché entrambi i criceti sono sazi?

importanza: 5

Abbiamo due criceti: `speedy` e `lazy`, che ereditano dall'oggetto `hamster`.

Quando nutriamo uno di loro, anche l'altro è sazio. Perché? Come possiamo sistemare il problema?

```
let hamster = {
  stomach: [],

  eat(food) {
    this.stomach.push(food);
  }
};

let speedy = {
  __proto__: hamster
};

let lazy = {
  __proto__: hamster
};

// Questo ha trovato il cibo
speedy.eat("apple");
alert( speedy.stomach ); // apple

// Anche questo lo ha ricevuto, perché? provate a sistemarlo
alert( lazy.stomach ); // apple
```

[Alla soluzione](#)

F.prototype

Ricordate, nuovi oggetti possono essere creati con un costruttore, come `new F()`.

Se `F.prototype` è un oggetto, l'operatore `new` si prenderà cura di impostare `[[Prototype]]` per il nuovo oggetto.

i Da notare:

JavaScript supporta la prototypal inheritance fin dall'inizio. Fu una delle caratteristiche principali del linguaggio.

Ma all'inizio non c'era un accesso diretto. L'unica cosa su cui ci si poteva affidare era la proprietà "prototype" del costruttore, descritta in questo capitolo. Per questo, esistono ancora molti script che ne fanno utilizzo.

Da notare che qui `F.prototype`, sta per una comune proprietà chiamata "prototype" in `F`. Sembra molto simile al termine "prototype", ma in questo caso intendiamo realmente riferirci ad una proprietà con questo nome.

Vediamo qui un esempio:

```
let animal = {
  eats: true
};

function Rabbit(name) {
  this.name = name;
}

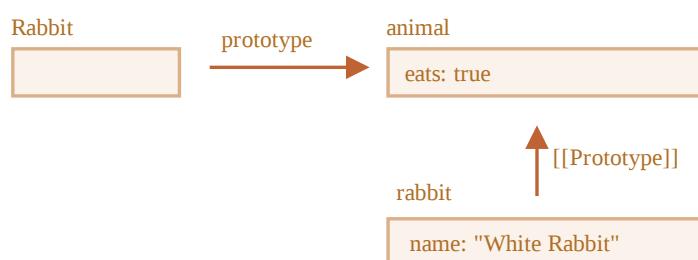
Rabbit.prototype = animal;

let rabbit = new Rabbit("White Rabbit"); // rabbit.__proto__ == animal

alert( rabbit.eats ); // true
```

Impostare `Rabbit.prototype = animal` fa letteralmente quanto segue: "Quando un nuovo `new Rabbit` viene creato, assegna il suo `[[Prototype]]` ad `animal`".

Questo è il risultato:



In figura, "prototype" è una freccia orizzontale, ciò significa che è una comune proprietà, mentre `[[Prototype]]` è verticale, quindi `rabbit` eredita da `animal`.

i `F.prototype` viene utilizzato solamente al momento in cui si invoca `new F`

`F.prototype` viene utilizzata solamente quando si invoca `new F`, e si occupa di assegnare `[[Prototype]]` del nuovo oggetto.

Se, dopo la creazione, `F.prototype` cambia (`F.prototype = <another object>`), allora verrà creato un nuovo oggetto con `new F` che avrà un altro oggetto come `[[Prototype]]`, ma gli oggetti già esistenti faranno riferimento a quello vecchio.

Default F.prototype, la proprietà constructor

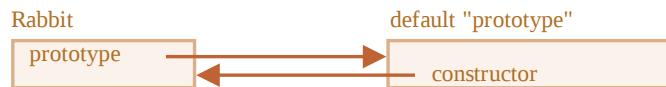
Ogni funzione possiede la proprietà "prototype" anche se non gliela forniamo direttamente.

Il "prototype" di default è un oggetto con un'unica proprietà, il `constructor` che punta alla funzione stessa.

Vediamo un esempio:

```
function Rabbit() {}

/* default prototype
Rabbit.prototype = { constructor: Rabbit };
*/
```



Possiamo verificarlo:

```
function Rabbit() {}
// di default:
// Rabbit.prototype = { constructor: Rabbit }

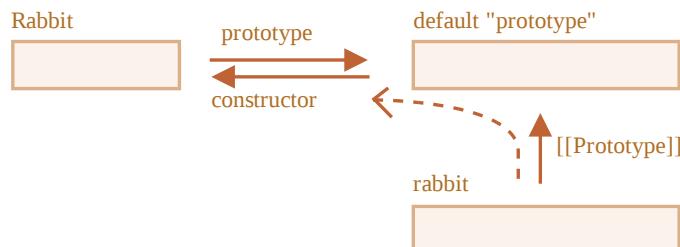
alert( Rabbit.prototype.constructor == Rabbit ); // true
```

Naturalmente, se non facciamo nulla, il `constructor` sarà disponibile a tutti i `rabbit` attraverso `[[Prototype]]`:

```
function Rabbit() {}
// di default:
// Rabbit.prototype = { constructor: Rabbit }

let rabbit = new Rabbit(); // eredita da {constructor: Rabbit}

alert(rabbit.constructor == Rabbit); // true (dal prototype)
```



Possiamo utilizzare il `constructor` per creare un nuovo oggetto utilizzando lo stesso costruttore dell'oggetto già esistente.

Come nell'esempio:

```

function Rabbit(name) {
  this.name = name;
  alert(name);
}

let rabbit = new Rabbit("White Rabbit");

let rabbit2 = new rabbit.constructor("Black Rabbit");

```

Questo torna molto utile quando abbiamo un oggetto, ma non sappiamo quale costruttore è stato utilizzato (ad esempio se arriva da una libreria di terze parti), e abbiamo bisogno di crearne un altro dello stesso tipo.

Ma probabilmente la cosa più importante del "constructor" è che...

...JavaScript stesso non garantisce il giusto valore del "constructor".

E' vero, esiste di default nel "prototype" delle funzioni, ma questo è tutto. Ciò che accade dopo – è solo nostra responsabilità.

In particolare, se rimpiazziamo completamente il prototype di default, allora non ci sarà alcun "constructor".

Ad esempio:

```

function Rabbit() {}
Rabbit.prototype = {
  jumps: true
};

let rabbit = new Rabbit();
alert(rabbit.constructor === Rabbit); // false

```

Quindi, per mantenere il "constructor" corretto, possiamo decidere di aggiungere/rimuovere proprietà al "prototype" di default, invece che sovrascriverlo completamente:

```

function Rabbit() {}

// Non sovrascriviamo Rabbit.prototype completamente
// aggiungiamo semplicemente una proprietà
Rabbit.prototype.jumps = true
// il Rabbit.prototype.constructor viene così preservato

```

O, in alternativa, possiamo ricreare il constructor manualmente:

```

Rabbit.prototype = {
  jumps: true,
  constructor: Rabbit
};

// ora il costruttore è corretto, perché lo abbiamo aggiunto noi

```

Riepilogo

In questo capitolo abbiamo descritto brevemente il modo in cui impostare il `[[Prototype]]` per gli oggetti generati tramite il costruttore. Più avanti vedremo dei pattern più avanzati su cui fare affidamento.

Il tutto è abbastanza semplice, solo alcune note per renderlo più chiaro:

- La proprietà `F.prototype` (da non confondere con `[[Prototype]]`) imposta `[[Prototype]]` dei nuovi oggetti quando viene invocato `new F()`.
- Il valore di `F.prototype` può essere sia un oggetto che `null`: altri valori verranno ignorati.
- La proprietà `"prototype"` ha un effetto speciale quando impostata in un costruttore, ed invocata con `new`.

Negli oggetti “comuni” la proprietà `prototype` non ha alcun significato speciale:

```
let user = {
  name: "John",
  prototype: "Bla-bla" // nessuna magia
};
```

Di default tutte le funzioni hanno `F.prototype = { constructor: F }`, quindi possiamo ottenere il costruttore di un oggetto accedendo alla sua proprietà `"constructor"`.

✓ Esercizi

Cambiare "prototype"

importanza: 5

Nel codice sotto, andiamo a creare `new Rabbit`, e successivamente proviamo a modificare il suo prototype.

Inizialmente, abbiamo questo codice:

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

alert( rabbit.eats ); // true
```

1.

Aggiungiamo una o più stringhe. Cosa mostrerà `alert` ora?

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();
```

```
Rabbit.prototype = {};  
  
alert( rabbit.eats ); // ?
```

2.

...E se il codice è come il seguente (abbiamo rimpiazzato una sola riga)?

```
function Rabbit() {}  
Rabbit.prototype = {  
  eats: true  
};  
  
let rabbit = new Rabbit();  
  
Rabbit.prototype.eats = false;  
  
alert( rabbit.eats ); // ?
```

3.

E in questo caso (abbiamo rimpiazzato solo una riga)?

```
function Rabbit() {}  
Rabbit.prototype = {  
  eats: true  
};  
  
let rabbit = new Rabbit();  
  
delete rabbit.eats;  
  
alert( rabbit.eats ); // ?
```

4.

L'ultima variante:

```
function Rabbit() {}  
Rabbit.prototype = {  
  eats: true  
};  
  
let rabbit = new Rabbit();  
  
delete Rabbit.prototype.eats;  
  
alert( rabbit.eats ); // ?
```

[Alla soluzione](#)

Crea un oggetto con lo stesso costruttore

importanza: 5

Immagina di avere un oggetto arbitrario `obj`, creato da un costruttore – non sappiamo quale, ma vorremmo poter creare un nuovo oggetto utilizzandolo.

Possiamo farlo in questo modo?

```
let obj2 = new obj.constructor();
```

Fornite un esempio di costruttore per `obj` che permetta a questo codice di funzionare correttamente. Ed un esempio che non lo farebbe funzionare.

[Alla soluzione](#)

Native prototypes

La proprietà "prototype" viene largamente utilizzata da JavaScript stesso. Tutti i costruttori integrati ne fanno uso.

Come prima cosa andremo ad analizzare questa proprietà nel dettaglio; in un secondo momento vedremo come utilizzarla per aggiungere nuove funzionalità agli oggetti integrati.

Object.prototype

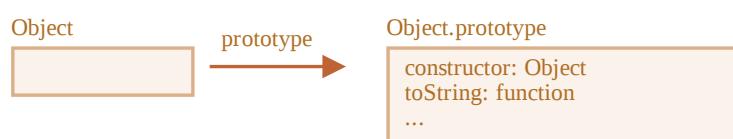
Ipotizziamo di dover mostrare un oggetto vuoto:

```
let obj = {};
alert( obj ); // "[object Object]" ?
```

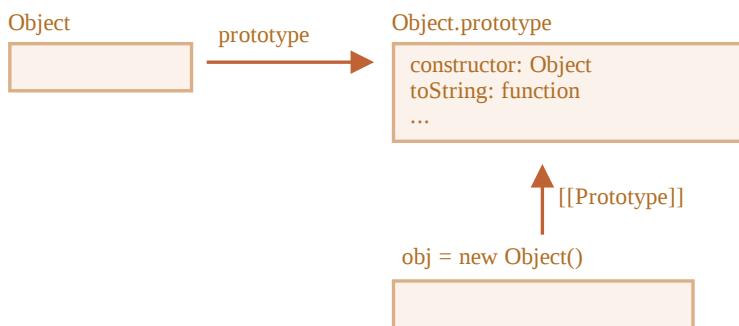
Da dove arriva il codice che genera la stringa "[object Object]"? E' il metodo integrato `toString`, ma dove lo possiamo trovare? L'oggetto `obj` è vuoto!

...La notazione `obj = {}` equivale a `obj = new Object()`, dove `Object` è un costruttore integrato, in cui la proprietà `prototype` fa riferimento ad un oggetto con `toString` e altri metodi.

Questo è ciò che accade:



Quando viene invocato `new Object()` (o viene creato un literal object `{...}`), il suo `[[Prototype]]` viene impostato a `Object.prototype`, come abbiamo studiato nel capitolo precedente:



Quindi, quando `obj.toString()` viene invocato, il metodo viene cercato in `Object.prototype`.

Possiamo verificarlo in questo modo:

```

let obj = {};
alert(obj.__proto__ === Object.prototype); // true
alert(obj.toString === obj.__proto__.toString); //true
alert(obj.toString === Object.prototype.toString); //true

```

Da notare che non esistono ulteriori `[[Prototype]]` nella catena `Object.prototype`:

```
alert(Object.prototype.__proto__); // null
```

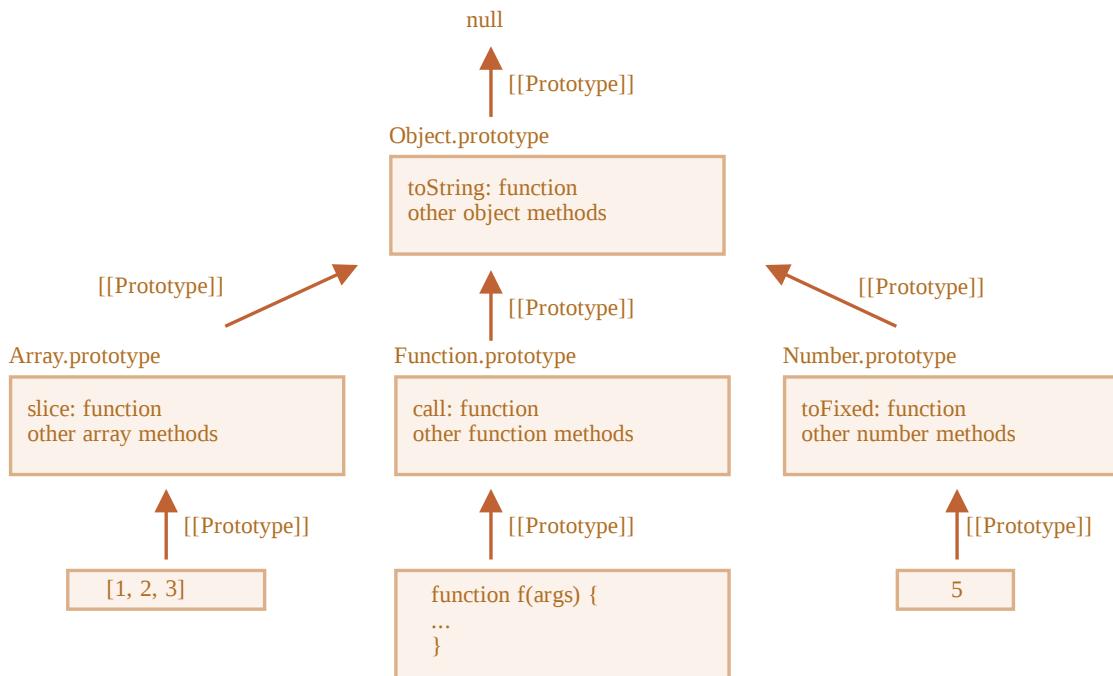
Altri prototypes integrati

Altri oggetti integrati, come `Array`, `Date`, `Function` ed altri, hanno i propri metodi in prototypes.

Ad esempio, quando creiamo un array `[1, 2, 3]`, il costruttore di default `new Array()` viene invocato internamente. Quindi `Array.prototype` ne diventa il prototipo e fornisce i suoi metodi. Questo comportamento rende l'utilizzo della memoria molto efficiente.

Come definito nella specifica, tutti i prototype integrati hanno `Object.prototype` in cima. Questo è il motivo per cui alcune persone dicono che “tutto deriva dagli oggetti”.

Qui vediamo il quadro complessivo (di 3 oggetti integrati):



Proviamo a controllare il prototype manualmente:

```

let arr = [1, 2, 3];

// eredita da Array.prototype?
alert( arr.__proto__ === Array.prototype ); // true

// e successivamente da Object.prototype?
alert( arr.__proto__.__proto__ === Object.prototype ); // true

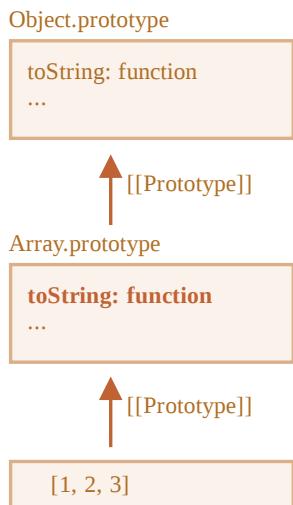
// e infine null
alert( arr.__proto__.__proto__.__proto__ ); // null
    
```

Alcuni metodi in prototype potrebbero essere stati sovrascritti, ad esempio, `Array.prototype` possiede una sua implementazione personalizzata di `toString`, che elenca gli elementi separandoli con una virgola:

```

let arr = [1, 2, 3]
alert(arr); // 1,2,3 <-- il risultato di Array.prototype.toString
    
```

Come abbiamo visto in precedenza, `Object.prototype` possiede una sua implementazione di `toString`, ma poiché `Array.prototype` è molto più vicina nella catena dei prototype, questa sarà la variante utilizzata.



Strumenti integrati nel browser, come la console che puoi trovare nei Chrome developer tools (strumenti per sviluppatori), mostrano l'ereditarietà (potreste utilizzare `console.dir` per gli oggetti integrati):

```

> console.dir([1,2,3])
▼ Array[3] ⓘ
  0: 1
  1: 2
  2: 3
  length: 3
  ▼ __proto__:=Array.prototype
    ► concat: function concat() { [native code] }
    ► ...
    ► unshift: function unshift() { [native code] }
    ▼ __proto__:=Object.prototype
      ► ...
      ► constructor: function Object() { [native code] }
      ► hasOwnProperty: function hasOwnProperty() { [native code] }
      ► isPrototypeOf: function isPrototypeOf() { [native code] }
      ► ...
  
```

Gli altri oggetti integrati funzionano allo stesso modo. Anche le funzioni – poiché sono oggetti di un costruttore integrato `Function`, e i suoi metodi (`call` / `apply` e gli altri) sono presi da `Function.prototype`. Anche le funzioni possiedono una loro implementazione di `toString`.

```

function f() {}

alert(f.__proto__ == Function.prototype); // true
alert(f.__proto__.__proto__ == Object.prototype); // true, eredità da objects
  
```

Primitivi

La situazione è molto più complessa con strings, numbers e booleans.

Come abbiamo già visto, questi non sono oggetti. Ma se proviamo ad accedere alle loro proprietà, viene creato un oggetto temporaneo utilizzando i rispettivi costruttori `String`, `Number` e `Boolean`. Essi forniscono metodi e poi spariscono.

Questi oggetti vengono creati di “nascosto” e in realtà molti motori ottimizzano il loro utilizzo, ma la specifica li descrive in questo modo. I metodi di questi oggetti sono memorizzati nella proprietà del loro prototype, e sono disponibili tramite `String.prototype`, `Number.prototype` e `Boolean.prototype`.

⚠️ I valori `null` e `undefined` non possiedono degli oggetti che li contengono

I valori speciali `null` e `undefined` si comportano diversamente. Non possiedono degli oggetti contenitori, quindi non avremmo a disposizione proprietà e metodi. E non avremmo nemmeno il propotype corrispondente.

Modificare i native prototypes

I Native prototypes possono essere modificati. Ad esempio, se aggiungiamo il metodo `String.prototype`, questo diventa disponibile a tutte le string:

```
String.prototype.show = function() {
  alert(this);
};

"BOOM!".show(); // BOOM!
```

Durante lo sviluppo, potremmo avere bisogno di nuovi metodi integrati che ci piacerebbe avere, e potremmo quindi essere tentati di aggiungerli ai native prototype. Generalmente questa è una pessima idea.

⚠️ Importante:

I prototype sono globali, quindi è molto facile generare conflitti. Se due librerie differenti aggiungono un metodo `String.prototype.show`, allora uno dei due sovrascriverà l'altro.

Quindi, generalmente, modificare i native prototype viene considerata una cattiva pratica.

Nella programmazione moderna, esiste solamente un caso in cui è accettabile sovrascrivere i native prototype. Per fare polyfilling.

Polyfilling è una pratica che prevede di sostituire un oggetto definito nella specifica JavaScript, che non è ancora stato implementato da un particolare engine.

Possiamo implementarlo noi manualmente e popolare i prototype integrati con la nostra implementazione.

Ad esempio:

```
if (!String.prototype.repeat) { // se questo metodo non esiste
  // lo aggiungiamo al prototype

  String.prototype.repeat = function(n) {
    // ripetiamo la stringa n volte

    // in realtà, il codice dovrebbe essere leggermente più complesso di così
    // (l'algoritmo completo è descritto nella specifica)
```

```

    // ma generalmente anche un polyfill imperfetto è considerato sufficiente
    return new Array(n + 1).join(this);
}
}

alert( "La".repeat(3) ); // LaLaLa

```

Prendere in prestito dai prototypes

Nel capitolo [*Decorators* e forwarding, call/apply](#) abbiamo parlato di come “prendere in prestito” metodi.

Questo avviene quando prendiamo un metodo da un oggetto e lo copiamo in un altro.

Alcuni metodi dei native prototype sono presi in prestito.

Ad esempio, se stiamo costruendo un oggetto simil-array, potremmo voler copiare alcuni metodi degli `Array`.

Esempio.

```

let obj = {
  0: "Hello",
  1: "world!",
  length: 2,
};

obj.join = Array.prototype.join;

alert( obj.join(',') ); // Hello,world!

```

Funziona perché l’algoritmo integrato del metodo `join` necessita solamente degli indici corretti e della proprietà `length`. Non va a verificare se un oggetto sia effettivamente un array. Molti metodi integrati si comportano in questo modo.

Un’altra possibilità è quella di ereditare di default `obj.__proto__` da `Array.prototype`, quindi tutti i metodi di `Array` diventeranno automaticamente disponibili in `obj`.

Ma questo è impossibile se `obj` eredita già da un altro oggetto. Ricorda, possiamo ereditare solamente da un oggetto per volta.

La pratica di “prendere in prestito” i metodi è flessibile, consente di ereditare funzionalità da oggetti differenti se necessario.

Riepilogo

- Tutti gli oggetti integrati seguono lo stesso comportamento:
 - I metodi vengono memorizzati nel prototype (`Array.prototype`, `Object.prototype`, `Date.prototype`, etc.)
 - L’oggetto memorizza solamente i dati (gli elementi dell’array, le proprietà dell’object, la data)
- I tipi di dato primitivi memorizzano i metodi nel prototype, utilizzando degli oggetti “contenitori”: `Number.prototype`, `String.prototype` e `Boolean.prototype`. Fanno eccezione

`undefined` e `null` che non possiedono alcun oggetto contenitore.

- I prototype integrati possono essere modificati o popolati con nuovi metodi. Ma questa è una pratica sconsigliata. L'unico caso in cui è accettabile aggiungere nuovi metodi è per fornire l'implementazione di funzionalità definite nella specifica JavaScript agli engines che ancora non le supportano.

✓ Esercizi

Aggiungi il metodo "f.defer(ms)" alle funzioni

importanza: 5

Aggiungi al prototype di tutte le funzioni il metodo `defer(ms)`, che si occupa di eseguire la funzione dopo `ms` millisecondi.

Una volta fatto, il seguente codice dovrebbe funzionare:

```
function f() {  
  alert("Hello!");  
}  
  
f.defer(1000); // mostra "Hello!" dopo 1 secondo
```

[Alla soluzione](#)

Aggiungi il decorator "defer()" alle funzioni

importanza: 4

Aggiungi al prototype di tutte le funzioni il metodo `defer(ms)`, il quale ritorna un wrapper (contenitore), che si occupa di invocare la funzione dopo `ms` millisecondi.

Qui vediamo un esempio di come dovrebbe funzionare:

```
function f(a, b) {  
  alert( a + b );  
}  
  
f.defer(1000)(1, 2); // mostra 3 dopo 1 secondo
```

Da notare che gli argomenti devono essere passati alla funzione originale.

[Alla soluzione](#)

Metodi di prototype, objects senza `__proto__`

Nel primo capitolo di questa sezione, abbiamo menzionato il fatto che esistono metodi più moderni per impostare il prototype.

La proprietà `__proto__` viene considerata datata, e in un certo senso anche deprecata (negli standard JavaScript per i browser).

Alcuni dei metodi più moderni sono:

- `Object.create(proto, [descriptors])` – crea un oggetto vuoto, impostando il `proto` come `[[Prototype]]` e dei descrittori di proprietà opzionali.
- `Object.getPrototypeOf(obj)` – ritorna il `[[Prototype]]` di `obj`.
- `Object.setPrototypeOf(obj, proto)` – imposta il `[[Prototype]]` di `obj` a `proto`.

Questi metodi dovrebbero sempre essere preferiti a `__proto__`.

Ad esempio:

```
let animal = {
  eats: true
};

// creiamo un nuovo oggetto con animal come prototype
let rabbit = Object.create(animal);

alert(rabbit.eats); // true

alert(Object.getPrototypeOf(rabbit) === animal); // true

Object.setPrototypeOf(rabbit, {}); // cambia il prototype di rabbit a {}
```

`Object.create` supporta un secondo argomento opzionale: il `property descriptors` (descrittori di proprietà). Possiamo fornire proprietà aggiuntive al nuovo oggetto, in questo modo:

```
let animal = {
  eats: true
};

let rabbit = Object.create(animal, {
  jumps: {
    value: true
  }
});

alert(rabbit.jumps); // true
```

I descrittori vanno forniti nel formato descritto nel capitolo [Attributi e descrittori di proprietà](#).

Possiamo utilizzare `Object.create` per clonare un oggetto in maniera più efficace rispetto al copiare le proprietà con un `for..in`:

```
let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));
```

Questa chiamata crea una copia esatta di `obj`, includendo tutte le proprietà: enumerable e non-enumerable, e i relativi setters/getters – tutto, impostando anche il giusto `[[Prototype]]`.

Una breve storia

Se contiamo tutti i modi che abbiamo a disposizione per gestire `[[Prototype]]`, questi sono molti! Abbiamo moltissime modalità per fare la stessa cosa!

Perché?

Motivazioni storiche.

- La proprietà "prototype" di un costruttore è disponibile fin dai primi tempi in JavaScript.
- Più tardi, nel 2012, è apparso nello standard `Object.create`. Il quale permette di creare oggetti fornendogli un prototype, ma non consente di impostarlo o di ottenerlo. Quindi i browser implementarono il metodo non-standard `__proto__`, come proprietà di accesso per impostare o ottenere il prototype in qualsiasi momento.
- Più tardi, nel 2015, `Object.setPrototypeOf` e `Object.getPrototypeOf` vennero aggiunti allo standard, con le stesse funzionalità di `__proto__`. Poiché `__proto__` era di fatto implementato ovunque, entrò in un fase di deprecazione, nella sezione Annex B dello standard, ovvero: opzionale per gli ambienti non-browser.

Ad oggi abbiamo molti metodi a nostra disposizione.

Perché `__proto__` è stato rimpiazzato dalle funzioni

`getPrototypeOf/setPrototypeOf`? Questa è una domanda interessante; dobbiamo capire perché l'utilizzo di `__proto__` non è una buona pratica. Continuate a leggere per avere la risposta.

Non cambiate il `[[Prototype]]` ad oggetti esistenti se la velocità è importante

Tecnicamente, possiamo impostare/ottenere il `[[Prototype]]` in qualsiasi momento. Ma solitamente lo impostiamo in fase di creazione dell'oggetto e successivamente non lo modifichiamo più: `rabbit` eredita da `animal`, e questo non dovrebbe cambiare.

I motori JavaScript sono altamente ottimizzati per questo. Cambiare il prototype durante l'esecuzione, con `Object.setPrototypeOf` o `obj.__proto__ =` è un'operazione molto lenta, poiché vanifica le ottimizzazioni interne fatte per le operazioni di accesso alle proprietà. Quindi evitate questa pratica, a meno che non siate consci di ciò che state facendo, e la velocità di esecuzione non è un problema per voi.

"Very plain" objects

Come sappiamo, gli oggetti possono essere utilizzati come un array associativo per memorizzare coppie chiave/valore.

...Ma se proviamo a memorizzare chiavi *fornite dall'utente* (ad esempio, un dizionario con vocaboli forniti dall'utente), noteremo un piccolo bug: tutte le chiavi funzioneranno senza problemi, ad eccezione di `"__proto__"`.

Vediamo un esempio:

```
let obj = {};  
  
let key = prompt("What's the key?", "__proto__");  
obj[key] = "some value";
```

```
alert(obj[key]); // [object Object], non "some value"!
```

In questo esempio, se l'utente digita `__proto__`, l'assegnazione è ignorata!

Questo non dovrebbe sorprenderci. La proprietà `__proto__` è speciale: deve contenere un oggetto o `null`. Una stringa non può fungere da prototype.

Ma il nostro *intento* non è quello di implementare questo comportamento, giusto? Vogliamo semplicemente memorizzare una coppia chiave/valore, ma utilizzando come chiave il termine "`__proto__`" questo non viene memorizzato correttamente. Quindi, questo è un bug!

In questo caso le conseguenze non sono così terribili. Ma in altri casi potremmo assegnarli oggetti, andando a modificare il valore del prototype. Risultato: l'esecuzione fallirà in maniera imprevedibile.

Ancora peggio – solitamente gli sviluppatori non pensano affatto a questa eventualità. Questo lo rende un bug veramente difficile da trovare e può portare a diverse vulnerabilità, specialmente se il codice viene eseguito server-side.

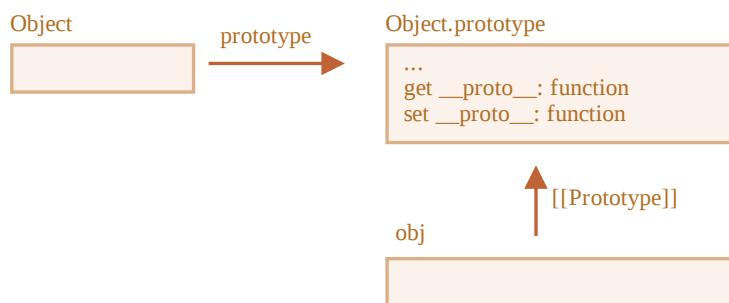
Questi comportamenti inaspettati accadono anche se proviamo ad assegnare la chiave `toString`, la quale è una funzione di default, e lo stesso vale per gli altri metodi integrati.

Come possiamo evitare questo problema?

Come prima cosa, possiamo semplicemente utilizzare `Map` per la memorizzazione dei valori al posto di un oggetto, in questo modo non avremo problemi.

Ma `Object` potrebbe esserci utile, perché i creatori del linguaggio hanno pensato a questo problema molto tempo fa.

`__proto__` non è una proprietà di un oggetto, ma una proprietà di accesso per `Object.prototype`:



Quindi, se `obj.__proto__` viene letta o impostata, il corrispondente getter/setter viene chiamato dal suo prototype, il quale legge/imposta il `[[Prototype]]`.

Come detto all'inizio di questa sezione: `__proto__` è un modo per accedere al `[[Prototype]]`, non è il `[[Prototype]]` stesso.

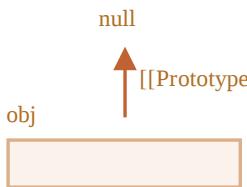
Ora, se il nostro scopo è quello di utilizzare un oggetto come array associativo, e vogliamo evitare questo tipo di problemi, possiamo farlo in questo modo:

```
let obj = Object.create(null);

let key = prompt("What's the key?", "__proto__");
obj[key] = "some value";
```

```
alert(obj[key]); // "some value"
```

`Object.create(null)` crea un oggetto vuoto senza un prototype (`[[Prototype]]` vale `null`):



Quindi, non si ha alcun getter/setter ereditato per `__proto__`. D'ora in poi verrà trattata come una comune proprietà; l'esempio visto sopra funzionerà senza problemi.

Questo tipo di oggetti vengono chiamati “very plain” (“molto semplici”) o “pure dictionary” (“dizionari puri”), poiché sono molto più semplici dei normali plain object `{ . . . }`.

Il lato negativo di questi oggetti è che mancano di tutti i metodi integrati, ad esempio `toString`:

```
let obj = Object.create(null);  
  
alert(obj); // Error (non esiste toString)
```

...Ma questo può andarci bene per gli array associativi.

Da notare che molti dei metodi relativi agli oggetti sono come `Object.something(...)`, ad esempio `Object.keys(obj)` – non sono contenuti all'interno del prototype, quindi continueranno a funzionare anche con questo tipo di oggetti:

```
let chineseDictionary = Object.create(null);  
chineseDictionary.hello = "你好";  
chineseDictionary.bye = "再见";  
  
alert(Object.keys(chineseDictionary)); // hello,bye
```

Riepilogo

I metodi moderni per impostare e leggere il prototype sono:

- `Object.create(proto, [descriptors])` – crea un oggetto vuoto utilizzando `proto` come `[[Prototype]]` (può essere anche `null`) e dei property descriptors (descrittori di proprietà).
- `Object.getPrototypeOf(obj)` – ritorna il `[[Prototype]]` di `obj` (equivale a `__proto__`).
- `Object.setPrototypeOf(obj, proto)` – imposta il `[[Prototype]]` di `obj` a `proto` (equivale a `__proto__`).

La proprietà integrata `__proto__`, utilizzata come getter/setter, non è sicura nel caso in cui volessimo inserire in un oggetto chiavi fornite dall'utente. Un utente potrebbe inserire

"__proto__" come chiave, che genererebbe un errore; potrebbe non avere gravi conseguenze, ma generalmente non è prevedibile.

Le alternative disponibili sono: usare `Object.create(null)` per creare un "very plain" object, senza `__proto__`, o in alternativa, utilizzare `Map`.

Inoltre, `Object.create` consente di creare una shallow-copy ('copia non profonda') di un oggetto, compresi i suoi property descriptors:

```
let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));
```

Abbiamo anche chiarito che `__proto__` è un getter/setter per `[[Prototype]]` e risiede in `Object.prototype`, proprio come gli altri metodi.

Possiamo creare un oggetto senza prototype utilizzando `Object.create(null)`. Questo tipo di oggetti vengono utilizzati come "puri dizionari", e non causano problemi nel caso in cui venga utilizzata "`__proto__`" come chiave.

Altri metodi:

- `Object.keys(obj)` ↪ / `Object.values(obj)` ↪ / `Object.entries(obj)` ↪ – ritorna un array di stringhe contenente le proprietà enumerable, i valori e le coppie chiave/valore.
- `Object.getOwnPropertySymbols(obj)` ↪ – ritorna un array con tutte le chiavi di tipo Symbol che appartengono all'oggetto.
- `Object.getOwnPropertyNames(obj)` ↪ – ritorna un array di stringhe con tutte le proprietà che appartengono all'oggetto.
- `Reflect.ownKeys(obj)` ↪ – ritorna un array con tutte le chiavi che appartengono all'oggetto.
- `obj.hasOwnProperty(key)` ↪ : ritorna `true` se `obj` possiede una sua chiave `key` (non ereditata).

Tutti i metodi che ritornano le proprietà di un oggetto (come `Object.keys` e le altre) – ritornano le proprietà "possedute". Se vogliamo ottenere anche le proprietà ereditate dobbiamo utilizzare un ciclo `for..in`.

✓ Esercizi

Aggiungi `toString` al dizionario

importanza: 5

Abbiamo un oggetto `dictionary`, creato come `Object.create(null)`, in cui memorizziamo coppie key/value.

Aggiungi un metodo `dictionary.toString()`, il quale dovrebbe ritornare un lista di chiavi separate da virgola. Il metodo `toString` non deve essere mostrato nei cicli `for..in`.

Dovrebbe funzionare così:

```
let dictionary = Object.create(null);  
  
// il tuo codice per aggiungere il metodo toString
```

```
// aggiungiamo dei valori
dictionary.apple = "Apple";
dictionary.__proto__ = "test"; // __proto__ è una proprietà comune in questo caso

// nel ciclo compaiono solo apple e __proto__
for(let key in dictionary) {
  alert(key); // "apple", poi "__proto__"
}

// la tua implementazione di toString in azione
alert(dictionary); // "apple,__proto__"
```

[Alla soluzione](#)

La differenza tra chiamate

importanza: 5

Creiamo un nuovo oggetto `rabbit`:

```
function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype.sayHi = function() {
  alert(this.name);
};

let rabbit = new Rabbit("Rabbit");
```

Queste chiamata fanno la stessa cosa o no?

```
rabbit.sayHi();
Rabbit.prototype.sayHi();
Object.getPrototypeOf(rabbit).sayHi();
rabbit.__proto__.sayHi();
```

[Alla soluzione](#)

Classi

Sintassi base delle classi

Nella programmazione orientata agli oggetti una classe è un costrutto di un linguaggio di programmazione usato come modello per creare oggetti. Il modello comprende attributi e metodi che saranno condivisi da tutti gli oggetti creati (istanze) a partire dalla classe. Un “oggetto” è, di fatto, l’istanza di una classe.

“ Wikipedia

In pratica, spesso abbiamo bisogno di creare più oggetti dello stesso tipo, come utenti, beni o altro.

Come già sappiamo dal capitolo [Costruttore, operatore "new", new function](#) ci può aiutare in questo.

Ma nel JavaScript moderno c'è un costrutto “class” più avanzato, che introduce nuove possibilità molto utili per la programmazione ad oggetti.

La sintassi di “class”

La sintassi base è:

```
class MyClass {  
    // metodi della classe  
    constructor() { ... }  
    method1() { ... }  
    method2() { ... }  
    method3() { ... }  
    ...  
}
```

`new MyClass()` creerà un nuovo oggetto con tutti i metodi presenti nella classe.

Il metodo `constructor()` viene chiamato automaticamente da `new`, dunque possiamo usarlo per inizializzare l'oggetto.

Per esempio:

```
class User {  
  
    constructor(name) {  
        this.name = name;  
    }  
  
    sayHi() {  
        alert(this.name);  
    }  
  
}  
  
// Utilizzo:  
let user = new User("John");  
user.sayHi();
```

Quando viene chiamato `new User("John")`:

1. Viene creato un nuovo oggetto;
2. Il metodo `constructor()` viene richiamato e assegna a `this.name` l'argomento dato.

...Ora possiamo chiamare i metodi, per esempio `user.sayHi`.

Niente virgole tra i metodi

Un errore comune per i principianti è separare i metodi con delle virgole, portando ad un syntax error.

La notazione delle classi non va confusa con la notazione letterale per gli oggetti. In una classe non sono richieste virgole.

Cos'è una classe?

Dunque, cos'è esattamente una `class`? A differenza di ciò che si potrebbe pensare, non si tratta di un concetto completamente nuovo.

Vediamo quindi cos'è effettivamente una classe. Questo ci aiuterà a comprendere aspetti più complessi.

In JavaScript, una classe è una specie di funzione.

Osserva:

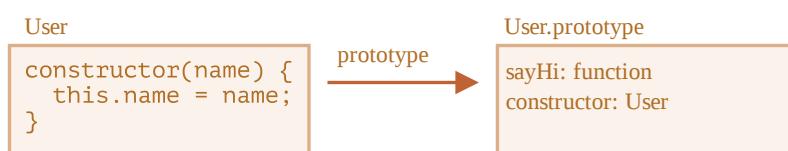
```
class User {  
  constructor(name) { this.name = name; }  
  sayHi() { alert(this.name); }  
}  
  
// prova: User è una funzione  
alert(typeof User); // function
```

Il costrutto `class User {...}` dunque:

1. Crea una funzione chiamata `User`, che diventa il risultato della dichiarazione della classe. Le istruzioni della funzione provengono dal metodo `constructor` (considerato vuoto se non presente);
2. Salva tutti i metodi (come `sayHi`) all'interno di `User.prototype`.

Quando richiameremo da un oggetto un metodo, questo verrà preso dal prototipo (`prototype`), come descritto nel capitolo [F.prototype](#). Dunque un oggetto `new User` ha accesso ai metodi della classe.

Possiamo rappresentare il risultato della dichiarazione di `class User` come:



Il codice seguente ti permetterà di analizzarlo:

```
class User {  
  constructor(name) { this.name = name; }  
  sayHi() { alert(this.name); }  
}
```

```
// una classe è una funzione
alert(typeof User); // function

// ...o, più precisamente, il costruttore
alert(User === User.prototype.constructor); // true

// I metodi sono in User.prototype:
alert(User.prototype.sayHi); // il codice del metodo sayHi

// ci sono due funzioni all'interno del prototipo
alert(Object.getOwnPropertyNames(User.prototype)); // constructor, sayHi
```

Non solo una semplificazione (syntax sugar)

Talvolta si pensa che `class` in JavaScript sia solo “syntax sugar” (una sintassi creata per semplificare la lettura, ma che non apporta nulla di nuovo), dato che potremmo potremmo dichiarare la stessa cosa senza utilizzare la parola chiave `class`:

```
// la classe User usando solo funzioni

// 1. Costruttore
function User(name) {
  this.name = name;
}

// tutte le funzioni hanno un costruttore predefinito (di default)
// dunque non va creato

// 2. Aggiungiamo un metodo al prototipo
User.prototype.sayHi = function() {
  alert(this.name);
};

// Utilizzo:
let user = new User("John");
user.sayHi();
```

Il risultato di questo codice è circa lo stesso. È quindi logico pensare che `class` sia solo una semplificazione sintattica (syntax sugar).

Ci sono però delle importanti differenze.

1. Una funzione creata attraverso `class` viene etichettata dalla speciale proprietà interna `[[IsClassConstructor]]`: `true`. Quindi non è esattamente uguale che crearla manualmente.

A differenza di una normale funzione, il costruttore di una classe può essere richiamato solo attraverso la parola chiave `new`:

```
class User {
  constructor() {}
}

alert(typeof User); // funzione
User(); // Errore: Il costruttore della classe può essere richiamato solo attraverso 'new'
```

Inoltre, nella maggior parte dei motori JavaScript il costruttore comincia con “class”

```
class User {  
  constructor() {}  
}  
  
alert(User); // class User { ... }
```

Ci sono altre differenze, che scopriremo più avanti.

2. I metodi delle classi non sono numerabili. La definizione di una classe imposta il flag `enumerable` a `false` per tutti i metodi all'interno di `"prototype"`.

Questo è un bene, dato che non vogliamo visualizzare i metodi quando utilizziamo un ciclo `for..in` per visualizzare un oggetto.

3. Il contenuto di una classe viene sempre eseguito in `strict`.

Oltre a queste, la sintassi `class` apporta altre caratteristiche, che esploreremo più avanti.

L'espressione `class`

Come le funzioni, le classi possono essere definite all'interno di un'altra espressione, passata come parametro, essere ritornata (returned), assegnata (assigned) ecc.

Qui c'è un piccolo esempio:

```
let User = class {  
  sayHi() {  
    alert("Hello");  
  }  
};
```

In maniera simile alle funzione nominate (Named Function Expression), le classi possono avere o meno un nome.

Se una classe ha un nome, esso è visibile solo all'interno della classe:

```
// "Named Class Expression"  
// (la classe non ha un nome)  
let User = class MyClass {  
  sayHi() {  
    alert(MyClass); // MyClass è visibile solo all'interno della classe  
  }  
};  
  
new User().sayHi(); // funziona, restituisce la definizione di MyClass  
  
alert(MyClass); // errore, MyClass non è visibile al di fuori della classe
```

Possiamo anche creare delle classi “on-demand”:

```

function makeClass(phrase) {
  // dichiara una classe e la restituisce
  return class {
    sayHi() {
      alert(phrase);
    }
  };
}

// Crea una nuova classe
let User = makeClass("Hello");

new User().sayHi(); // Hello

```

Getters/setters e altre scorciatoie

Così come negli oggetti letterali (literal objects), le classi possono includere getters/setters, generatori, proprietà eccetera.

L'esempio seguente implementa `user.name` attraverso `get/set`:

```

class User {

  constructor(name) {
    // invoca il setter
    this.name = name;
  }

  get name() {
    return this._name;
  }

  set name(value) {
    if (value.length < 4) {
      alert("Name is too short.");
      return;
    }
    this._name = value;
  }
}

let user = new User("John");
alert(user.name); // John

user = new User(""); // Nome troppo corto.

```

La dichiarazione della classe crea i getter e i setter all'interno di `User.prototype`:

Computed names [...]

A seguire un esempio con le proprietà:

```

function f() { return "sayHi"; }

```

```
class User {
  [f()]() {
    alert("Hello");
  }
}

new User().sayHi();
```

Per creare un metodo generatore è sufficiente aggiungere `*` prima del nome della funzione.

Proprietà di una classe

 **I vecchi browser potrebbero non supportarle**

Le proprietà di una classe dichiarata in questo modo sono una novità del linguaggio.

Negli esempi riportati sopra, la classe `User` conteneva solo dei metodi. Aggiungiamo una proprietà:

```
class User {
  name = "Anonymous";

  sayHi() {
    alert(`Hello, ${this.name}!`);
  }
}

new User().sayHi(); // Hello, John!
```

Quindi scriviamo semplicemente `" = "` nella dichiarazione.

La differenza importante dei campi di una classe è che vengono impostati sull'oggetto individuale e non su `User.prototype`:

```
class User {
  name = "John";
}

let user = new User();
alert(user.name); // John
alert(User.prototype.name); // undefined
```

Possiamo anche assegnare valori utilizzando espressioni più complesse e chiamate a funzioni:

```
class User {
  name = prompt("Name, please?", "John");
}

let user = new User();
alert(user.name); // John
```

Creazione di metodi vincolati a campi di classe

Come dimostrato nel capitolo [Function binding](#), le funzioni in JavaScript hanno un `this` dinamico che dipende dal contesto della chiamata.

Quindi, se un metodo di un object viene passato e chiamato in un altro contesto, `this` non sarà più un riferimento al suo object.

Per esempio, questo codice mostrerà `undefined`:

```
class Button {  
  constructor(value) {  
    this.value = value;  
  }  
  
  click() {  
    alert(this.value);  
  }  
}  
  
let button = new Button("hello");  
  
setTimeout(button.click, 1000); // undefined
```

Il problema viene chiamato "perdita del `this`".

Ci sono due differenti approcci per affrontare questo problema, come discusso nel capitolo [Function binding](#):

1. Passare una funzione contenitore, come `setTimeout(() => button.click(), 1000)`.
2. Associare il metodo all'oggetto, e.g. nel costruttore.

I campi di una classe forniscono un'altra sintassi molto più elegante:

```
class Button {  
  constructor(value) {  
    this.value = value;  
  }  
  
  click = () => {  
    alert(this.value);  
  }  
}  
  
let button = new Button("hello");  
  
setTimeout(button.click, 1000); // hello
```

Il campo della classe `click = () => { . . . }` viene creato per ogni oggetto, abbiamo quindi una funzione diversa per ogni `Button`, con il riferimento `this` che punta all'oggetto. Possiamo passare `button.click` ovunque, e il valore di `this` sarà sempre quello corretto.

Questo è particolarmente utile in ambiente browser, per gli event listeners (ascoltatori di eventi).

Riepilogo

Il seguente esempio riporta la sintassi base di una classe:

```
class MyClass {  
    prop = value; // proprietà  
  
    constructor(...) { // costruttore  
        // ...  
    }  
  
    method(...) {} // metodo  
  
    get something(...) {} // metodo getter  
    set something(...) {} // metodo setter  
  
    [Symbol.iterator]() {} // metodo creato con un vettore relazionale  
    // ...  
}
```

`MyClass` è tecnicamente una funzione (che corrisponde a `constructor`), mentre i metodi vengono scritti in `MyClass.prototype`.

Nei prossimi capitoli impareremo altri dettagli riguardo alle classi, come l'ereditarietà.

✓ Esercizi

Rewrite to class

importanza: 5

The `Clock` class (see the sandbox) is written in functional style. Rewrite it in the “class” syntax.

P.S. The clock ticks in the console, open it to see.

[Apri una sandbox per l'esercizio.](#) ↗

[Alla soluzione](#)

Ereditarietà delle classi

L'ereditarietà è una caratteristica che permette ad una classe di estendere le proprietà di altre classi.

La parola chiave “extends”

Ipotizziamo di avere una classe `Animal`:

```
class Animal {  
    constructor(name) {  
        this.speed = 0;  
        this.name = name;  
    }  
}
```

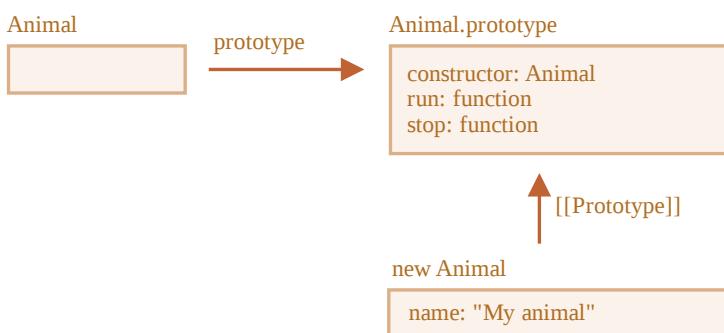
```

run(speed) {
  this.speed = speed;
  alert(`#${this.name} runs with speed ${this.speed}.`);
}
stop() {
  this.speed = 0;
  alert(`#${this.name} stands still.`);
}
}

let animal = new Animal("My animal");

```

Qui vediamo come rappresentare l'oggetto `animal` e la classe `Animal` graficamente:



...Potremmo voler creare un'altra `class Rabbit`.

Poiché i conigli sono animali, la classe `Rabbit` dovrebbe essere basata su `Animal`, avendo accesso a tutti i metodi di `Animal`, in questo modo `Rabbit` può assumere tutti i comportamenti di base di un `Animal`.

La sintassi utilizzate per estendere un'altra classe è: `class Child extends Parent`.

Creiamo `class Rabbit` che eredita da `Animal`:

```

class Rabbit extends Animal {
  hide() {
    alert(`#${this.name} hides!`);
  }
}

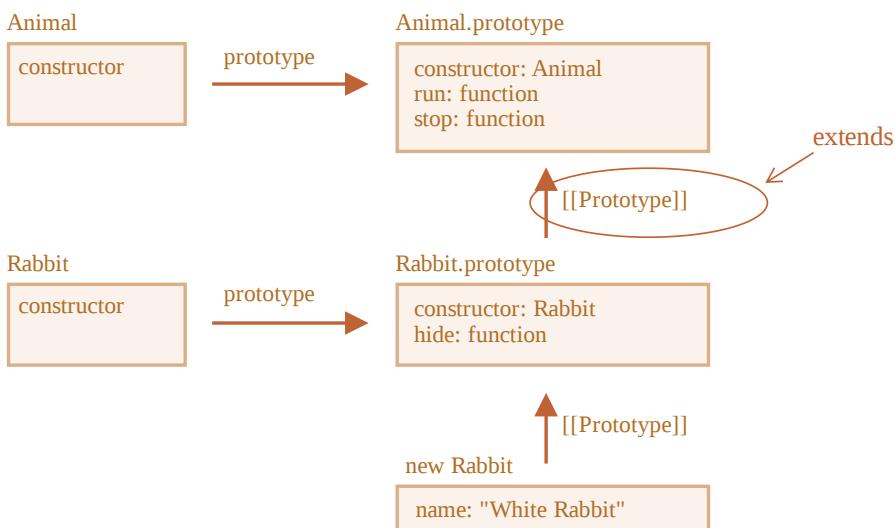
let rabbit = new Rabbit("White Rabbit");

rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.hide(); // White Rabbit hides!

```

L'oggetto della classe `Rabbit` ha accesso sia ai metodi di `Rabbit` (ad esempio `rabbit.hide()`) che a quelli di `Animal` (`rabbit.run()`).

Internamente, `extends` aggiunge da `Rabbit.prototype` un riferimento `[[Prototype]]` a `Animal.prototype`:



Ad esempio, per trovare il metodo `rabbit.run`, il motore JavaScript controlla (dal basso verso l'alto in figura):

1. L'oggetto `rabbit` (non possiede `run`).
2. Il suo prototype, che è `Rabbit.prototype` (possiede `hide`, ma non `run`).
3. Il suo prototype, che è (a causa di `extends`) `Animal.prototype`, che possiede il metodo `run`.

Come ricordiamo dal capitolo [Native prototypes](#), JavaScript stesso usa l'ereditarietà per prototipi per gli oggetti integrati. E.g. `Date.prototype.[[Prototype]]` è `Object.prototype`. Questo è il motivo per cui le date hanno accesso ai metodi generici di un oggetto.

i Qualsiasi espressione è ammessa dopo `extend`

Usare la parola chiave `class` permette di specificare non solo una classe, ma anche un'espressione dopo la parola `extends`.

Per esempio, una chiamata ad una funzione che genera la classe padre:

```
``js run function f(phrase) { return class { sayHi() { alert(phrase); } }; }
! class User extends f("Hello") {} !/
new User().sayHi(); // Hello
```

In questo codice la `class User` eredita dal risultato della funzione `f("Hello")`.

Questa particolarità può tornare utile nella programmazione avanzata, quando abbiamo bisogno di generare delle classi padre a seconda di vari parametri. ``

Sovrascrivere un metodo

Proseguiamo ora e vediamo come sovrascrivere un metodo. Di base, tutti i metodi che non vengono definiti in `class Rabbit` vengono presi “così come sono” da `class Animal`.

Ma se specifichiamo un metodo in `Rabbit`, come `stop()` allora verrà utilizzato questo:

```
class Rabbit extends Animal {
```

```

stop() {
    // ...questo verrà utilizzato per rabbit.stop()
    // piuttosto di stop() dal padre, class Animal
}

```

...Normalmente però non vogliamo rimpiazzare completamente il metodo ereditato, ma piuttosto costruire su di esso, modificarlo leggermente o estendere le sue funzionalità. Nel nostro metodo compiamo delle azioni, ma ad un certo punto richiamiamo il metodo ereditato.

Le classi forniscono la parola chiave "super" per questo scopo.

- `super.method(...)` per richiamare un metodo dal padre;
- `super(...)` per richiamare il costruttore del padre (valido solo all'interno del nostro costruttore).

Per esempio, facciamo sì che il nostro coniglio si nasconde automaticamente quando si ferma:

```

class Animal {

    constructor(name) {
        this.speed = 0;
        this.name = name;
    }

    run(speed) {
        this.speed = speed;
        alert(`#${this.name} runs with speed ${this.speed}.`);
    }

    stop() {
        this.speed = 0;
        alert(`#${this.name} stands still.`);
    }
}

class Rabbit extends Animal {
    hide() {
        alert(`#${this.name} hides!`);
    }

    stop() {
        super.stop(); // richiama il metodo stop() dal padre
        this.hide(); // and then hide
    }
}

let rabbit = new Rabbit("White Rabbit");

rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.stop(); // White Rabbit stands still. White Rabbit hides!

```

Ora `Rabbit` contiene il metodo `stop`, che richiama al suo interno il metodo `super.stop()`.

i Le funzioni a freccia (Arrow functions) non hanno `super`

Come accennato nel capitolo [Arrow functions rivisitate](#), all'interno delle funzioni a freccia (arrow functions) non si può utilizzare la parola `super`

Se acceduto, esso viene preso dalla funzione esterna. Per esempio:

```
class Rabbit extends Animal {  
    stop() {  
        setTimeout(() => super.stop(), 1000); // richiama il metodo stop dal padre dopo 1 secondo  
    }  
}
```

Il `super` nella funzione a freccia (arrow function) è lo stesso di `stop()`, quindi funziona come dovrebbe. Se specificassimo una funzione "regolare" (regular) otterremmo un errore:

```
// Unexpected super  
setTimeout(function() { super.stop() }, 1000);
```

Sovrascrivere il costruttore

Sovrascrivere un costruttore è leggermente più complicato.

Finora, `Rabbit` non ha avuto il suo metodo `constructor`.

Secondo le [specifiche ↗](#), se una classe ne estende un'altra e non ha un suo metodo `constructor` viene generato il seguente `constructor` "vuoto":

```
class Rabbit extends Animal {  
    // generato per classi figlie senza un costruttore proprio  
    constructor(...args) {  
        super(...args);  
    }  
}
```

Come possiamo vedere, esso richiama il `constructor` del padre, passandogli tutti gli argomenti. Questo accade se non creiamo un costruttore ad hoc.

Aggiungiamo quindi un `constructor` personalizzato per `Rabbit`, che specificherà, oltre al `name`, anche la proprietà `earLength`:

```
class Animal {  
    constructor(name) {  
        this.speed = 0;  
        this.name = name;  
    }  
    // ...  
}  
  
class Rabbit extends Animal {
```

```

constructor(name, earLength) {
  this.speed = 0;
  this.name = name;
  this.earLength = earLength;
}

// ...

}

// Non funziona!
let rabbit = new Rabbit("White Rabbit", 10); // Error: this is not defined. (Errore: "this" non è definito)

```

Ops! Abbiamo ricevuto un errore. Ora non possiamo creare conigli (rabbits). Cosa è andato storto?

La risposta breve è:

- **I costruttori nelle classi che ereditano devono chiamare `super(...)`, e bisogna farlo (!) prima di utilizzare `this`.**

...Ma perché? Cosa sta succedendo? In effetti, questa richiesta sembra un po' strana.

Ovviamente una spiegazione c'è. Addentriamoci nei dettagli, così da capire cosa effettivamente succede.

In JavaScript vi è una netta distinzione tra il “metodo costruttore di una classe figlia” e tutte le altre. In una classe figlia, il costruttore viene etichettato con una proprietà interna speciale: `[[ConstructorKind]] : "derived"`.

La differenza è:

- Quando viene eseguito un costruttore normale, esso crea un oggetto vuoto chiamato `this` e continua a lavorare su quello. Questo non avviene quando il costruttore di una classe figlia viene eseguito, dato che si aspetta che il costruttore del padre lo faccia per lui.

Se stiamo creando il costruttore di un figlio dobbiamo per forza richiamare `super`, altrimenti l'oggetto referenziato da `this` non verrebbe creato. E riceveremmo un errore.

Per far funzionare `Rabbit` dobbiamo richiamare `super()` prima di usare `this`:

```

class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

  // ...
}

class Rabbit extends Animal {

  constructor(name, earLength) {
    super(name);
    this.earLength = earLength;
  }
}

```

```
// ...
}

// finalmente
let rabbit = new Rabbit("White Rabbit", 10);
alert(rabbit.name); // White Rabbit
alert(rabbit.earLength); // 10
```

Sovrascrivere i campi di una classe

Nota avanzata

Questa nota assume che voi abbiate una certa esperienza con le classi, anche in altri linguaggi di programmazione.

Fornisce una spiegazione più dettagliata del linguaggio e ne illustra il comportamento che potrebbe essere fonte di errori (anche se molto rari).

Se trovate questa sezione troppo difficile da capire, saltatela pure, continuate a leggere, e rileggetela in un secondo momento.

In una sotto-classe possiamo estendere non solo i metodi, ma anche i campi di classe.

Anche se, si verifica un comportamento strano quando proviamo ad accedere ad un campo sovrascritto nel costruttore genitore, piuttosto differente da altri linguaggi di programmazione.

Consideriamo questi esempi:

```
class Animal {
  name = 'animal';

  constructor() {
    alert(this.name); // (*)
  }
}

class Rabbit extends Animal {
  name = 'rabbit';
}

new Animal(); // animal
new Rabbit(); // animal
```

Qui, la classe `Rabbit` estende `Animal` e sovrascrive il campo `name` con il suo valore.

Non c'è alcun costruttore in `Rabbit`, quindi viene invocato quello di `Animal`.

E' interessante notare che in entrambi i casi: `new Animal()` e `new Rabbit()`, l'istruzione di `alert` nella riga (*) mostra `animal`.

In altre parole, il costruttore genitore utilizza sempre i suoi campi dati, non quelli sovrascritti.

Cosa c'è di strano in questo?

Se non è ancora chiaro, confrontiamo con i metodi.

Qui abbiamo lo stesso codice, ma invece del campo `this.name` invochiamo il metodo `this.showName()`:

```
class Animal {  
    showName() { // invece di this.name = 'animal'  
        alert('animal');  
    }  
  
    constructor() {  
        this.showName(); // invece di alert(this.name);  
    }  
}  
  
class Rabbit extends Animal {  
    showName() {  
        alert('rabbit');  
    }  
}  
  
new Animal(); // animal  
new Rabbit(); // rabbit
```

Notiamo che l'output è differente.

E questo è quello che ci aspetteremmo. Quando il costruttore genitore viene invocato da una classe derivata, utilizzate i metodi sovrascritti.

...Ma per i campi dati non è così. Come già detto, il costruttore genitore utilizza sempre i suoi campi dati.

Perché c'è questa differenza?

Il motivo sta nell'ordine di inizializzazione dei campi dati. I campi dati di una classe vengono inizializzati:

- Prima del costruttore per la classe base,
- Subito dopo `super()` per le classi derivate.

Nel nostro caso, `Rabbit` è la classe derivata. Non c'è alcun `constructor()` al suo interno. Come detto precedentemente, questo equivale ad avere un costruttore vuoto con la sola chiamata a `super(...args)`.

Quindi, `new Rabbit()` invoca `super()`, che esegue il costruttore genitore, e (per le regole che segue la classe derivata) solamente dopo vengono inizializzati i suoi campi dati. Al momento dell'esecuzione del costruttore genitore, non esiste alcun capo dato in `Rabbit`, questo è il motivo per cui vengono utilizzati i campi dati di `Animal`.

Abbiamo quindi una sottile differenza di trattamento tra i campi dati ed i metodi in JavaScript.

Fortunatamente, questo comportamento si verifica solamente se un campo dati va a sovrascrivere quelli della classe genitore. Potrebbe essere difficile da capire come comportamento, per questo lo abbiamo spiegato.

Se dovesse verificarsi questo problema, si possono utilizzare i metodi invece dei campi dati.

Super: internamente, [[HomeObject]]

Informazioni avanzate

Se state leggendo il tutorial per la prima volta – questa sezione può essere saltata.

Qui spiegheremo i meccanismi interni che stanno dietro l'ereditarietà e `super`.

Andiamo un pò più a fondo del metodo `super`. Scopriremo alcune cose interessanti a riguardo.

Beh, proviamo a chiederci, come può funzionare? Quando un metodo viene eseguito, il suo oggetto di appartenenza viene indicato con `this`. Se richiamiamo `super.method()`, dunque, esso dovrà recuperare il metodo dal prototipo dell'oggetto corrente.

Questa attività può sembrare semplice, ma non lo è. Il motore (engine) conosce l'oggetto `this`, quindi potrebbe ottenere il metodo dalla classe padre attraverso `this.__proto__.method`. Sfortunatamente, una soluzione così “naïf” non funzionerà.

Dimostriamo il problema, usando per semplicità degli oggetti piani (plain objects).

Nell'esempio sottostante, `rabbit.__proto__ = animal`. Ora proviamo: in `rabbit.eat()` richiamiamo `animal.eat()` attraverso `this.__proto__`:

```
let animal = {
  name: "Animal",
  eat() {
    alert(`#${this.name} eats.`);
  }
};

let rabbit = {
  __proto__: animal,
  name: "Rabbit",
  eat() {
    // super.eat() dovrebbe funzionare presumibilmente così
    this.__proto__.eat.call(this); // (*)
  }
};

rabbit.eat(); // Rabbit eats.
```

Alla linea `(*)` prendiamo `eat` dal prototipo (`animal`) e lo richiamiamo all'interno dell'oggetto. Nota che `.call(this)` è importante, dato che `this.__proto__.eat()` richiamerebbe il metodo `eat` nel contesto della classe padre, non nella classe figlio.

Nell'esempio precedente in effetti il metodo funzionava a dovere: abbiamo ricevuto l'`alert` corretto.

Ora proviamo ad aggiungere un altro oggetto. Vedremo cosa non va:

```
let animal = {
  name: "Animal",
  eat() {
    alert(`#${this.name} eats.`);
  }
};

let rabbit = {
```

```

__proto__: animal,
eat() {
  // ...salta in giro come un coniglio e richiama il metodo dalla classe padre (animal)
  this.__proto__.eat.call(this); // (*)
}
};

let longEar = {
  __proto__: rabbit,
  eat() {
    // ...fa qualcosa con longEar e richiama il metodo dalla classe padre (rabbit)
    this.__proto__.eat.call(this); // (**)
  }
};

longEar.eat(); // Error: Maximum call stack size exceeded (Errore: limite massimo di chiamate al

```

Il codice non funziona più! Possiamo vedere l'errore provando a richiamare `longEar.eat()`.

Potrebbe non essere così scontato, ma se tracciamo la chiamata di `longEar.eat()` possiamo capire perché ciò accade. Nelle linee `(*)` e `(**)` il valore di `this` è l'oggetto corrente (`longEar`). Questo è fondamentale: tutti i metodi di un oggetto ricevono l'oggetto corrente come `this`, non attraverso un prototipo o simili.

Quindi, sia nella linea `(+)` che nella linea `(**)` il valore di `this.__proto__` è esattamente lo stesso: `rabbit`. Entrambi richiamano `rabbit.eat` senza salire la catena, generando un ciclo (loop) infinito.

Questa immagine rappresenta ciò che accade:

```

let rabbit = {
  __proto__: animal,
  eat(){
    this.__proto__.eat.call(this); (*)
  }
};

let longEar = {
  __proto__: rabbit,
  eat() {
    this.__proto__.eat.call(this); (**)
  }
};

```

1. Dentro a `longEar.eat()`, la linea `(**)` richiama `rabbit.eat` assieme a `this=longEar`.

```

// dentro a longEar.eat() abbiamo this = longEar
this.__proto__.eat.call(this) // (**)
// diventa
longEar.__proto__.eat.call(this)
// che è uguale a
rabbit.eat.call(this);
```

```

2. Poi nella linea `(*)` di `rabbit.eat` vorremo passare la chiamata ancora più in alto nella catena.

```
```js
```

```
// dentro a rabbit.eat() abbiamo ancora this = longEar
this.__proto__.eat.call(this) // (*)
// diventa
longEar.__proto__.eat.call(this)
// oppure (nuovamente)
rabbit.eat.call(this);
```

```

3. ...Dunque `rabbit.eat` richiama sé stesso in un ciclo (loop) infinito, perché non può più salire.

Il problema non può essere risolto utilizzando solo `this`.

```
`[[HomeObject]]`
```

Per dare una soluzione, JavaScript ha un'altra speciale proprietà interna per le funzioni: `[[HomeObject]]`.

Quando una funzione appartiene ad una classe o ad un metodo, la sua proprietà `[[HomeObject]]` dà

Quindi viene utilizzata da `super` per capire il prototipo del padre e i suoi metodi.

Vediamo come funziona:

```
```js run
let animal = {
name: "Animal",
eat() {           // animal.eat.[[HomeObject]] == animal
  alert(`#${this.name} eats.`);
}
};

let rabbit = {
__proto__: animal,
name: "Rabbit",
eat() {           // rabbit.eat.[[HomeObject]] == rabbit
  super.eat();
}
};

let longEar = {
__proto__: rabbit,
name: "Long Ear",
eat() {           // longEar.eat.[[HomeObject]] == longEar
  super.eat();
}
};

// funziona correttamente
longEar.eat();  // Long Ear eats.
```

Funziona come dovrebbe, grazie alle meccaniche di `[[HomeObject]]`. Un metodo, per esempio `longEar.eat`, conosce il suo `[[HomeObject]]` e prende il metodo della classe padre da quel prototipo, senza utilizzare `this`.

I metodi non sono “liberi”

Come abbiamo già visto, generalmente le funzioni sono libere, ovvero non sono legate ad un oggetto in JavaScript, così da poter essere copiate tra gli oggetti ed essere richiamate con un altro `this`.

L'esistenza di `[[HomeObject]]` viola questo principio, perché i metodi "ricordano" i loro oggetti. `[[HomeObject]]` non può essere modificato, quindi questo legame dura per sempre.

L'unico posto in cui `[[HomeObject]]` viene utilizzato è in `super`. Quindi, se un metodo non utilizza `super` è ancora libero e copiabile. Ma con `super` le cose potrebbero andar male.

Qui di seguito è rappresentato un utilizzo sbagliato di `super`:

```
let animal = {
  sayHi() {
    alert(`I'm an animal`);
  }
};

// rabbit inherits from animal
let rabbit = {
  __proto__: animal,
  sayHi() {
    super.sayHi();
  }
};

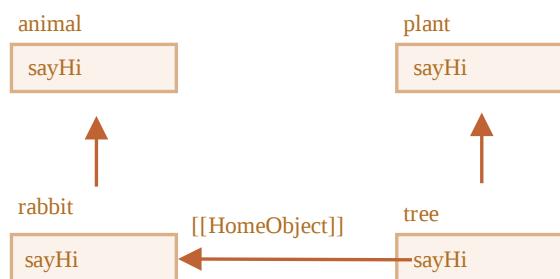
let plant = {
  sayHi() {
    alert("I'm a plant");
  }
};

// tree inherits from plant
let tree = {
  __proto__: plant,
  sayHi: rabbit.sayHi
};

tree.sayHi(); // I'm an animal (?!?)
```

Una chiamata a `tree.sayHi()` mostra "I'm an animal". Completamente sbagliato.

- Quindi il suo `[[HomeObject]]` è `rabbit`, dato che è stato creato in `rabbit`. Non c'è modo di cambiare `[[HomeObject]]`;
- Il codice di `tree.sayHi()` contiene `super.sayHi()`, che va fino a `rabbit` e prende il metodo da `animal`.



Metodi, non proprietà di una funzione

`[[HomeObject]]` viene definito per metodi appartenenti a classi e ad oggetti piani (plain objects), ma per gli oggetti i metodi vanno definiti come `method()`, non `"method": "function()"`.

La differenza potrebbe non essere rilevante per noi, ma lo è per JavaScript.

Nel prossimo esempio, viene utilizzata una sintassi errata (non-method syntax) per fare un confronto. La proprietà `[[HomeObject]]` non viene impostata e l'ereditarietà non funziona:

```
let animal = {
  eat: function() { // dovrebbe corrispondere a eat(){...}
    // ...
  }
};

let rabbit = {
  __proto__: animal,
  eat: function() {
    super.eat();
  }
};

rabbit.eat(); // Errore nella chiamata a super (dato che [[HomeObject] non esiste])
```

Riepilogo

1. Per estendere una classe: `class Child extends Parent :`

- Questo significa che `Child.prototype.__proto__` diventerà `Parent.prototype`, quindi i metodi vengono ereditati.

2. Quando sovrascriviamo un costruttore:

- Dobbiamo richiamare il costruttore del padre attraverso `super()` nel costruttore di `Child` prima di utilizzare `this`.

3. Quando sovrascriviamo un metodo:

- Possiamo usare `super.method()` in un metodo di `Child` per richiamare il metodo da `Parent`.

4. Meccanismi interni:

- I metodi tengono traccia del loro oggetto o della loro classe nella proprietà `[[HomeObject]]`, così da poter utilizzare `super` per accedere ai metodi della classe padre.
- Non è quindi sicuro copiare un metodo in un altro oggetto attraverso `super`.

Inoltre:

- Le funzioni a freccia (arrow functions) non hanno un loro `this` o `super`, dunque si adattano al contesto in cui si trovano.

✓ Esercizi

Error creating an instance

importanza: 5

Here's the code with `Rabbit` extending `Animal`.

Unfortunately, `Rabbit` objects can't be created. What's wrong? Fix it.

```
class Animal {  
  
  constructor(name) {  
    this.name = name;  
  }  
  
}  
  
class Rabbit extends Animal {  
  constructor(name) {  
    this.name = name;  
    this.created = Date.now();  
  }  
}  
  
let rabbit = new Rabbit("White Rabbit"); // Error: this is not defined  
alert(rabbit.name);
```

[Alla soluzione](#)

Extended clock

importanza: 5

We've got a `Clock` class. As of now, it prints the time every second.

```
class Clock {  
  constructor({ template }) {  
    this.template = template;  
  }  
  
  render() {  
    let date = new Date();  
  
    let hours = date.getHours();  
    if (hours < 10) hours = '0' + hours;  
  
    let mins = date.getMinutes();  
    if (mins < 10) mins = '0' + mins;  
  
    let secs = date.getSeconds();  
    if (secs < 10) secs = '0' + secs;  
  
    let output = this.template  
      .replace('h', hours)  
      .replace('m', mins)  
      .replace('s', secs);  
  
    console.log(output);  
  }  
}
```

```

    stop() {
      clearInterval(this.timer);
    }

    start() {
      this.render();
      this.timer = setInterval(() => this.render(), 1000);
    }
}

```

Create a new class `ExtendedClock` that inherits from `Clock` and adds the parameter `precision` – the number of `ms` between “ticks”. Should be `1000` (1 second) by default.

- Your code should be in the file `extended-clock.js`
- Don’t modify the original `clock.js`. Extend it.

[Apri una sandbox per l'esercizio.](#) ↗

[Alla soluzione](#)

Proprietà e metodi statici

Possiamo anche assegnare metodi alle classi stesse, non solamente al loro “prototype”. Questi metodi sono detti *statici*.

All'interno della classe, questi vengono preceduti dalla keyword `static`, come possiamo vedere nell'esempio:

```

class User {
  static staticMethod() {
    alert(this === User);
  }
}

User.staticMethod(); // true

```

Questo avrà lo stesso effetto di assegnarla direttamente come proprietà:

```

class User { }

User.staticMethod = function() {
  alert(this === User);
};

User.staticMethod(); // true

```

Il valore di `this` nella chiamata `User.staticMethod()` è rappresentato dal costruttore della classe `User` (la regola dell’ “oggetto prima del punto”).

Solitamente, i metodi statici vengono utilizzati per rappresentare funzioni che appartengono alla classe, ma non ad un oggetto in particolare.

Ad esempio, potremmo avere degli oggetti di tipo `Article` e necessitare di una funzione per confrontarli. Una soluzione naturale sarebbe quella di aggiungere il metodo `Article.compare`, come nell'esempio:

```
class Article {  
    constructor(title, date) {  
        this.title = title;  
        this.date = date;  
    }  
  
    static compare(articleA, articleB) {  
        return articleA.date - articleB.date;  
    }  
}  
  
// usage  
let articles = [  
    new Article("HTML", new Date(2019, 1, 1)),  
    new Article("CSS", new Date(2019, 0, 1)),  
    new Article("JavaScript", new Date(2019, 11, 1))  
];  
  
articles.sort(Article.compare);  
  
alert(articles[0].title); // CSS
```

Qui `Article.compare` sta “al di sopra” degli articoli, poiché ha lo scopo di confrontarli. Non è un metodo di un articolo, ma piuttosto dell’intera classe.

Un altro esempio comune è quello del “factory method” (un particolare design pattern). Immaginiamo di avere bisogno di diverse modalità di creazione di un articolo:

1. Creazione con i parametri forniti (`title`, `date` etc).
2. Creazione di un articolo vuoto con la data di oggi.
3. ...o qualsiasi altra modalità.

Il primo metodo può essere implementato tramite il costruttore. Mentre per il secondo, possiamo creare un metodo statico appartenente alla classe.

Come `Article.createTodays()` nell'esempio:

```
class Article {  
    constructor(title, date) {  
        this.title = title;  
        this.date = date;  
    }  
  
    static createTodays() {  
        // ricorda, this = Article  
        return new this("Today's digest", new Date());  
    }  
}  
  
let article = Article.createTodays();
```

```
alert( article.title ); // Today's digest
```

Ora, ogni volta in cui avremo bisogno di creare un “today’s digest”, possiamo invocare `Article.createTodays()`. Ripetiamolo nuovamente, questo non è un metodo per uno specifico articolo, ma piuttosto un metodo dell’intera classe.

I metodi statici vengono utilizzati anche nelle classi database-related (relative a database), per poter cercare/salvare/rimuovere elementi dal database, come nell’esempio:

```
// assumiamo che Article sia una classe speciale per la gestione degli articoli  
// metodo statico per la rimozione di un articolo:  
Article.remove({id: 12345});
```

Proprietà statiche

⚠ Aggiunta di recente

Questa funzionalità è stata aggiunta di recente al linguaggio. Gli esempi funzionano nei Chrome recenti.

E’ anche possibile definire proprietà statiche, queste sono molto simili alle proprietà della classe, ma sono precedute dalla keyword `static`:

```
class Article {  
  static publisher = "Ilya Kantor";  
}  
  
alert( Article.publisher ); // Ilya Kantor
```

Lo stesso che si otterrebbe con un assegnazione diretta ad `Article`:

```
Article.publisher = "Ilya Kantor";
```

Ereditarietà delle proprietà e dei metodi statici

Anche le proprietà ed i metodi statici vengono ereditati.

Ad esempio, `Animal.compare` e `Animal.planet` nel codice sotto, vengono ereditate e diventano quindi accessibili come `Rabbit.compare` e `Rabbit.planet`:

```
class Animal {  
  static planet = "Earth";  
  
  constructor(name, speed) {  
    this.speed = speed;  
    this.name = name;  
  }  
}
```

```

run(speed = 0) {
  this.speed += speed;
  alert(`#${this.name} runs with speed ${this.speed}.`);
}

static compare(animalA, animalB) {
  return animalA.speed - animalB.speed;
}

}

// Eredita da Animal
class Rabbit extends Animal {
  hide() {
    alert(`#${this.name} hides!`);
  }
}

let rabbits = [
  new Rabbit("White Rabbit", 10),
  new Rabbit("Black Rabbit", 5)
];

rabbits.sort(Rabbit.compare);

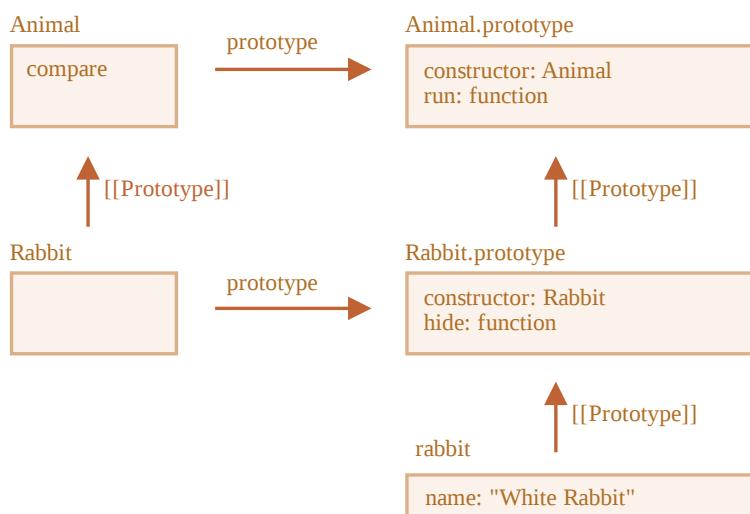
rabbits[0].run(); // Black Rabbit runs with speed 5.

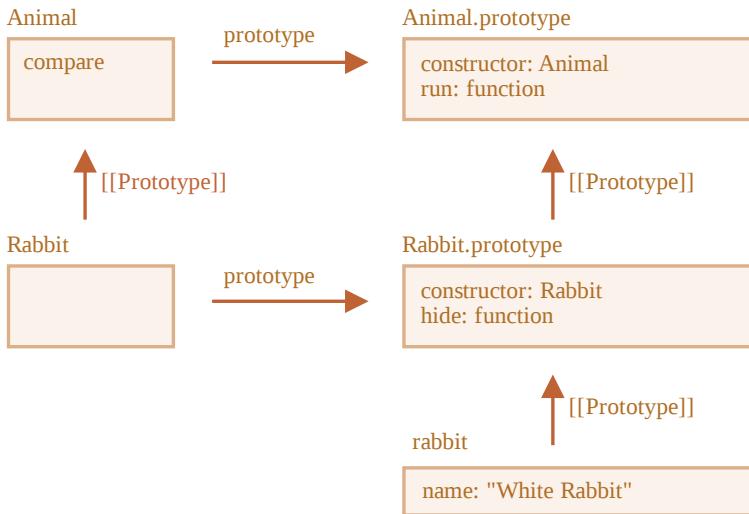
alert(Rabbit.planet); // Earth

```

Ora, quando invochiamo `Rabbit.compare`, verrà invocato il metodo `Animal.compare` ereditato.

Come funziona? Nuovamente, utilizzando il prototypes. Come potrete aver già intuito, `extends` fornisce a `Rabbit` il riferimento a `[[Prototype]]` di `Animal`.





1. La funzione `Rabbit` eredita dalla funzione di `Animal` .
2. `Rabbit.prototype` eredita il prototye di `Animal.prototype` .

Come risultato, l'ereditarietà funziona sia per i metodi regolari che per quelli statici.

Ora, verifichiamo quanto detto guardando al codice:

```

class Animal {}
class Rabbit extends Animal {}

// per proprietà statiche
alert(Rabbit.__proto__ === Animal); // true

// per proprietà regolari
alert(Rabbit.prototype.__proto__ === Animal.prototype); // true
  
```

Riepilogo

I metodi statici vengono utilizzati per funzionalità che appartengono all'intera classe. Non hanno nulla a che fare con l'istanza della classe.

Ad esempio, un metodo per il confronto `Article.compare(article1, article2)` o un factory method `Article.createTodays()`.

Queste vengono precedute dalla keyword `static` all'interno della dichiarazione della classe.

Le proprietà statiche vengono utilizzate quando si ha intenzione di memorizzare dati relativi alla classe, che non sono quindi legati ad un'istanza precisa.

La sintassi è:

```

class MyClass {
  static property = ...;

  static method() {
    ...
  }
}
  
```

Tecnicamente, le dichiarazioni di proprietà statiche equivalgono all'assegnazione diretta alla classe stessa:

```
MyClass.property = ...
MyClass.method = ...
```

Le proprietà ed i metodi statici vengono ereditati.

Nel caso in cui `class B extends A` il prototype della classe `B` punta ad `A : B`.
[[Prototype]] = `A`. Quindi se un campo non viene trovato in `B`, la ricerca continuerà in `A`.

✓ Esercizi

Class extends Object?

importanza: 3

Come ormai sappiamo, tutti gli oggetti, normalmente ereditano da `Object.prototype` ed hanno accesso ai metodi generici di `Object`, come `hasOwnProperty` etc.

Ad esempio:

```
class Rabbit {
  constructor(name) {
    this.name = name;
  }
}

let rabbit = new Rabbit("Rab");

// hasOwnProperty viene ereditato da Object.prototype
alert( rabbit.hasOwnProperty('name') ); // true
```

Ma se lo invocassimo esplicitamente in questo modo: "class Rabbit extends Object", allora il risultato sarebbe diverso da un semplice "class Rabbit"?

Qual'è la differenza?

Qui vediamo un esempio (perché non funziona? è possibile sistemarlo?):

```
class Rabbit extends Object {
  constructor(name) {
    this.name = name;
  }
}

let rabbit = new Rabbit("Rab");

alert( rabbit.hasOwnProperty('name') ); // Error
```

[Alla soluzione](#)

Proprietà e metodi privati e protetti

Uno dei concetti più importanti della programmazione ad oggetti è l'incapsulamento, ovvero la delimitazione delle interfacce interne da quelle esterne.

Questa pratica è un “must” nello sviluppo di una qualsiasi applicazione che sia più complessa di “hello world” .

Per comprenderla, usciamo dal mondo dello sviluppo e guardiamo al mondo reale.

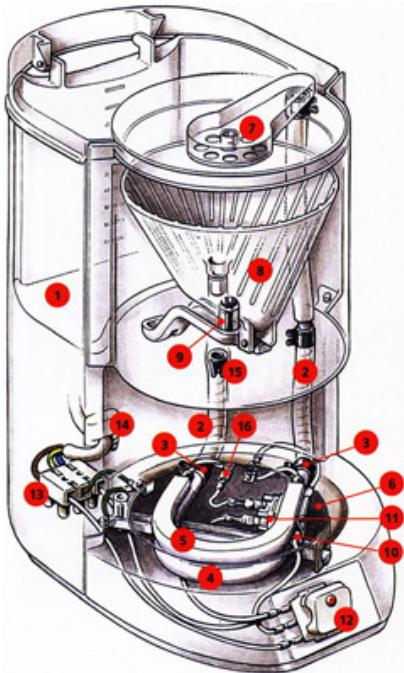
Solitamente, i dispositivi che utilizziamo sono piuttosto complessi. Poder delimitare la loro interfaccia interna da quella esterna, ci consente di utilizzarli senza grossi problemi.

Un esempio del mondo reale

Prendiamo l'esempio di una macchina del caffè. Semplice all'esterno: un bottone, un display, un paio di fori... E, ovviamente, il risultato: un ottimo caffè! :)



Ma internamente... (una rappresentazione dei suoi componenti)



Ci sono molti dettagli. Ma riusciamo comunque ad utilizzarla anche senza conoscerli.

Le macchine del caffè sono piuttosto affidabili, giusto? Possono durare per anni, e solamente nel caso in cui qualcosa smetta di funzionare, le portiamo a riparare.

Il segreto dietro all'affidabilità e alla semplicità di una macchina del caffè è che tutti i dettagli sono ottimizzati e *nascosti*.

Se rimuovessimo la copertura della macchina del caffè, allora il suo utilizzo sarebbe molto più complesso (dove dovremmo premere?), e pericoloso (potremmo prendere la scossa).

Come vedremo in seguito, nella programmazione gli oggetti sono come le macchine del caffè.

Ma per poter nascondere i loro dettagli interni, non utilizzeremo una copertura di sicurezza, ma piuttosto una speciale sintassi del linguaggio ed alcune convenzioni.

Interfaccia interna ed esterna

Nella programmazione orientata agli oggetti, le proprietà ed i metodi sono divisi in due gruppi:

- *Interfaccia interna* – metodi e proprietà, accessibili dagli altri metodi della classe, ma non dall'esterno.
- *Interfaccia esterna* – metodi e proprietà, accessibili anche dall'esterno della classe.

Continuando con l'analogia della macchina del caffè, ciò che è nascosto internamente (una pompa, un meccanismo di riscaldamento e così via) è la sua interfaccia interna.

L'interfaccia interna viene utilizzata per far funzionare l'oggetto, i suoi elementi interagiscono gli uni con gli altri. Ad esempio, la pompa è collegata al meccanismo di riscaldamento.

Ma vista dall'esterno, la macchina del caffè è protetta da una copertura, in modo che nessuno possa accedervi. I dettagli sono nascosti ed inaccessibili, ma possiamo sfrutarne le caratteristiche tramite la sua interfaccia esterna.

Quindi, tutto ciò di cui abbiamo bisogno per utilizzare un oggetto è la sua interfaccia esterna. Potremmo essere completamente inconsapevoli del suo funzionamento interno; e ciò andrebbe bene.

Questa era un'introduzione generale.

In JavaScript, esistono due tipi di campi per un oggetto (proprietà e metodi):

- Pubblici: accessibili ovunque. Questi ne definiscono l'interfaccia esterna. Finora abbiamo sempre utilizzato proprietà e metodi pubblici.
- Privati: accessibili solamente dall'interno della classe. Questi ne definiscono l'interfaccia interna.

In molti altri linguaggi di programmazione esiste anche il concetto di campo “protected” (protetto): accessibile solamente dall'interno della classe e da quelle che la estendono (come i campi privati, ma in aggiunta sono accessibili anche dalle classi che ereditano). Questi sono altrettanto utili per la definizione dell'interfaccia interna. Generalmente sono più diffusi dei campi privati, poiché solitamente la nostra intenzione è quella di renderli accessibili anche nelle sotto-classi.

I campi protetti non sono implementati in JavaScript a livello di linguaggio, ma nella pratica risultano essere molto utili, per questo vengono spesso emulati.

Ora costruiremo una macchina del caffè in JavaScript, con tutti i tipi di proprietà descritti. Una macchina del caffè è composta da molti dettagli; non la modelleremo per intero (anche se potremmo), in modo da mantenere l'esempio semplice.

Protecting “waterAmount”

Come prima cosa creiamo una semplice classe sul modello di una macchina del caffè:

```
class CoffeeMachine {  
    waterAmount = 0; // la quantità di acqua contenuta  
  
    constructor(power) {  
        this.power = power;  
        alert(`Created a coffee-machine, power: ${power}`);  
    }  
  
    // creiamo la macchina del caffè  
    let coffeeMachine = new CoffeeMachine(100);  
  
    // aggiungiamo acqua  
    coffeeMachine.waterAmount = 200;
```

Per ora le proprietà `waterAmount` e `power` sono pubbliche. Possiamo leggerle/modificarle dall'esterno con un qualsiasi valore.

Proviamo a modificare la proprietà `waterAmount` rendendola protetta, in modo da avere un maggior controllo su di essa. Ad esempio, non vorremmo che qualcuno possa impostarla con un valore negativo.

Le proprietà protette, solitamente, vengono prefissate con un underscore `_`.

Questa non è una forzatura del linguaggio, ma piuttosto una convenzione diffusa tra i programmatore, che specifica che queste proprietà e metodi non dovrebbero essere accessibili dall'esterno.

Quindi la nostra proprietà diventa `_waterAmount`:

```
class CoffeeMachine {
  _waterAmount = 0;

  set waterAmount(value) {
    if (value < 0) {
      value = 0;
    }
    this._waterAmount = value;
  }

  get waterAmount() {
    return this._waterAmount;
  }

  constructor(power) {
    this._power = power;
  }
}

// creiamo la macchina del caffè
let coffeeMachine = new CoffeeMachine(100);

// aggiungiamo acqua
coffeeMachine.waterAmount = -10; // _waterAmount diventerà 0, non -10
```

Ora l'accesso è sotto controllo, quindi non è più possibile impostare la quantità d'acqua ad un valore negativo.

Read-only “power”

Proviamo a rendere la proprietà `power` come read-only (sola lettura). In alcuni casi, potremmo aver bisogno di definire una proprietà in fase di costruzione, e non volerla più modificare in seguito.

Questo è esattamente il caso per un macchina del caffè: la potenza non può variare.

Per farlo, possiamo semplicemente definire un getter, e nessun setter:

```
class CoffeeMachine {
  // ...

  constructor(power) {
    this._power = power;
  }

  get power() {
    return this._power;
  }
}

// creiamo la macchina del caffè
let coffeeMachine = new CoffeeMachine(100);
```

```
alert(`Power is: ${coffeeMachine.power}W`); // Power is: 100W  
coffeeMachine.power = 25; // Errore (nessun setter)
```

➊ Le funzioni getter/setter

Qui abbiamo utilizzato la sintassi getter/setter.

Ma nella maggior parte dei casi, le funzioni `get.../set...` si preferisce definirle in questo modo:

```
class CoffeeMachine {  
    _waterAmount = 0;  
  
    setWaterAmount(value) {  
        if (value < 0) value = 0;  
        this._waterAmount = value;  
    }  
  
    getWaterAmount() {  
        return this._waterAmount;  
    }  
}  
  
new CoffeeMachine().setWaterAmount(100);
```

Potrebbe sembrare leggermente più lungo, ma l'utilizzo di funzioni li rende più flessibili. Possono accettare più argomenti (anche se per ora non ne abbiamo bisogno).

D'altra parte però, la sintassi `get/set` è più breve. In definitiva, non esiste una vera e propria regola, sta a voi decidere.

➋ I campi protetti vengono ereditati

Se ereditiamo `class MegaMachine extends CoffeeMachine`, allora nulla ci vieterà di accedere a `this._waterAmount` o `this._power` dai metodi nella nuova classe.

Quindi, i metodi protetti vengono ereditati. A differenza di quelli privati, che vedremo tra poco.

Private “#waterLimit”

⚠ Aggiunta di recente

Questa funzionalità è stata aggiunta di recente al linguaggio. Non supportata dai JavaScript engines, oppure supportata ancora parzialmente, richiede polyfilling.

Esiste una proposta JavaScript in via di approvazione, che fornisce il supporto per le proprietà ed i metodi privati.

I campi privati dovrebbero essere preceduti da `#`. Questi saranno accessibili solamente dall'interno della classe.

Ad esempio, qui abbiamo una proprietà privata `#waterLimit` e un metodo privato per il controllo del livello dell'acqua `#fixWaterAmount`:

```
class CoffeeMachine {
    #waterLimit = 200;

    #fixWaterAmount(value) {
        if (value < 0) return 0;
        if (value > this.#waterLimit) return this.#waterLimit;
    }

    setWaterAmount(value) {
        this.#waterLimit = this.#fixWaterAmount(value);
    }
}

let coffeeMachine = new CoffeeMachine();

// non possiamo accedere ai metodi privati dall'esterno della classe
coffeeMachine.#fixWaterAmount(123); // Errore
coffeeMachine.#waterLimit = 1000; // Errore
```

A livello di linguaggio, `#` è un carattere speciale per indicare che quel campo è privato. Non possiamo quindi accedervi dall'esterno o da una sotto-classe.

Inoltre i campi privati non entrano in conflitto con quelli pubblici. Possiamo avere sia un campo privato `#waterAmount` che uno pubblico `waterAmount`.

Ad esempio, facciamo sì che `waterAmount` sia una proprietà per accedere a `#waterAmount`:

```
class CoffeeMachine {

    #waterAmount = 0;

    get waterAmount() {
        return this.#waterAmount;
    }

    set waterAmount(value) {
        if (value < 0) value = 0;
        this.#waterAmount = value;
    }
}

let machine = new CoffeeMachine();

machine.waterAmount = 100;
alert(machine.#waterAmount); // Errore
```

A differenza di quelli protetti, i campi privati sono forniti dal linguaggio stesso. E questa è una buona cosa.

Nel caso in cui stessimo ereditando da `CoffeeMachine`, allora non avremmo accesso diretto a `#waterAmount`. Dovremmo affidarci al getter/setter `waterAmount`:

```
class MegaCoffeeMachine extends CoffeeMachine {  
    method() {  
        alert( this.#waterAmount ); // Errore: è possibile accedervi solamente da CoffeeMachine  
    }  
}
```

In molti casi, una limitazione del genere è troppo severa. Se estendiamo una `CoffeeMachine`, potremmo giustamente voler accedere ai suoi campi interni. Questo è il motivo per cui i campi protetti vengono usati più spesso, anche se non sono realmente supportati dalla sintassi del linguaggio.

I campi privati non sono accessibili come `this[name]`

I campi privati sono speciali.

Come sappiamo, solitamente possiamo accedere ai campi utilizzando `this[name]`:

```
class User {  
    ...  
    sayHi() {  
        let fieldName = "name";  
        alert(`Hello, ${this[fieldName]}`);  
    }  
}
```

Con i campi privati questo è impossibile: `this['#name']` non funzionerebbe. Questa è una limitazione sintattica per garantire la privacy.

Riepilogo

In termini di OOP (Programmazione Orientata agli Oggetti), la delimitazione dell'interfaccia interna da quella esterna si chiama [incapsulamento](#).

Fornisce diversi vantaggi:

Protezione per gli utenti, in modo che questi non possano spararsi ai piedi

Immaginiamo un team di sviluppatori che utilizzi una macchina del caffè costruita dall'azienda “Best CoffeeMachine”, che funziona correttamente, ma la cui protezione viene rimossa. In questo modo, la sua interfaccia interna viene esposta.

Tutti gli sviluppatori sono educati, ed utilizzano la macchina del caffè come previsto. Ma uno di loro, John, che crede di essere il più intelligente, effettua alcune modifiche alla macchina, la quale si rompe due giorni dopo.

Questa non è sicuramente colpa di John, ma piuttosto della persona che ha rimosso la protezione e ha permesso a John di manometterla.

Lo stesso vale nella programmazione. Se un utente prova a cambiare campi che non dovrebbero essere modificati dall'esterno, le conseguenze saranno imprevedibili.

Sostenibile

La situazione, nella programmazione, è più complessa rispetto ad una macchina del caffè reale, la quale viene semplicemente comprata ed utilizzata. Il codice è costantemente soggetto a sviluppo e miglioramenti.

Se limitiamo l'accesso all'interfaccia interna, allora lo sviluppatore della classe ha la possibilità di modificarla, anche senza dover informare gli utenti.

Se sei lo sviluppatore di una classe di questo tipo, ti farà piacere sapere che i metodi privati possono essere rinominati in totale sicurezza; i parametri possono essere modificati, o addirittura rimossi, poiché nessun codice esterno dipende da questi.

Per gli utenti, quando esce una nuova versione, questa potrebbe essere cambiata completamente al suo interno, ma l'aggiornamento rimane comunque un'operazione semplice se la sua interfaccia esterna è rimasta la stessa.

Nasconde la complessità

Le persone adorano utilizzare cose semplici. Almeno esternamente. Ciò che è interno è una questione diversa.

I programmati non fanno eccezione.

E' sempre preferibile nascondere i dettagli implementativi e fornire una semplice e ben documentata interfaccia esterna.

Per nasconde i componenti interni di un'interfaccia possiamo utilizzare le proprietà protette o private:

- I campi protetti vengono preceduti da `_`. Questa è una convenzione ben conosciuta, non implementata a livello di linguaggio. I programmati dovrebbero sempre accedere ai campi preceduti da `_` solamente dalla classe stessa o dalle sue sotto-classi.
- I campi privati vengono preceduti da `#`. JavaScript si assicura che questi siano accessibili solamente dalla loro classe.

Attualmente, i campi privati non sono completamente supportati dai browser, ma esistono dei polyfill.

Estendere le classi built-in

Le classi built-in (integrate) come `Array`, `Map` e tutte le altre, sono anch'esse estendibili.

Ad esempio, qui vediamo `PowerArray` ereditare dall'`Array` nativo:

```
// aggiungiamo un metodo (possiamo fare di più)
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }
}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false

let filteredArr = arr.filter(item => item >= 10);
```

```
alert(filteredArr); // 10, 50
alert(filteredArr.isEmpty()); // false
```

Notiamo una cosa interessante. I metodi built-in come `filter`, `map` e così via, ritornano nuovi oggetti del tipo ereditato, cioè `PowerArray`. La loro implementazione interna utilizzata la proprietà oggetto `constructor` per farlo.

Nell'esempio sopra,

```
arr.constructor === PowerArray
```

Quando invochiamo `arr.filter()`, questo creerà internamente il nuovo array contenente i risultati utilizzando `arr.constructor`, non l'oggetto `Array` standard. Questo è molto utile, poiché successivamente possiamo utilizzare i metodi di `PowerArray` sul risultato ottenuto.

Inoltre, possiamo personalizzarne il comportamento.

Possiamo aggiungere uno speciale getter statico `Symbol.species` alla classe. Questo dovrebbe ritornare il costruttore che Javascript utilizzerà internamente per creare le nuove entità in `map`, `filter` e così via.

Se, ad esempio, volessimo che metodi come `map` o `filter` restituiscano un array standard, possiamo ritornare `Array` in `Symbol.species`, come nell'esempio:

```
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }

  // i metodi built-in lo utilizzeranno come costruttore
  static get [Symbol.species]() {
    return Array;
  }
}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false

// filter crea un nuovo array utilizzando arr.constructor[Symbol.species] come costruttore
let filteredArr = arr.filter(item => item >= 10);

// filteredArr non è di tipo PowerArray, ma è un Array standard
alert(filteredArr.isEmpty()); // Error: filteredArr.isEmpty is not a function
```

Come potete osservare, ora `.filter` restituisce un `Array`. Quindi l'estensione delle funzionalità non sarà più disponibile.

Le altre collezioni funzionano in maniera simile

Le altre collezioni, come `Map` e `Set`, funzionano in maniera molto simile. Anche queste utilizzano `Symbol.species`.

Con gli oggetti built-in non si ereditano le proprietà statiche

Gli oggetti built-in possiedono i loro metodi statici, ad esempio `Object.keys`, `Array.isArray` etc.

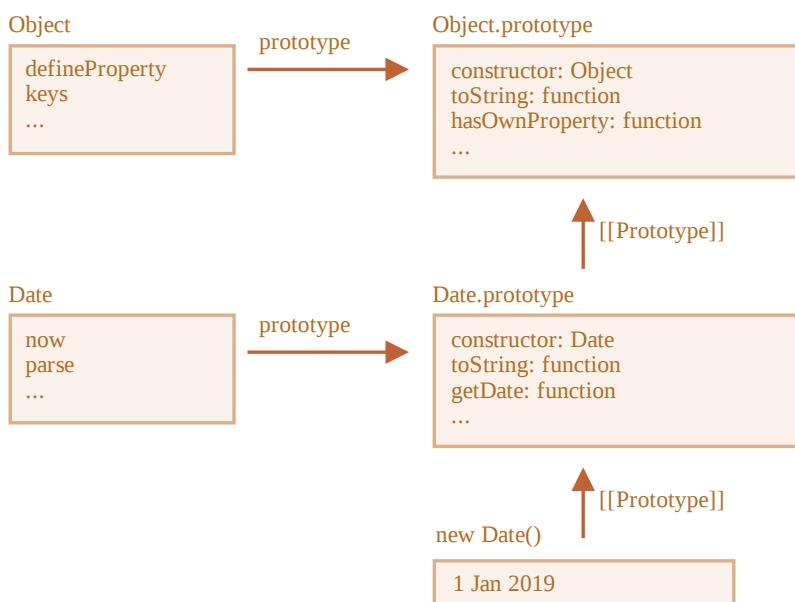
Come già sappiamo, le classi integrate si estendono a vicenda. Ad esempio, `Array` estende `Object`.

Normalmente, quando una classe ne estende un'altra, sia i metodi statici che quelli non-statici vengono ereditati. Questo è stato ampiamente spiegato nell'articolo [Proprietà e metodi statici](#).

Ma le classi built-in fanno eccezione. Queste, infatti, non ereditano i membri statici a le une dalle altre.

Ad esempio, sia `Array` che `Date` ereditano da `Object`, quindi le loro istanze possiedono i metodi di `Object.prototype`. Ma `Array.[[Prototype]]` non fa riferimento ad `Object`, quindi, ad esempio, non si ha alcun metodo statico come `Array.keys()` (o `Date.keys()`).

Qui vediamo raffigurata la struttura per `Date` e `Object`:



Questa è un'importante differenza dell'ereditarietà tra gli oggetti integrati, rispetto a quella che otteniamo tramite `extends`.

Verifica delle classi: "instanceof"

L'operatore `instanceof` ci consente di verificare se un oggetto appartiene ad una specifica classe. Anche l'ereditarietà viene presa in considerazione.

Questo tipo di controllo potrebbe essere necessario in diversi casi. Ad esempio, può essere utilizzato per costruire una funzione *polimorfa*, ossia una funzione che tratta gli argomenti differentemente in base al loro tipo.

L'operatore `instanceof`

La sintassi è:

```
obj instanceof Class
```

Ritorna `true` se `obj` è di tipo `Class` o è una sua sotto-classe.

Ad esempio:

```
class Rabbit {}  
let rabbit = new Rabbit();  
  
// è un oggetto di tipo Rabbit?  
alert( rabbit instanceof Rabbit ); // true
```

Funziona anche con i costruttori:

```
// invece di usare class  
function Rabbit() {}  
  
alert( new Rabbit() instanceof Rabbit ); // true
```

...E con le classi integrate come `Array`:

```
let arr = [1, 2, 3];  
alert( arr instanceof Array ); // true  
alert( arr instanceof Object ); // true
```

Da notare che `arr` appartiene anche alla classe `Object`. Questo accade perché `Array` eredita da `Object`.

Normalmente `instanceof` esamina la catena dei prototype per effettuare questa verifica. Possiamo anche definire una logica personalizzata nel metodo statico `Symbol.hasInstance`.

L'algoritmo di `obj instanceof Class` funziona, a grandi linee, in questo modo:

1. Se è stato definito un metodo statico `Symbol.hasInstance`, allora questo verrà invocato: `Class[Symbol.hasInstance](obj)`. Dovrebbe ritornare `true` o `false`, questo è tutto. In questo modo possiamo personalizzare il comportamento di `instanceof`.

For example:

```
// impostiamo il controllo instanceof in modo che assuma che  
// qualsiasi cosa con la proprietà canEat sia un animale  
class Animal {  
  static [Symbol.hasInstance](obj) {  
    if (obj.canEat) return true;  
  }  
}  
  
let obj = { canEat: true };  
  
alert(obj instanceof Animal); // true: Animal[Symbol.hasInstance](obj) è stato invocato
```

2. Molte classi non hanno `Symbol.hasInstance`. In questo caso, viene utilizzata la logica standard: `obj instanceof Class` che controlla se `Class.prototype` equivale ad uno dei prototype nella catena dei prototype di `obj`.

In altre parole, li confronta tutti uno alla volta:

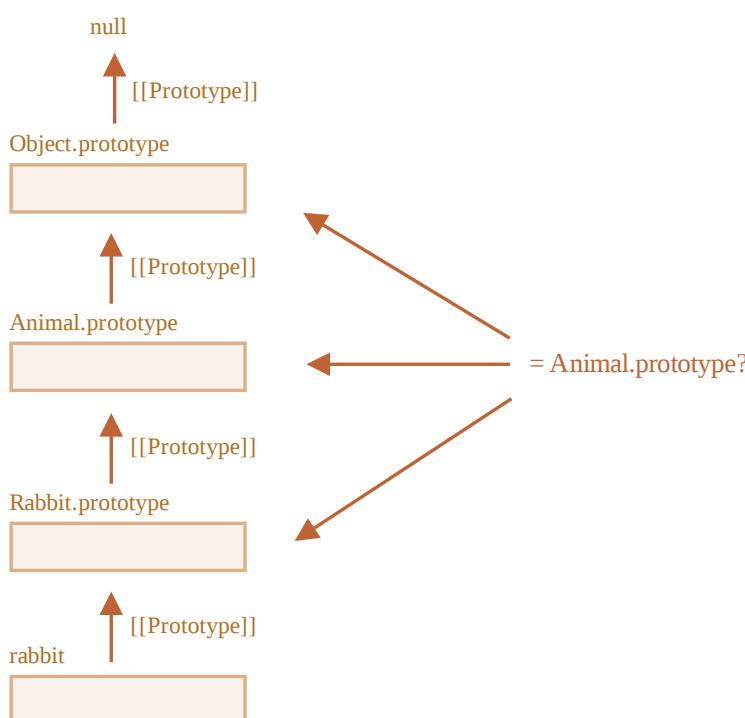
```
obj.__proto__ === class.prototype?  
obj.__proto__.__proto__ === Class.prototype?  
obj.__proto__.__proto__.__proto__ === Class.prototype?  
...  
// se una di questa è true, allora viene ritornato true  
// altrimenti, una volta arrivati al termine della catena, ritorna false
```

Nell'esempio sopra `rabbit.__proto__ === Rabbit.prototype`, quindi riceviamo immediatamente una risposta.

In caso di ereditarietà, il riscontro avverrà al secondo passo:

```
class Animal {}  
class Rabbit extends Animal {}  
  
let rabbit = new Rabbit();  
alert(rabbit instanceof Animal); // true  
  
// rabbit.__proto__ === Animal.prototype (no match)  
// rabbit.__proto__.__proto__ === Animal.prototype (match!)
```

Qui vediamo raffigurato cosa `rabbit instanceof Animal` confronta con `Animal.prototype`:



Comunque, abbiamo a disposizione anche il metodo `objA.isPrototypeOf(objB)` ↗, che ritorna `true` se `objA` si trova nella catena dei prototype di `objB`. Quindi la verifica `obj`

```
instanceof Class può essere riformulata come  
Class.prototype.isPrototypeOf(obj).
```

Un fatto divertente, è che il costruttore stesso della `Class`, non viene coinvolto nella verifica! Solamente la catena dei prototype e `Class.prototype` vengono valutati.

Questo può portare a diverse conseguenze quando la proprietà `prototype` viene modificata dopo la creazione dell'oggetto.

Come nell'esempio:

```
function Rabbit() {}  
let rabbit = new Rabbit();  
  
// modifichiamo il prototype  
Rabbit.prototype = {};  
  
// ...non è più un rabbit!  
alert( rabbit instanceof Rabbit ); // false
```

Bonus: Object.prototype.toString per il tipo

Sappiamo già che gli oggetti semplici vengono convertiti a stringa come `[object Object]`:

```
let obj = {};  
  
alert(obj); // [object Object]  
alert(obj.toString()); // lo stesso
```

Questa è la loro implementazione del metodo `toString`. Ma esiste una funzionalità nascosta che rende `toString` molto più potente di così. Possiamo utilizzarlo come un'estensione di `typeof` e come alternativa di `instanceof`.

Sembra strano? Lo è! Capiamo perché.

Secondo le [specifiche ↗](#), il metodo integrato `toString` può essere estratto dall'oggetto ed eseguito nel contesto di un qualsiasi altro valore. Ed il suo risultato dipende da quel valore.

- Per un numero, sarà `[object Number]`
- Per un boolean, sarà `[object Boolean]`
- Per `null`: `[object Null]`
- Per `undefined`: `[object Undefined]`
- Per gli array: `[object Array]`
- ...etc (personalizzabile).

Dimostriamolo:

```
// copiamo il metodo toString in una variabile per comodità  
let objectToString = Object.prototype.toString;  
  
// di che tipo è questo?
```

```
let arr = [];

alert( objectToString.call(arr) ); // [object Array]
```

Qui abbiamo utilizzato `call` come descritto nel capitolo *Decorators* e forwarding, `call/apply` per eseguire la funzione `objectToString` nel contesto `this=arr`.

Internamente, l'algoritmo `toString` esamina `this` e ritorna il risultato corrispondente. Altri esempi:

```
let s = Object.prototype.toString;

alert( s.call(123) ); // [object Number]
alert( s.call(null) ); // [object Null]
alert( s.call(alert) ); // [object Function]
```

Symbol.toStringTag

Il comportamento di `Object.toString` può essere personalizzato utilizzando una proprietà speciale dell'oggetto `Symbol.toStringTag`.

Ad esempio:

```
let user = {
  [Symbol.toStringTag]: "User"
};

alert( {}.toString.call(user) ); // [object User]
```

Per molti oggetti specifici di un ambiente, esiste questa proprietà. Qui vediamo alcuni esempi specifici per il browser:

```
// toStringTag per l'oggetto specifico d'ambiente:
alert( window[Symbol.toStringTag] ); // Window
alert( XMLHttpRequest.prototype[Symbol.toStringTag] ); // XMLHttpRequest

alert( {}.toString.call(window) ); // [object Window]
alert( {}.toString.call(new XMLHttpRequest()) ); // [object XMLHttpRequest]
```

Come potete vedere, il risultato è esattamente `Symbol.toStringTag` (se esiste), racchiuso in `[object ...]`.

Al termine avremo “`typeof` on steroids” che non funziona solamente con i tipi di dato primitivo, ma anche con gli oggetti integrati, inoltre può essere personalizzato.

Possiamo utilizzare `{}.toString.call` piuttosto di `instanceof` per gli oggetti integrati quando vogliamo ottenerne il tipo come stringa invece di effettuare una semplice verifica.

Riepilogo

Ricapitoliamo i metodi di verifica del tipi:

funzionano con	ritorna
<code>typeof</code>	primitivi
<code>{}.toString</code>	primitivi, oggetti integrati, oggetti con <code>Symbol.toStringTag</code>
<code>instanceof</code>	oggetti

Come possiamo vedere, `{}.toString` è tecnicamente più avanzato di `typeof`.

Invece l'operatore `instanceof` funziona veramente bene quando lavoriamo con una classe e vogliamo controllarne l'ereditarietà.

✓ Esercizi

Uno strano `instanceof`

importanza: 5

Nel codice sottostante, perché `instanceof` ritorna `true`? Possiamo facilmente vedere che `a` non è creato da `B()`.

```
function A() {}
function B() {}

A.prototype = B.prototype = {};

let a = new A();

alert( a instanceof B ); // true
```

[Alla soluzione](#)

Mixins

In JavaScript possiamo ereditare solamente da un oggetto. Può esserci solamente un `[[Prototype]]` per oggetto. Ed una classe può estendere solamente un'altra classe.

In certi casi questo può essere un limite. Ad esempio, abbiamo una classe `StreetSweeper` ed una classe `Bicycle`, e vogliamo crearne un mix: un `StreetSweepingBicycle`.

Oppure abbiamo una classe `User` ed una classe `EventEmitter` che implementa la generazione degli eventi, e vorremmo poter aggiungere la funzionalità di `EventEmitter` a `User`, cosicché i nostri utenti possano emettere eventi.

Esiste un concetto che può aiutare in questi casi, chiamato “mixins”.

Come definito in Wikipedia, un [mixin](#) è una classe contenente metodi che possono essere utilizzati da altre classi, senza che ci sia la necessità di ereditare da questa classe.

In altre parole, un *mixin* fornisce dei metodi che implementano delle funzionalità specifiche, che non andremo ad utilizzare da soli, ma piuttosto andremo ad aggiungere ad altre classi.

Un esempio di mixin

Il modo più semplice per implementare un mixin in JavaScript è quello di creare un oggetto con dei metodi utili, in questo modo potremo fonderli molto semplicemente nel prototype di un'altra classe.

Ad esempio, qui vediamo il mixin `sayHiMixin` che viene utilizzato per aggiungere la funzionalità di “parlare” a `User`:

```
// mixin
let sayHiMixin = {
  sayHi() {
    alert(`Hello ${this.name}`);
  },
  sayBye() {
    alert(`Bye ${this.name}`);
  }
};

// utilizzo:
class User {
  constructor(name) {
    this.name = name;
  }
}

// copiamo i metodi
Object.assign(User.prototype, sayHiMixin);

// ora User può salutare
new User("Dude").sayHi(); // Hello Dude!
```

Non abbiamo utilizzato l'ereditarietà, ma abbiamo semplicemente copiato un metodo. Quindi `User` può tranquillamente ereditare da un'altra classe, ed includere il mixin per aggiungere funzionalità, come nell'esempio:

```
class User extends Person {
  // ...
}

Object.assign(User.prototype, sayHiMixin);
```

I mixins possono utilizzare a loro volta l'ereditarietà.

Ad esempio, qui abbiamo `sayHiMixin` che eredita da `sayMixin`:

```
let sayMixin = {
  say(phrase) {
    alert(phrase);
  }
};

let sayHiMixin = {
  __proto__: sayMixin, // (oppure potremmo utilizzare Object.setPrototypeOf per impostare il pro
```

```

sayHi() {
    // invocazione del metodo genitore
    super.say(`Hello ${this.name}`); // (*)
},
sayBye() {
    super.say(`Bye ${this.name}`); // (*)
}
};

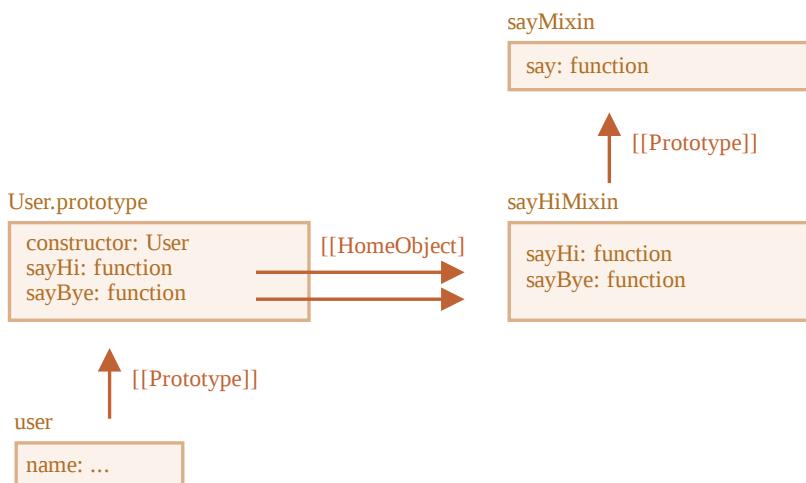
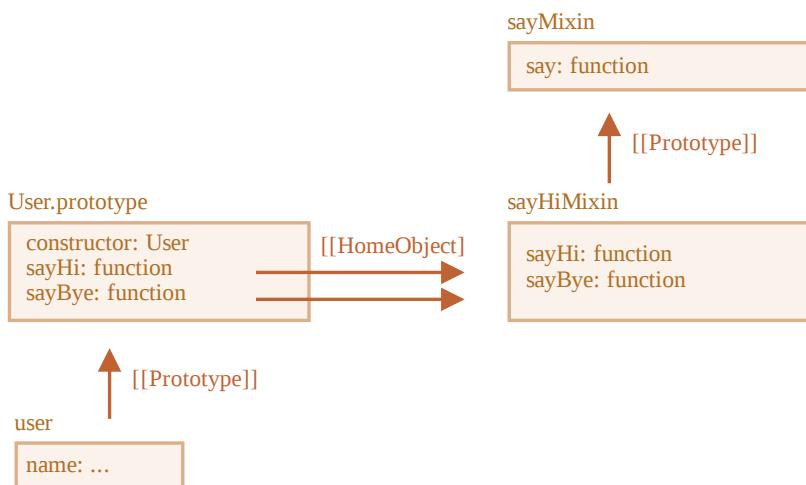
class User {
    constructor(name) {
        this.name = name;
    }
}

// copiamo i metodi
Object.assign(User.prototype, sayHiMixin);

// ora User può salutare
new User("Dude").sayHi(); // Hello Dude!

```

Da notare che l'invocazione al metodo padre `super.say()` da `sayHiMixin` (alla riga etichettata con `(*)`) cerca il metodo nel prototype di quel mixin, non in quello della classe.



Questo accade perché i metodi `sayHi` e `sayBye` sono stati creati in `sayHiMixin`. Quindi, anche dopo essere stati copiati, le loro proprietà `[[HomeObject]]` fanno riferimento a

`sayHiMixin`, come mostrato nella figura.

Poiché `super` ricerca i metodi in `[[HomeObject]].[[Prototype]]`, ciò significa che ricerca `sayHiMixin.[[Prototype]]`, non `User.[[Prototype]]`.

EventMixin

Ora creiamo un mixin per la vita reale.

Una caratteristica importante di molti oggetti del browser (ad esempio) è che questi possono generare eventi. Gli eventi sono un'ottimo modo per “trasmettere informazioni” a chiunque ne sia interessato. Quindi creiamo un mixin che ci consenta di aggiungere funzioni relative agli eventi, ad una qualsiasi classe/oggetto.

- Il mixin fornirà un metodo `.trigger(name, [...data])` per “generare un evento” quando qualcosa di significativo accade. L’argomento `name` è il nome dell’evento, ed altri argomenti opzionali possono essere aggiunti con dati relativi all’evento.
- Anche il metodo `.on(name, handler)`, che aggiunge una funzione `handler` come listener degli eventi con il nome fornito. Sarà invocato nel momento in cui un evento con il `name` fornito verrà invocato dalla chiamata `.trigger`.
- ...Ed il metodo `.off(name, handler)` che rimuove il listener `handler`.

Dopo aver aggiunto il mixin, un oggetto `user` sarà in grado di generare un evento di `"login"` quando l’utente effettua l’accesso. Ed un altro oggetto, diciamo, `calendar` può stare in ascolto di questi eventi in modo da caricare il calendario della persona autenticata.

Oppure un `menu` può generare un evento di `"select"` quando un elemento viene selezionato, ed un altro oggetto stare in ascolto dell’evento. E così via.

Qui vediamo il codice:

```
let eventMixin = {
  /**
   * Iscrizione ad un evento, utilizzo:
   * menu.on('select', function(item) { ... })
  */
  on(eventName, handler) {
    if (!this._eventHandlers) this._eventHandlers = {};
    if (!this._eventHandlers[eventName]) {
      this._eventHandlers[eventName] = [];
    }
    this._eventHandlers[eventName].push(handler);
  },

  /**
   * Cancellare l'iscrizione, utilizzo:
   * menu.off('select', handler)
  */
  off(eventName, handler) {
    let handlers = this._eventHandlers?.[eventName];
    if (!handlers) return;
    for (let i = 0; i < handlers.length; i++) {
      if (handlers[i] === handler) {
        handlers.splice(i--, 1);
      }
    }
  }
};
```

```

        },
        /**
         * Generare un evento con uno specifico nome ed i dati relativi
         * this.trigger('select', data1, data2);
        */
trigger(eventName, ...args) {
  if (!this._eventHandlers?.[eventName]) {
    return; // nessun gestore per questo evento
  }

  // invochiamo i gestori
  this._eventHandlers[eventName].forEach(handler => handler.apply(this, args));
}
};


```

- `.on(eventName, handler)` – assegna la funzione `handler` in modo tale che venga eseguita quando l'evento con il nome fornito viene generato. Tecnicamente, avremmo a disposizione anche la proprietà `_eventHandlers` che memorizza un array di gestori per ogni tipo di evento, quindi potremmo semplicemente aggiungerlo alla lista.
- `.off(eventName, handler)` – rimuove la funzione dalla lista dei gestori.
- `.trigger(eventName, ...args)` – genera l'evento: tutti i gestori in `_eventHandlers[eventName]` vengono invocati con la lista degli argomenti `...args`.

Utilizzo:

```

// Definiamo una classe
class Menu {
  choose(value) {
    this.trigger("select", value);
  }
}
// Aggiungiamo il mixin con i metodi relativi agli eventi
Object.assign(Menu.prototype, eventMixin);

let menu = new Menu();

// aggiungiamo un gestore, da invocare alla selezione:
menu.on("select", value => alert(`Value selected: ${value}`));

// inneschiamo l'evento => il gestore definito sopra verrà invocato e mostrerà:
// Value selected: 123
menu.choose("123");

```

Ora, nel caso volessimo che un'altra parte di codice reagisca alla selezione nel menu, ci basterà semplicemente aggiungere un listener con `menu.on(...)`.

E grazie al mixin `eventMixin`, questo comportamento diventa molto semplice da integrare in tutte le classi in cui desideriamo aggiungerlo, senza che questo vada ad interferire con l'ereditarietà.

Riepilogo

Mixin – è un termine utilizzato nella programmazione orientata agli oggetti: un classe che contiene metodi utili per altre classi.

Molti altri linguaggi di programmazione consentono l'ereditarietà multipla. JavaScript non la supporta, ma possiamo implementare i mixin copiando i loro metodi all'interno del prototype.

Possiamo utilizzare i mixins per migliorare una classe, andando ad aggiungere diversi comportamenti, come la gestione degli eventi vista sopra.

I mixins potrebbero creare conflitti nel caso in cui andassero a sovrascrivere metodi già esistenti nella classe. Quindi, generalmente, i nomi dei metodi nei mixin vanno scelti con attenzione, in modo tale da minimizzare il rischio che si generino tali conflitti.

Gestione degli errori

Gestione degli errori, "try...catch"

Non importa quanto siamo bravi a programmare, a volte i nostri scripts contengono errori. Questo può accadere a causa di un nostro errore, un input da parte dell'utente inatteso, a una risposta sbagliata da parte del server e per un altro centinaio di ragioni.

Di solito, uno script “muore” (si ferma immediatamente) al verificarsi di un errore, stampandolo in console.

Ma esiste il costrutto `try...catch` che permette di “catturare” gli errori e, anziché farlo morire, ci permette di fare qualcosa di più ragionevole.

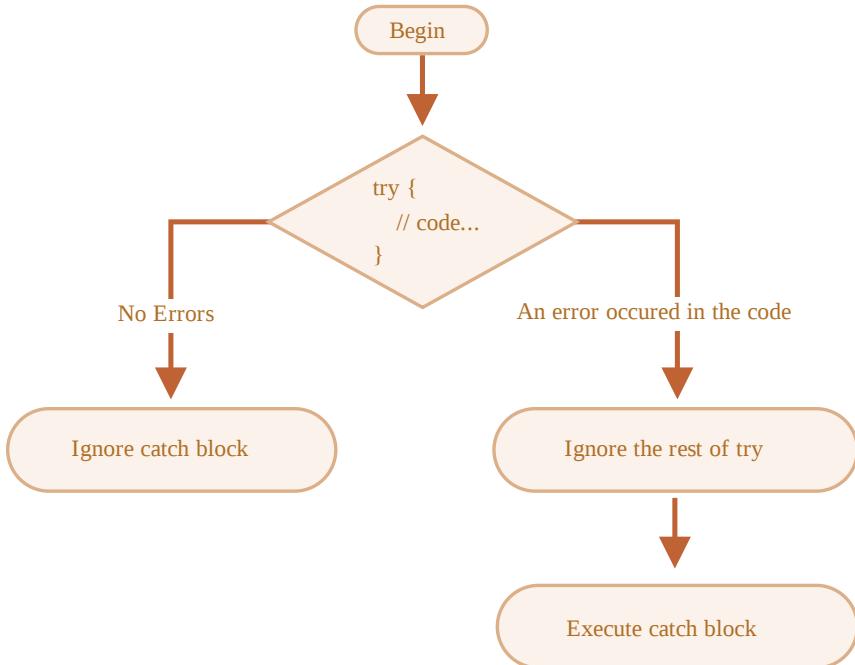
La sintassi “try...catch”

Il costrutto `try...catch` è composto da due blocchi principali: `try` e `catch`:

```
try {  
    // codice...  
}  
  catch (err) {  
    // gestione dell'errore  
}
```

Funziona in questo modo:

1. Per prima cosa, il codice all'interno del blocco `try { . . . }` viene eseguito.
2. Se non si verifica alcun errore, allora `catch(err)` viene ignorato: viene eseguito tutto il codice all'interno del `try` e viene saltato quello all'interno del `catch`.
3. Se si verifica un errore, allora l'esecuzione del resto del codice all'interno del `try` viene interrotta, e si passa all'esecuzione del codice all'interno di `catch(err)`. La variabile `err` (può essere usato ogni nome) contiene un oggetto di tipo Error (Error Object) con i dettagli riguardo a cosa sia successo.



Quindi, un errore all'interno del blocco `try { ... }` non ucciderà lo script: avremo la possibilità di gestirlo all'interno del blocco `catch`.

Vediamo degli esempi.

- Un esempio senza errori: saranno visualizzati `alert (1)` e `(2)`:

```

try {

  alert('Inizio l\'esecuzione di try'); // (1) <-- 

  // ...qui nessun errore

  alert('Fine dell\'esecuzione di try'); // (2) <-- 

} catch (err) {

  alert('Catch viene ignorato, dato che non ci sono errori'); // (3)

}
  
```

- Un esempio con un errore: saranno visualizzati `(1)` e `(3)`:

```

try {

  alert('Inizio l\'esecuzione di try'); // (1) <-- 

  lalala; // errore, variabile non definita!

  alert('Fine dell\'esecuzione di try (mai raggiunta)'); // (2)

} catch (err) {

  alert(`Si è verificato un errore!`); // (3) <-- 

}
  
```

⚠️ `try...catch` funziona solamente per gli errori che si verificano durante l'esecuzione (runtime errors)

In modo che `try...catch` funzioni, il codice deve essere eseguibile. In altre parole, dev'essere un script JavaScript valido.

Non funzionerà se il codice è sintatticamente errato, per esempio se ha delle parentesi graffe non accoppiate:

```
try {
    {{{
} } catch (err) {
    alert("Il motore (engine) non riesce a interpretare il codice, esso non è valido");
}
```

Il motore di JavaScript dapprima legge il codice, dopodiché lo esegue. Gli errori che si presentano durante la fase di lettura vengono definiti “parse-time” e sono non recuperabili (unrecoverable) (dal codice stesso). Questo perché il motore non riesce a interpretare il codice.

Quindi, `try...catch` può solo gestire gli errori presenti in un codice comunque valido. Tali errori vengono chiamati “errori di runtime” (runtime errors) o, a volte, “eccezioni” (exceptions).

⚠️ try...catch funziona in maniera sincrona

Se un'eccezione si verifica all'interno di codice "schedulato", come nel caso di `setTimeout`, allora `try...catch` non riesce ad intercettarlo:

```
try {  
    setTimeout(function() {  
        variabileNonDefinita; // script morirà qui  
    }, 1000);  
} catch (e) {  
    alert( "non funziona" );  
}
```

Questo accade perché il codice all'interno della funzione sarà eseguito successivamente, quando il motore avrà già interpretato il costrutto `try...catch`.

Per intercettare un'eccezione all'interno di una funzione schedulata, `try...catch` dev'essere all'interno di tale funzione

```
setTimeout(function() {
  try {
    variabileNonDefinita; // try...catch gestisce l'errore!
  } catch {
    alert( "qui c'è un errore!" );
  }
}, 1000);
```

Oggetto di tipo Error (Error Object)

Quando un errore si verifica, JavaScript genera un oggetto contenente i dettagli al riguardo. L'oggetto è passato come argomento al `catch`:

```
try {
  // ...
} catch(err) { // <- "error object", potreste usare un'altra parola al posto di err
  // ...
}
```

Per tutti gli errori standard, incorporati, l'oggetto errore ha due proprietà principali:

`name`

Il nome dell'errore. Ad esempio, per una variabile non definita sarà `"ReferenceError"`.

`message`

Il messaggio testuale con i dettagli dell'errore.

Esistono altre proprietà non standard disponibili in diverse condizioni. Una di quelle più largamente utilizzate e supportate è:

`stack`

Lo stack alla chiamata corrente: una stringa con le informazioni inerenti la sequenza delle chiamate effettuate che hanno portato all'errore. Utile a scopo di debugging.

Ad esempio:

```
try {
  lalala; // errore, variabile non definita!
} catch (err) {
  alert(err.name); // ReferenceError
  alert(err.message); // lalala non è definito
  alert(err.stack); // ReferenceError: lalala non è definito a (...call stack)

  // Può essere anche visualizzato nel suo complesso
  // L'errore è convertito in una stringa del tipo "name: message"
  alert(err); // ReferenceError: lalala non è definito
}
```

associazione “`catch`” opzionale

Aggiunta di recente

Questa funzionalità è stata aggiunta di recente al linguaggio. Vecchi browsers potrebbero richiedere polyfills.

Se non abbiamo bisogno dei dettagli dell'errore, `catch` può ometterlo:

```
try {
  // ...
} catch { // <-- senza (err)
  // ...
}
```

Usare “try...catch”

Esploriamo quindi l'uso di `try...catch` nella vita reale.

Come già sappiamo, JavaScript supporta il metodo [JSON.parse\(str\)](#) per leggere le variabili codificate in JSON.

Generalmente è usato per decodificare i dati ricevuti attraverso la rete, dal server o da altri sorgenti.

Riceviamo essi e chiamiamo `JSON.parse` così:

```
let json = '{"name":"John", "age": 30}'; // dati dal server
let user = JSON.parse(json); // converto il testo che rappresenta un oggetto JS

// ora user è un oggetto con delle proprietà
alert( user.name ); // John
alert( user.age ); // 30
```

Puoi trovare maggiori informazioni riguardo il JSON nel capitolo [Metodi JSON, toJSON](#).

Se `json` è malformato, `JSON.parse` genererà un errore, quindi lo script “morirà”.

Dovremmo essere soddisfatti di questo? Ovviamente, no!

In questo modo, se qualcosa va storto, il visitatore non saprà mai perché (a meno che non apra la console per sviluppatori). E in genere gli utenti non gradiscono affatto che qualche cosa sia andata storta senza avere alcun messaggio di errore.

Quindi usiamo `try...catch` per gestire l'errore:

```
let json = "{ bad json }";
try {
  let user = JSON.parse(json); // <-- si verifica un errore..
  alert( user.name ); // non funziona
} catch (err) {
  // ...l'esecuzione prosegue qui
  alert( "ci scusiamo, ma i dati contengono errori, proveremo a chiederli nuovamente." );
  alert( err.name );
  alert( err.message );
}
```

Qui abbiamo utilizzato il blocco `catch` solamente per visualizzare un messaggio, ma possiamo fare molto altro: inviare una nuova richiesta al server, suggerire un'alternativa al visitatore, inviare

le informazioni inerenti l'errore a un servizio di logging, ... Molto meglio che far semplicemente morire lo script.

Usare i nostri errori personalizzati

Che succede se il `json` è sintatticamente corretto, ma non la proprietà richiesta `name`?

Tipo così:

```
let json = '{ "age": 30 }'; // dati incompleti

try {

  let user = JSON.parse(json); // <-- non ci sono errori
  alert( user.name ); // nessuna proprietà name!

} catch (err) {
  alert( "non posso eseguire" );
}
```

Qui `JSON.parse` viene eseguito correttamente, ma l'assenza di `name` è per noi un errore.

Per unificare la gestione degli errori, useremo l'operatore `throw`.

Operatore “Throw”

L'operatore `throw` serve a generare un errore.

La sintassi è:

```
throw <error object>
```

Tecnicamente, possiamo usare qualsiasi cosa come oggetto errore (error object). Potrebbe essere una qualunque primitiva, come un numero (`number`) o una stringa (`string`), ma è meglio utilizzare un oggetto (`object`), preferibilmente con le proprietà `name` e `message` (per mantenere la compatibilità con gli errori già inclusi).

JavaScript ha già molti costruttori integrati per errori generici: `Error`, `SyntaxError`, `ReferenceError`, `TypeError` e altri. Possiamo usarli per creare un oggetto errore.

La sintassi è:

```
let error = new Error(message);
// or
let error = new SyntaxError(message);
let error = new ReferenceError(message);
// ...
```

Per gli errori integrati (non per qualunque oggetto, solo per gli errori), la proprietà `name` è esattamente il nome del costruttore. E `message` è preso dall'argomento.

Ad esempio:

```

let error = new Error("Qualcosa è successo o_0");

alert(error.name); // Errore
alert(error.message); // Qualcosa è successo o_0

```

Vediamo quindi che errore viene generato da `JSON.parse`:

```

try {
  JSON.parse("{ json errato o_0 }");
} catch(err) {
  alert(err.name); // SyntaxError
  alert(err.message); // Simbolo inaspettato (Unexpected token) o nel JSON alla posizione 0
}

```

Come possiamo vedere, è un `Errore di Sintassi (SyntaxError)`.

Vediamo l'altro caso, in cui l'assenza di `name` è un errore, poiché gli utenti devono avere la proprietà `name`.

Quindi eseguiamo:

```

let json = '{ "age": 30 }'; // dati incompleti

try {
  let user = JSON.parse(json); // <-- nessun errore

  if (!user.name) {
    throw new SyntaxError("Dati incompleti: manca name"); // (*)
  }

  alert( user.name );
}

} catch(err) {
  alert( "JSON Error: " + err.message ); // Errore JSON: dati incompleti (Incomplete data): manc
}

```

Alla linea `(*)`, l'operatore `throw` genera un `SyntaxError` con il `message` fornito, allo stesso modo in cui lo genererebbe JavaScript. L'esecuzione di `try` si ferma immediatamente e il controllo passa direttamente all'interno di `catch`.

Quindi `catch` diventa un singolo posto per la gestione di tutti gli errori: sia per `JSON.parse` che per tutti gli altri casi.

Rethrowing

Nel precedente esempio abbiamo usato `try...catch` per gestire i dati non corretti. Ma è possibile che *un altro errore inaspettato* si verifichi all'interno del blocco `try {...}`? Come un errore di programmazione (variabile non definita) o qualcos'altro, non solo qualcosa come i "dati non corretti".

Come questo:

```

let json = '{ "age": 30 }'; // dati incompleti

try {
  user = JSON.parse(json); // <-- abbiamo dimenticato di inserire "let" prima di user

  // ...
} catch (err) {
  alert("JSON Error: " + err); // Errore JSON: ReferenceError: la variabile utente non è definita
  // (non c'è nessun errore JSON)
}

```

Ovviamente, tutto è possibile! I programmatori commettono errori. Anche nelle utility open source utilizzate da milioni di persone per decenni – improvvisamente può essere scoperto un bug che porta a terribili hack.

Nel nostro caso, `try...catch` è pensato per intercettare errori per “dati non corretti”. Ma per sua natura, `catch` prende *tutti* gli errori in `try`. Qui intercetta un errore inaspettato, tuttavia visualizza ugualmente il messaggio “JSON Error”. Questo è sbagliato e rende il debug del codice più difficoltoso.

Per evitare questi problemi, possiamo utilizzare la tecnica di “rethrowing”. La regola è molto semplice:

Catch dovrebbe processare solamente gli errori che riconosce e “rilanciare” (rethrow) tutti gli altri.

La tecnica “rethrowing” può essere spiegata più in dettaglio come:

1. Catch intercetta tutti gli errori.
2. Nel blocco `catch (err) {...}` analizziamo l’oggetto errore (Object Error) `err`.
3. Se non sappiamo come gestirlo, allora ne usciremo con `throw err`.

Usually, we can check the error type using the `instanceof` operator:

```

try {
  user = { /*...*/ };
} catch (err) {
  if (err instanceof ReferenceError) {
    alert('ReferenceError'); // "ReferenceError" for accessing an undefined variable
  }
}

```

Possiamo ottenere il nome della classe di errore dalla proprietà `err.name`. Tutti gli errori nativi la possiedono. Un’altra opzione può esser quella di leggere `err.constructor.name`.

La regola è semplice:

Catch dovrebbe processare solamente gli errori che riconosce e “rilanciare” (rethrow) tutti gli altri.

La tecnica “rethrowing” può essere spiegata più in dettaglio come:

1. Catch intercetta tutti gli errori.
2. Nel blocco `catch(err) {...}` analizziamo l’oggetto errore (Object Error) `err`.
3. Se non sappiamo come gestirlo, allora ne usciremo con `throw err`.

Nel codice seguente, useremo rethrowing in modo che `catch` gestisca solamente un `SyntaxError`:

```
let json = '{ "age": 30 }'; // dati incompleti
try {

  let user = JSON.parse(json);

  if (!user.name) {
    throw new SyntaxError("Dati incompleti: manca la proprietà name");
  }

  blabla(); // errore inaspettato

  alert( user.name );

} catch (err) {

  if (err instanceof SyntaxError) {
    alert( "JSON Error: " + err.message );
  } else {
    throw err; // rethrow (*)
  }
}

}
```

Genereremo un errore nel blocco `catch` alla linea (*) “uscendo” dal `try...catch` e potremo catturare nuovamente quest’errore con un costrutto `try...catch` più esterno (se esiste), altrimenti lo script morirà.

Quindi, attualmente il blocco `catch` gestisce solamente gli errori che conosce e per cui è stato istruito e “ignora” tutti gli altri.

Il seguente esempio dimostra come altri errori possono essere catturati da più livelli di `try...catch`:

```
function readData() {
  let json = '{ "age": 30 }';

  try {
    // ...
    blabla(); // errore!
  } catch (err) {
    // ...
    if (!(err instanceof SyntaxError)) {
      throw err; // rethrow (non so come gestirlo)
    }
  }
}

try {
  readData();
} catch (err) {
  alert( "External catch got: " + err ); // catturato!
}
```

In questo caso `readData` sa solamente come gestire un `SyntaxError`, mentre il `try...catch` più esterno sa come gestire tutto il resto.

try...catch...finally

Aspetta, non è tutto.

Il costrutto `try...catch` può avere una o più clausole: `finally`.

Se esiste, il codice all'interno delle clausole verrà eseguito in ogni caso:

- dopo `try`, se non si sono verificati errori,
- dopo `catch`, se si sono verificati errori.

La sintassi estesa sarà più o meno così:

```
try {  
  ... provo ad eseguire il codice ...  
} catch (err) {  
  ... gestisco gli errori ...  
} finally {  
  ... eseguo in ogni caso ...  
}
```

Proviamo ad eseguire questo codice:

```
try {  
  alert( 'try' );  
  if (confirm('Vuoi generare un errore?')) BAD_CODE();  
} catch (err) {  
  alert( 'catch' );  
} finally {  
  alert( 'finally' );  
}
```

Il codice ha due modi per terminare l'esecuzione:

1. Se si risponde "Sì" alla domanda "Vuoi generare un errore?", allora `try -> catch -> finally`.
2. Se si risponde "No", allora `try -> finally`.

La clausola `finally` è spesso utilizzata quando iniziamo a fare qualcosa e vogliamo che in ogni caso finalizzare il risultato.

Per esempio, vogliamo misurare il tempo che impiega una funzione di Fibonacci `fib(n)`. Naturalmente, dobbiamo iniziare la misurazione prima che essa venga eseguita e terminarla subito dopo. Ma cosa accade se si verifica un errore durante il richiamo della funzione? In particolare, l'implementazione di `fib(n)` nel codice che segue ritorna un errore in caso di numeri negativi o non interi.

La clausola `finally` è il posto migliore dove terminare la misurazione senza dover tener conto di cosa sia successo.

In questo caso `finally` garantisce la misurazione del tempo impiegato correttamente in entrambe le situazioni – sia nel caso di un'esecuzione corretta di `fib` che nel caso si verifichi un errore in essa:

```
let num = +prompt("Inserire un numero positivo?", 35)

let diff, result;

function fib(n) {
  if (n < 0 || Math.trunc(n) != n) {
    throw new Error("Deve non essere negativo, oltre che intero.");
  }
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

let start = Date.now();

try {
  result = fib(num);
} catch (err) {
  result = 0;
} finally {
  diff = Date.now() - start;
}

alert(result || "si è verificato un errore");

alert(`l'esecuzione è durata ${diff}ms`);
```

Possiamo verificare il codice eseguendolo e inserendo `35` al `prompt` – verrà eseguito normalmente, `finally` dopo `try`. E se inseriamo `-1` – ci sarà un errore immediato, e l'esecuzione durerà `0ms`. Entrambe le misurazioni saranno corrette.

In altre parole, la funzione potrà terminare con `return` o `throw`, non avrà alcuna importanza. La clausola `finally` verrà eseguita in ogni caso.

i Le variabili sono locali all'interno di `try...catch...finally`

Presta attenzione al fatto che le variabili `result` e `diff` nel codice precedente sono dichiarate prima del `try...catch`.

Altrimenti, se dichiariamo `let` all'interno del blocco `try`, risulterà visibile solamente all'interno del blocco stesso.

i finally e return

La clausola `finally` funziona indifferentemente da come si esce dal blocco `try...catch`. Questo include anche un esplicito `return`.

Nel codice seguente, c'è un `return` nel `try`. In questo caso, `finally` è eseguito giusto prima che il controllo passi al resto del codice.

```
function func() {  
  
    try {  
        return 1;  
  
    } catch (err) {  
        /* ... */  
    } finally {  
        alert('finally');  
    }  
}  
  
alert( func() ); // prima viene eseguito l>alert del finally, e successivamente questo qua
```

i try...finally

Anche il costrutto `try...finally`, senza la clausola `catch` può risultare utile. Lo useremo se non vogliamo gestire l'errore in questo momento (ignorandolo), ma vogliamo essere sicuri che il processo che abbiamo avviato sia finalizzato ugualmente.

```
function func() {  
    // iniziamo ad eseguire qualcosa che necessita di essere completata (come una misurazione)  
    try {  
        // ...  
    } finally {  
        // completo qualunque cosa sia accaduta  
    }  
}
```

Nel codice qui sopra, un errore all'interno di `try` vi farà uscire sempre fuori dal costrutto, perché non c'è `catch`. Ma `finally` verrà eseguito ugualmente prima che il flusso lascerà la funzione.

Catch globale

⚠️ Strettamente legato all'ambiente di esecuzione

Le informazioni all'interno di questa sezione non fanno parte strettamente di JavaScript.

Immaginiamo di incorrere in un errore fatale fuori dal `try...catch`, e lo script muore. Esattamente come un errore di programmazione, non è il massimo, se non un terribile risultato.

Esiste un modo per reagire a un situazione simile? Possiamo creare un log dell'errore, mostrare qualcosa all'utente (che normalmente non vede i messaggi di errore), ecc.

Non esiste nulla nelle specifiche, ma l'ambiente in genere ci viene incontro, poiché risulta veramente utile. Ad esempio, Node.js ha `process.on("uncaughtException")` [per questo](#). E nel browser possiamo assegnare una funzione alla proprietà speciale `window.onerror` [, che verrà eseguita nel caso di un errore non catturato.](#)

La sintassi:

```
window.onerror = function(message, url, line, col, error) {  
    // ...  
};
```

message

Messaggio di errore.

url

Indirizzo URL dello script in cui si è verificato l'errore.

line, col

Linea e colonna in cui si è verificato l'errore.

error

L'oggetto errore (Error Object).

Ad esempio:

```
<script>  
window.onerror = function(message, url, line, col, error) {  
    alert(`#${message}\n At ${line}:${col} of ${url}`);  
}  
  
function readData() {  
    badFunc(); // Ops, qualcosa è andato storto!  
}  
  
readData();  
</script>
```

Il ruolo del gestore globale `window.onerror` non è quello di risolvere l'esecuzione dello script – cosa probabilmente impossibile nell'eventualità di errori di programmazione, ma d'inviare messaggi di errore agli sviluppatori.

Esistono anche dei web-services che forniscono servizi di error-logging, come <https://errorception.com> [o http://www.muscula.com](#) [.](#)

Funziona all'incirca così:

1. Ci registriamo al servizio e inseriamo il pezzo di codice JS (o lo URL allo script) che ci viene fornito.

- Quindi lo script JS imposta una funzione personalizzata `window.onerror`.
- Quando si verifica un errore, quest'ultimo invia una richiesta contenente i dettagli al servizio.
- A questo punto noi possiamo autenticarci tramite l'interfaccia web del servizio e vedere gli errori.

Riepilogo

Il costrutto `try...catch` permette la gestione degli errori al momento dell'esecuzione. Letteralmente permette di “provare” (“try”) il codice e “catturare” (“catch”) gli errori che si possono verificare.

La sintassi è:

```
try {  
    // esegui il codice  
} catch (err) {  
    // se un errore si presenta, passo qui  
    // err è l'oggetto errore (object error)  
} finally {  
    // viene eseguito in ogni caso subito dopo try/catch  
}
```

Sia la sezione `catch` che `finally` possono essere omesse, quindi i costrutti brevi `try...catch` e `try...finally` sono ugualmente validi.

L'oggetto errore ha le seguenti proprietà:

- `message` – il messaggio di errore “human-readable”.
- `name` – la stringa con il nome dell'errore (error constructor name).
- `stack` (non standard, ma ben supportato) – lo stack al momento della creazione dell'errore.

Se un oggetto errore non è necessario, possiamo ometterlo usando `catch {}` anziché `catch (err) {}`.

Possiamo anche generare un nostro errore personalizzato usando l'operatore `throw`.

Tecnicamente, l'argomento di `throw` può essere qualunque cosa, ma in genere è un oggetto errore (object error) che estende la classe integrata `Error`. Puoi leggerne di più nel prossimo capitolo.

Rethrowing è un pattern veramente importante per la gestione degli errori: un blocco `catch` in genere si aspetta e gestisce un particolare tipo di errore, quindi dovrebbe “rilanciare” (rethrow) gli errori che non è in grado di gestire.

In ogni caso, se non abbiamo `try...catch`, molti ambienti permettono d'impostare un gestore “globale” per intercettare gli errori che ci “buttano fuori”. All'interno del browser c'è `window.onerror`.

✓ Esercizi

Finally o solamente il codice?

importanza: 5

Confronta i due frammenti di codice.

1.

Il primo utilizza `finally` per eseguire il codice dopo `try...catch`:

```
try {
    lavoro lavoro
} catch (e) {
    gestione errori
} finally {
    ripulisci lo spazio di lavoro
}
```

2.

Il secondo posiziona la pulizia subito dopo il `try...catch`:

```
try {
    lavoro lavoro
} catch (e) {
    gestione errori
}

ripulisci lo spazio di lavoro
```

Abbiamo decisamente bisogno di ripulire dopo il lavoro, sia che si verifichi un errore o meno.

Esiste un vantaggio nell'usare `finally` o ambedue i frammenti di codice sono equivalenti? Se c'è qualche vantaggio, allora fornisci un esempio di quanto sia importante.

[Alla soluzione](#)

Errori personalizzati, estendere la classe Error

Quando sviluppiamo qualcosa, spesso nasce la necessità di avere delle classi di errore che riflettano eventi specifici che possono accadere nei nostri tasks. Per errori durante le operazioni di rete abbiamo bisogno di `HttpError`, per operazioni sul database `DbError`, per operazioni di ricerca `NotFoundError` e così via.

Le classi di errore dovrebbero supportare delle proprietà di base come `message`, `name` e, preferibilmente, `stack`. Ma possono anche avere altre proprietà, a.e. l'oggetto `HttpError` può avere una proprietà `statusCode` con valori tipo `404` o `403` o `500`.

JavaScript permette di usare `throw` con un argomento, quindi tecnicamente non è necessario che le nostre classi personalizzate ereditino da `Error`. Ma se ereditiamo, diventa possibile utilizzare `obj instanceof Error` per identificare gli oggetti di tipo errore. Quindi è meglio ereditare da esso.

Man mano che l'applicazione cresce, i nostri errori formeranno naturalmente una gerarchia. Per esempio, `HttpTimeoutError` può ereditare da `HttpError`, e così via.

Estendere “Error”

Come esempio, consideriamo una funzione `readUser(json)` che dovrebbe leggere un JSON con i dati dell’utente.

Questo è un esempio di come dovrebbe apparire un `json` valido:

```
let json = `{"name": "John", "age": 30}`;
```

Internamente, useremo `JSON.parse` che, se riceve un `json` malformato, lancia `SyntaxError`. Ma anche se il `json` è sintatticamente corretto, questo non significa che sia un utente valido, ok? Potrebbero mancare i dati necessari. Ad esempio, potrebbe non avere le proprietà `name` e `age` che sono essenziali per i nostri utenti.

La nostra funzione `readUser(json)` non solo leggerà il JSON, ma validerà i dati. Se non ci sono i campi richiesti, o il formato è errato, allora c’è un errore. E non è un `SyntaxError`, dato che è sintatticamente corretto, ma un altro tipo di errore. Lo chiameremo `ValidationError` e creeremo una classe per esso. Un errore di questo tipo dovrebbe contenere le informazioni riguardo il campo incriminato.

La nostra classe `ValidationError` dovrebbe ereditare dalla built-in class `Error`.

Questa classe è incorporata, ma ecco il suo codice approssimativo per capire meglio come la andremo ad estendere:

```
// Il "pseudocodice" per la built-in class Error definita da JavaScript
class Error {
  constructor(message) {
    this.message = message;
    this.name = "Error"; // (differenti nomi per differenti classi di errori incorporate)
    this.stack = <call stack>; // non-standard, ma la maggior parte degli ambienti lo supporta
  }
}
```

Ora ereditiamo `ValidationError` da esso e proviamolo in azione:

```
class ValidationError extends Error {
  constructor(message) {
    super(message); // (1)
    this.name = "ValidationError"; // (2)
  }
}

function test() {
  throw new ValidationError("Whoops!");
}

try {
  test();
} catch(err) {
  alert(err.message); // Whoops!
  alert(err.name); // ValidationError
}
```

```
    alert(err.stack); // a list of nested calls with line numbers for each
}
```

Poniamo attenzione: alla linea (1) richiamiamo il costruttore genitore. JavaScript ci richiede di richiamare `super` nel costruttore figlio, quindi è obbligatorio. Il costruttore genitore impone la proprietà `message`.

Il genitore impone anche la proprietà `name` in `"Error"`, quindi nella linea (2) re-impostiamo il corretto valore.

Proviamo ad usarlo in `readUser(json)`:

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

// Utilizzo
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new ValidationError("Mancava un campo: age");
  }
  if (!user.name) {
    throw new ValidationError("Mancava un campo: name");
  }

  return user;
}

// Esempio funzionante con try..catch

try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
    alert("Invalid data: " + err.message); // Dati non validi: Mancava un campo: name
  } else if (err instanceof SyntaxError) { // (*)
    alert("JSON Syntax Error: " + err.message);
  } else {
    throw err; // errore sconosciuto, lo rilancio (**)
  }
}
```

Il blocco `try..catch` nel codice qui sopra gestisce sia il nostro `ValidationError` che l'errore `SyntaxError` lanciato da `JSON.parse`.

Poniamo particolare attenzione a come usiamo `instanceof` per verificare errori specifici nella linea (*).

Potremmo anche verificare tramite `err.name`, nel seguente modo:

```
// ...
```

```
// al posto di (err instanceof SyntaxError)
} else if (err.name == "SyntaxError") { // (*)
// ...
```

La versione con `instanceof` è sicuramente migliore, perchè in futuro andremo a estendere `ValidationError`, creando sottotipi di esso, come `PropertyRequiredError`. E il controllo `instanceof` continuerà a funzionare per le nuove classi ereditate. Quindi è a prova di futuro.

È anche importante che se `catch` incontra un errore sconosciuto, lo rilanci alla linea `(**)`. Il blocco `catch` sa solamente come gestire la validazione e gli errori di sintassi, altri tipi (ad esempio un errore di battitura nel codice o altri sconosciuti) dovrebbero fallire.

Ulteriori Eredità

La classe `ValidationError` è veramente generica. Molte cose possono andare storte. Una proprietà può essere assente o può essere in un formato sbagliato (come una stringa per `age`). Quindi creiamo una classe più concreta `PropertyRequiredError`, esattamente per le proprietà assenti. Essa conterrà le informazioni addizionali riguardo le proprietà che mancano.

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("Manca la proprietà: " + property);
    this.name = "PropertyRequiredError";
    this.property = property;
  }
}
```

```
// Uso
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new PropertyRequiredError("age");
  }
  if (!user.name) {
    throw new PropertyRequiredError("name");
  }

  return user;
}

// Esempio funzionante con try..catch

try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
```

```

    alert("Dati non validi: " + err.message); // Dati non validi: Manca una proprietà: name
    alert(err.name); // PropertyRequiredError
    alert(err.property); // name
} else if (err instanceof SyntaxError) {
    alert("JSON Syntax Error: " + err.message);
} else {
    throw err; // unknown error, rethrow it
}
}

```

La nuova classe `PropertyRequiredError` è facile da usare: dobbiamo solamente fornire il nome della proprietà: `new PropertyRequiredError(property)`. Il messaggio `message` è generato dal costruttore.

Poniamo particolare attenzione al fatto che `this.name` nel costruttore `PropertyRequiredError` è di nuovo assegnato manualmente. Questa cosa potrebbe risultare un po' noiosa – assegnare `this.name = <class name>` in ogni errore personalizzato. Possiamo evitarlo creando la nostra classe “basic error” che assegna `this.name = this.constructor.name`, quindi ereditare da questa tutti i nostri errori personalizzati.

Quindi chiamiamola `MyError`.

Qui il codice con `MyError` e altre classi personalizzate, semplificate:

```

class MyError extends Error {
    constructor(message) {
        super(message);
        this.name = this.constructor.name;
    }
}

class ValidationError extends MyError {}

class PropertyRequiredError extends ValidationError {
    constructor(property) {
        super("No property: " + property);
        this.property = property;
    }
}

// name is correct
alert( new PropertyRequiredError("field").name ); // PropertyRequiredError

```

Ora il codice degli errori personalizzati sarà più corto, specialmente `ValidationError`, dato che ci siamo sbarazzati della linea con `"this.name = . . ."` nel costruttore.

Wrapping exceptions

Lo scopo della funzione `readUser` nel codice precedente è di “leggere i dati dell’utente”. Possono accadere diverse cose durante questo processo. Per adesso abbiamo `SyntaxError` e `ValidationError`, ma in futuro la funzione `readUser` potrebbe crescere e probabilmente generare altri tipi di errore.

Il codice che richiama `readUser` dovrebbe gestire questi errori. Per ora utilizziamo diversi `if` nel blocco `catch`, che verificano la classe, ne gestiscono gli errori e rilanciano quelli sconosciuti.

Lo schema è simile al seguente:

```
try {
  ...
  readUser() // La potenziale fonte di errore
  ...
} catch (err) {
  if (err instanceof ValidationError) {
    // gestisco gli errori di validazione
  } else if (err instanceof SyntaxError) {
    // gestisco gli errori di sintassi
  } else {
    throw err; // errore sconosciuto, lo rilancio
  }
}
```

Nel codice qui sopra possiamo notare due tipi di errore, ma ce ne possono essere molti di più.

Se la funzione `readUser` genera diversi tipi di errore, allora dovremmo chiederci: vogliamo veramente controllare tutti i tipi di errore un alla volta ogni volta?

Spesso la risposta è “No”: vorremo stare tutto sommato “un livello sopra tutto questo”. A noi interessa sapere se c’è un “errore nella lettura dei dati” – perchè esattamente questo accade è spesso irrilevante (il messaggio di errore già lo descrive). O, ancora meglio, vorremo avere un modo per ottenere i dettagli dell’errore, ma solo quando ne abbiamo bisogno.

La tecnica che andiamo qui a descrivere è chiamata “wrapping exceptions”.

1. Creeremo una nuova classe `ReadError` che rappresenta un errore generico di “lettura dei dati”.
2. La funzione `readUser` catturerà gli errori di lettura che avvengono al suo interno, come `ValidationError` e `SyntaxError`, e genererà un `ReadError`.
3. L’oggetto `ReadError` terrà i riferimenti all’errore originale nella sua proprietà `cause`.

Quindi il codice che richiama `readUser` dovrà solamente controllare se si verifica un `ReadError`, e non ogni tipo di errore nella lettura dei dati. E se abbiamo la necessità di approfondire i dettagli dell’errore, lo potremo fare controllando la proprietà `cause`.

Questo è il codice che definisce `ReadError` e la dimostrazione di come usarlo in `readUser` e nel `try..catch`:

```
class ReadError extends Error {
  constructor(message, cause) {
    super(message);
    this.cause = cause;
    this.name = 'ReadError';
  }
}

class ValidationError extends Error { /*...*/ }
class PropertyRequiredError extends ValidationError { /* ... */ }
```

```

function validateUser(user) {
  if (!user.age) {
    throw new PropertyRequiredError("age");
  }

  if (!user.name) {
    throw new PropertyRequiredError("name");
  }
}

function readUser(json) {
  let user;

  try {
    user = JSON.parse(json);
  } catch (err) {
    if (err instanceof SyntaxError) {
      throw new ReadError("Syntax Error", err);
    } else {
      throw err;
    }
  }

  try {
    validateUser(user);
  } catch (err) {
    if (err instanceof ValidationError) {
      throw new ReadError("Validation Error", err);
    } else {
      throw err;
    }
  }
}

try {
  readUser('{bad json}');
} catch (e) {
  if (e instanceof ReadError) {
    alert(e);
    // L'errore originale: SyntaxError: token inaspettato nel JSON alla posizione 1
    alert("Errore originale: " + e.cause);
  } else {
    throw e;
  }
}

```

Nel codice qui sopra, `readUser` funziona esattamente come descritto – Intercetta gli errori di sintassi e di validazione e lancia l'errore `ReadError` (gli errori sconosciuti saranno rilanciati come prima).

Quindi il codice più esterno controllerà per `instanceof ReadError` e basta. Non è necessario controllare tutti i tipi di errore.

Questo approccio è chiamato “wrapping exceptions”, perché controlliamo le eccezioni di “basso livello” e le “inglobiamo” in `ReadError` che è più astratto. Questo approccio è largamente utilizzato nella programmazione ad oggetti.

Sommario

- Possiamo ereditare da `Error` e altri classi di errore incorporate. Dobbiamo fare attenzione alla proprietà `name` e non dimenticare di richiamare `super`.
- Possiamo utilizzare `instanceof` per controllare un errore particolare. Questo funziona anche con l'ereditarietà. Ma a volte abbiamo un oggetto di tipo errore che proviene da librerie di terze parti e non c'è un modo semplice per verificare queste classi. Possiamo quindi usare la proprietà `name` per fare un minimo di verifica.
- “Wrapping exceptions” è una tecnica molto usata: una funzione gestisce le eccezioni di basso livello e crea errori di alto livello anziché singoli errori di basso livello. Le eccezioni di basso livello diventano proprietà dell'oggetto, come `err.cause` nell'esempio visto, ma non è strettamente richiesto.

✓ Esercizi

Eredita da SyntaxError

importanza: 5

Crea una classe `FormatError` che eredita dalla classe incorporata `SyntaxError`.

Dovrebbe supportare le proprietà `message`, `name` e `stack`.

Esempio di esecuzione:

```
let err = new FormatError("formatting error");

alert( err.message ); // Errore nella formattazione
alert( err.name ); // FormatError
alert( err.stack ); // stack

alert( err instanceof FormatError ); // vero
alert( err instanceof SyntaxError ); // vero (poiché eredita SyntaxError)
```

[Alla soluzione](#)

Promises, async/await

Introduzione: callbacks

Utilizzeremo i metodi browser in questi esempi

Per dimostrare l'utilizzo di callbacks, promise ed altri concetti astratti, utilizzeremo alcuni metodi del browser: nello specifico, caricamento di script e semplici manipolazioni del documento.

Se questi metodi non vi sono familiari, e il loro utilizzo negli esempi vi risulta di difficile comprensione, potreste voler leggere un paio di capitolo della [prossima parte](#) del tutorial.

Anche se, proveremo a mantenere le spiegazioni più chiare possibili. Non ci sarà nulla di realmente complesso dal punto di vista del browser.

Molte funzioni vengono fornite da ambienti JavaScript che permettono di schedulare azioni *asincrone*. In altre parole, azioni che iniziano ora, ma finiranno in un secondo momento.

Ad esempio, una di queste funzioni è `setTimeout`.

Ci sono altri esempi molto utili, e.g caricamento di script e moduli (che studieremo nei successivi capitoli).

Diamo un'occhiata alla funzione `loadScript(src)`, che carica uno script dal percorso `src`:

```
function loadScript(src) {
    // creates a <script> tag and append it to the page
    // this causes the script with given src to start loading and run when complete
    let script = document.createElement('script');
    script.src = src;
    document.head.append(script);
}
```

La funzione inserisce nel documento il nuovo script creato dinamicamente `<script src="...">` con il dato `src`. Il browser eseguirà automaticamente il caricamento ed una volta terminato eseguirà lo script.

Possiamo usare la funzione in questo modo:

```
// carica ed esegue lo script
loadScript('/my/script.js');
```

La funzione è chiamata in modo asincrono, perché l'azione (il caricamento dello script) non finirà adesso ma in seguito.

Se c'è del codice sotto `loadScript(...)`, non dovrà attender fino al caricamento dello script.

```
loadScript('/my/script.js');
// il codice sotto loadScript non attende che il caricamento di loadScript sia completo
// ...
```

Ora diciamo che vogliamo eseguire il nuovo script quando carica. Probabilmente dichiarerà nuove funzioni, quindi vorremmo eseguirle.

Ma se lo facciamo immediatamente dopo la chiamata `loadScript(...)` non funzionerebbe:

```
loadScript('/my/script.js'); // lo script ha "function newFunction() {...}"  
newFunction(); // nessuna funzione!
```

Naturalmente, con buona probabilità il browser non ha avuto tempo di caricare lo script. Quindi la chiamata immediata alla nuova funzione fallirà. Allo stato attuale la funzione `loadScript` non prevede un modo di tracciare l'avvenuto caricamento. Lo script carica e poi viene eseguito, questo è quanto. Ma vorremmo sapere quando accade in modo da utilizzare nuove funzioni e variabili da quello script.

Aggiungiamo una funzione `callback` come secondo argomento a `loadScript` che dovrebbe essere eseguito una volta che lo script è stato caricato.

```
function loadScript(src, callback) {  
  let script = document.createElement('script');  
  script.src = src;  
  
  script.onload = () => callback(script);  
  
  document.head.append(script);  
}
```

Ora se volessimo chiamare nuove funzioni dallo script, dovremmo scriverlo nella callback:

```
loadScript('/my/script.js', function() {  
  // la callback viene eseguita dopo che lo script caricato  
  newFunction(); // quindi adesso funziona  
  ...  
});
```

Questa è l'idea: il secondo argomento è una funzione (solitamente anonima) che viene eseguita quando l'azione è completata.

Ecco un esempio eseguibile con un vero script:

```
function loadScript(src, callback) {  
  let script = document.createElement('script');  
  script.src = src;  
  script.onload = () => callback(script);  
  document.head.append(script);  
}  
  
loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {  
  alert(`Cool, the ${script.src} is loaded`);  
  alert(_); // funzione dichiarata nello script caricato  
});
```

Questo è lo stile di programmazione asincrona “callback-based”. Una funzione che fa qualcosa in modo asincrono dovrebbe prevedere un argomento `callback` in cui mettiamo la funzione da eseguire al completamento dell'operazione asincrona.

In questo esempio lo abbiamo fatto in `loadScript` ma, ovviamente, è un approccio generale.

Callback dentro callback

Come possiamo caricare due script sequenzialmente: prima il primo e dopo il secondo?

La soluzione naturale sarebbe quella di mettere la seconda chiamata `loadScript` all'interno della callback in questo modo:

```
loadScript('/my/script.js', function(script) {  
  
    alert(`Cool, the ${script.src} is loaded, let's load one more`);  
  
    loadScript('/my/script2.js', function(script) {  
        alert(`Cool, the second script is loaded`);  
    });  
  
});
```

Dopo che la funzione `loadScript` più esterna è completata, la callback comincia quella interna.

Ma se volessimo un altro script...?

```
loadScript('/my/script.js', function(script) {  
  
    loadScript('/my/script2.js', function(script) {  
  
        loadScript('/my/script3.js', function(script) {  
            // ...continua quando tutti gli script sono stati caricati  
        });  
  
    });  
  
});
```

Quindi, ogni nuova azione è dentro una callback. Questo va bene quando abbiamo poche azioni, ma non quando ne abbiamo molte, quindi in seguito vedremo altre alternative.

Gestione degli errori

Negli esempi precedenti non abbiamo considerato gli errori. Cosa accade se il caricamento dello script fallisce? La nostra callback dovrebbe essere in grado di gestire questa eventualità.

Ecco una versione migliorata di `loadScript` che traccia gli errori di caricamento:

```
function loadScript(src, callback) {  
    let script = document.createElement('script');  
    script.src = src;  
  
    script.onload = () => callback(null, script);  
    script.onerror = () => callback(new Error(`Errore di caricamento dello script per ${src}`));  
  
    document.head.append(script);  
}
```

Chiama `callback(null, script)` per i caricamenti con successo e `callback(error)` altrimenti.

L'utilizzo:

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // gestione dell'errore
  } else {
    // script caricato con successo
  }
});
```

Ancora una volta, la ricetta che abbiamo usato per `loadScript` è abbastanza comune. È chiamato “error-first callback” style.

La convenzione è:

1. Il primo argomento di `callback` è riservato per un errore se si verifica. In questo caso la chiamata è `callback(err)`.
2. Il secondo argomento (e quelli successivi se necessario) sono per il risultato in caso di successo. In questo caso la chiamata è `callback(null, result1, result2...)`.

Quindi la singola funzione `callback` è usata sia per riportare gli errori che per passare i risultati.

Piramide del fato (Pyramid of Doom)

Ad una prima occhiata, è un modo pratico di programmare in modo asincrono. Ed infatti lo è. Per una, forse due, chiamate annidate sembra che funzioni.

Ma per molte azioni asincrone che si susseguono una dopo l'altra avremo codice come questo:

```
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...continua dopo che tutti gli script sono caricati (*)
          }
        });
      }
    });
  }
});
```

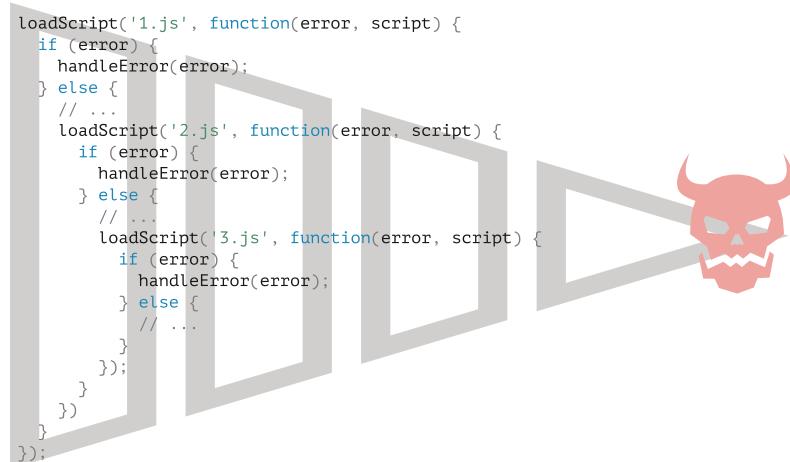
```
});
```

Nel codice sopra:

1. Carichiamo `1.js`, poi se non ci sono errori.
2. Carichiamo `2.js`, poi se non ci sono errori.
3. Carichiamo `3.js`, poi se non ci sono errori – facciamo qualcos'altro (*) .

Mano a mano che le chiamate diventano più annidate, il codice diventa più profondo e via via più complicato da gestire, specialmente se abbiamo codice reale invece di ..., che può includere più cicli, condizioni e così via.

Questo viene chiamato “callback hell” o “pyramid of doom.”



La “piramide” di chiamate annidate cresce verso destra per ogni azione asincrona. Presto la situazione sarà fuori controllo.

Per questo motivo questo modo di programmare non è molto ottimale.

Possiamo provare ad alleviare il problema rendendo ogni azione una funzione a se stante come qui:

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}
```

```

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...continua dopo che tutti gli script sono caricati(*)
  }
}

```

Visto? Fa la stessa cosa, e non ci sono annidamenti profondi perché abbiamo reso ogni azione una funzione separata di primo livello.

Funziona ma il codice sembra un foglio di lavoro diviso. È difficile da leggere e probabilmente hai notato che bisogna saltare con lo sguardo tra i vari pezzi quando lo si legge. Non è conveniente, in particolare se il lettore non è familiare con il codice e non sa dove saltare con lo sguardo.

Inoltre, le funzioni chiamate `step*` sono tutte usate una sola volta, sono create solo per evitare la “pyramid of doom.” Nessuno le riutilizzerà al di fuori della catena di azioni. Quindi abbiamo un po’ di inquinamento del namespace.

Ci piacerebbe avere qualcosa di meglio.

Fortunatamente, ci sono altri modi di evitare queste piramidi. Uno dei modi migliori è di usare le “promise” descritte nel capitolo successivo.

Promise

Immagina di essere un cantante famoso, ed i fan ti chiedono giorno e notte del tuo nuovo singolo.

Per avere un pò di sollievo, prometti di inviarglielo quando sarà pubblicato. Fornisci ai tuoi fan una lista. Loro possono compilarla con la loro email, quindi quando la funzione sarà disponibile, tutti gli iscritti la riceveranno. E anche se qualcosa dovesse andare storto, ad esempio un incendio nello studio, che ti impedisce di pubblicare la canzone, i fan verranno comunque notificati.

Tutti sono felici: tu, perché le persone non ti disturbano più, ed i fan, poiché in questo modo non si perderanno nessuna canzone.

1. Un “codice produttore” (producing code) che fa qualcosa e che richiede tempo. Per esempio, il codice che carica uno script remoto. Questo è un “cantante”.
2. Un “codice consumatore” (consuming code) che vuole il risultato del “codice produttore” una volta che è pronto. Molte funzioni possono aver bisogno di questo risultato. Queste sono i “fan”.
3. Una *promise* è uno speciale oggetto JavaScript che collega il “codice produttore” con il “codice consumatore”. Nei termini della nostra analogia: questa è “la lista abbonamenti”. Il “codice produttore” si prende tutto il tempo necessario a produrre il risultato promesso, e la “promise” rende il risultato disponibile per tutto il codice iscritto quando è pronto.

L’analogia non è completamente accurata, perché le promise di JavaScript sono più complesse di una semplice lista di abbonamenti: hanno altre caratteristiche e limiti. Ma va bene per iniziare.

La sintassi del costruttore per un oggetto promise è:

```

let promise = new Promise(function(resolve, reject) {

```

```
// esecutore (il codice produttore, "cantante")
});
```

La funzione passata a `new Promise` è chiamata `esecutore (executor)`. Quando la promise è creata, questa funzione esecutore viene eseguita automaticamente. Contiene il codice produttore, che eventualmente produrrà un risultato. Nei termini dell'analogia precedente: l'esecutore è il "cantante".

I suoi argomenti `resolve` e `reject` sono delle callback fornite da JavaScript stesso. Il nostro codice sta solamente dentro l'esecutore.

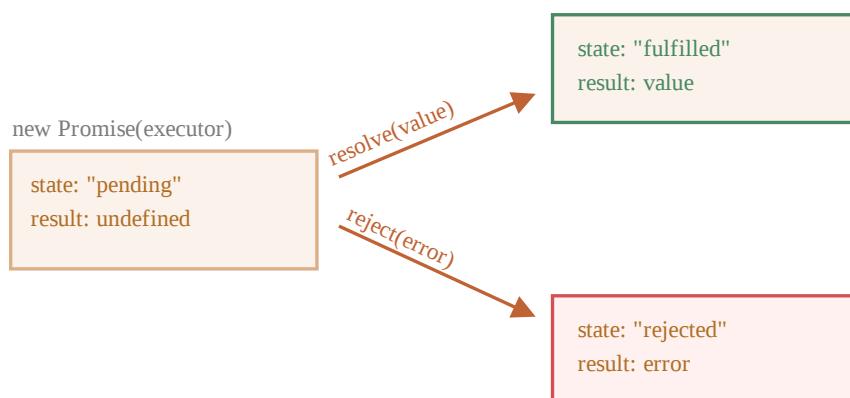
- `resolve(value)` — se il processo termina correttamente, col risultato `value`.
- `reject(error)` — se si verifica un errore, `error` è l'oggetto errore.

Ricapitolando: l'esecutore parte automaticamente e tenta di eseguire un compito. Quando l'esecuzione termina, viene invocato `resolve` in caso di successo, oppure `reject` in caso di errore.

L'oggetto `promise` restituito ha le seguenti proprietà interne:

- `state` — inizialmente "pending", poi cambia in "fulfilled" se viene invocato `resolve` o in "rejected" se viene invocato `reject`.
- `result` — inizialmente `undefined`, poi cambia in `value` se viene invocato `resolve(value)` o in `error` se viene invocato `reject(error)`.

Quindi l'esecutore, alla fine, mette la promise in uno di questi stati:



Più avanti vedremo come questi cambiamenti diventano noti ai "fan".

Qui vediamo un esempio di costruzione di una Promise ed un semplice esecutore ritardato (tramite `setTimeout`):

```
let promise = new Promise(function(resolve, reject) {
  // la funzione è eseguita automaticamente quando la promise è costruita

  // dopo 1 secondo segnala che il lavoro è fatto con risultato "done"
  setTimeout(() => resolve("done"), 1000);
});
```

Possiamo vedere due cose eseguendo il codice sopra:

1. L'esecutore è chiamato automaticamente ed immediatamente (da `new Promise`).
2. L'esecutore riceve due argomenti: `resolve` e `reject` — queste funzioni sono predefinite dal motore JavaScript. Quindi non abbiamo bisogno di crearle. Dovremo invece scrivere l'esecutore per chiamarle quando è il momento.

Dopo un secondo di “elaborazione” l'esecutore chiama `resolve("done")` per produrre il risultato. Questo cambia lo stato dell'oggetto `promise`:

`new Promise(executor)`

state: "pending"
result: undefined

`resolve("done")`

state: "fulfilled"
result: "done"

Questo era un esempio di un lavoro completato con successo, una “fulfilled promise”.

Ed ora un esempio dell'esecutore respingere (rejecting) la promise con un errore:

```
let promise = new Promise(function(resolve, reject) {  
  // dopo 1 secondo segnala che il lavoro è finito con un errore  
  setTimeout(() => reject(new Error("Whoops!")), 1000);  
});
```

La chiamata a `reject(...)` sposta lo stato della Promise a "rejected" :

`new Promise(executor)`

state: "pending"
result: undefined

`reject(error)`

state: "rejected"
result: error

Per riassumere, l'esecutore dovrebbe svolgere un lavoro (di solito qualcosa che richiede tempo) e successivamente chiamare `resolve` o `reject` per cambiare lo stato dell'oggetto Promise corrispondente.

La Promise che è soddisfatta (resolved) o respinta (rejected) è chiamata “ferma (settled)”, al contrario di Promise “in attesa (pending)”.

i Può esserci solo un risultato (result) o un errore (error)

L'esecutore può chiamare solo un `resolve` o un `reject`. Il cambiamento di stato della promise è definitivo.

Tutte le chiamate successive a 'resolve' o 'reject' sono ignorate:

```
let promise = new Promise(function(resolve, reject) {
  resolve("done");

  reject(new Error(...)); // ignorato
  setTimeout(() => resolve(...)); // ignorato
});
```

L'idea è che il lavoro fatto dall'esecutore può avere solo un risultato o un errore.

Inoltre, 'resolve'/'reject' prevedono solo un argomento (o nessuno) ed ignoreranno argomenti successivi.

i Reject con oggetti Error

Nel caso in cui qualcosa vada male, possiamo chiamare `reject` con qualunque tipo di argomento (come `resolve`). Ma è raccomandato utilizzare gli oggetti `Error` (o oggetti che estendono `Error`). La ragione di questo sarà presto evidente.

i Chiamare immediatamente `resolve` / `reject`

In pratica, un esecutore di norma fa qualcosa in modo asincrono e chiama `resolve/reject` dopo un po' di tempo, ma non è obbligato a farlo. Possiamo anche chiamare `resolve` o `reject` immediatamente, come sotto:

```
let promise = new Promise(function(resolve, reject) {
  // non prendiamo il nostro tempo per svolgere il lavoro
  resolve(123); // diamo immediatamente il risultato: 123
});
```

Per esempio, questo può accadere quando iniziamo a fare un lavoro ma poi vediamo che tutto è già stato completato.

Questo va bene. Abbiamo immediatamente una Promise soddisfatta, non c'è niente di sbagliato in questo.

i `state` e `result` sono interni

Le proprietà `state` e `result` dell'oggetto Promise sono interne. Non possiamo accedervi direttamente dal nostro "codice consumatore". Possiamo usare i metodi `.then` / `.catch` / `.finally` per questo. Questi metodi sono descritti sotto.

Consumatori (consumers): `then`, `catch`, `finally`

Un oggetto Promise fa da collegamento tra l'esecutore (il "codice produttore" o "cantante") e le funzioni consumatore (i "fan"), che riceveranno il risultato o un errore. Le funzioni consumatori possono essere registrate (subscribed) usando i metodi `.then`, `.catch` e `.finally`.

then

Il più importante e fondamentale è `.then`.

La sintassi è:

```
promise.then(  
  function(result) { /* gestisce un risultato in caso di successo */ },  
  function(error) { /* gestisce un errore */ }  
)
```

Il primo argomento di `.then` è una funzione che esegue quando una promise viene risolta, e ne riceve il risultato.

Il secondo argomento di `.then` è una funzione che esegue quando una promise viene rifiutata e riceve l'errore.

Per esempio, ecco una reazione ad una promise soddisfatta:

```
let promise = new Promise(function(resolve, reject) {  
  setTimeout(() => resolve("fatto!"), 1000);  
});  
  
// resolve esegue la prima funzione in in .then  
promise.then(  
  result => alert(result), // mostra "fatto!" dopo 1 secondo  
  error => alert(error) // non viene eseguito  
)
```

La prima funzione è stata eseguita.

E in caso di rifiuto (rejection) – la seconda:

```
let promise = new Promise(function(resolve, reject) {  
  setTimeout(() => reject(new Error("Whoops!")), 1000);  
});  
  
// reject runs the second function in .then  
promise.then(  
  result => alert(result), // non viene eseguita  
  error => alert(error) // mostra "Error: Whoops!" dopo 1 secondo  
)
```

Se siamo interessati solo ai completamenti con successo, allora possiamo fornire solo una funzione come argomento a `.then`:

```
let promise = new Promise(resolve => {  
  setTimeout(() => resolve("fatto!"), 1000);  
})
```

```
promise.then(alert); // mostra "fatto!" dopo 1 secondo
```

catch

Se siamo interessati solo agli errori, allora possiamo usare `null` come primo argomento:
`.then(null, errorHandlingFunction)`. Oppure possiamo usare
`.catch(errorHandlingFunction)`, che è esattamente lo stesso:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promise.catch(alert); // mostra "Error: Whoops!" dopo 1 secondo
```

La chiamata `.catch(f)` è completamente analoga a `.then(null, f)`, è solo un'abbreviazione.

finally

Proprio come c'è la clausola `finally` in un regolare `try {...} catch {...}`, c'è `finally` nelle promise.

La chiamata `.finally(f)` è simile a `.then(f, f)` nel senso che viene sempre eseguita quando la promise è ferma (settled): che sia soddisfatta o respinta.

`finally` è un buon handler per fare pulizia, ad esempio fermare i nostri indicatori di caricamento, dato che non sono più necessari, indipendentemente dall'esito.

Ad esempio:

```
new Promise((resolve, reject) => {
  /* fa qualcosa che prende tempo, poi chiama resolve/reject */
})
  // viene eseguito quando la promise è ferma (settled), non conta se con successo o no
  .finally(() => ferma l'indicatore di caricamento)
  .then(result => show result, err => mostra l'errore)
```

Tuttavia non è esattamente un alias. Ci sono diverse importanti differenze:

1. Un handler `finally` non ha argomenti. In `finally` non sappiamo se la promise ha successo oppure no. Questo va bene, dato che il nostro compito è solitamente quello di eseguire procedure di finalizzazione “generiche”.
2. Finally passa risultati ed errori al prossimo handler.

Per esempio, qui il risultato è passato da `finally` a `then`:

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve("result"), 2000)
})
  .finally(() => alert("Promise ready"))
  .then(result => alert(result)); // <-- .then gestisce il risultato
```

Ed ecco un errore nella promise, passata da `finally` a `catch`:

```
new Promise((resolve, reject) => {
  throw new Error("error");
})
  .finally(() => alert("Promise ready"))
  .catch(err => alert(err)); // <-- .catch gestisce l'oggetto errore
```

Questo è molto utile, perché `finally` non è inteso per processare i risultati della promise. Quindi li passa avanti.

Parleremo di più della catena di promise ed il passaggio di risultati tra handler nel prossimo capitolo

3. Ultimo, ma non meno importante, `.finally(f)` è una sintassi più conveniente di `.then(f, f)`: non c'è bisogno di duplicare la funzione.

i Sulle promise ferme gli handler vengono eseguiti immediatamente

Se una promise è pending, gli handler `.then/catch/finally` aspettano il risultato. Altrimenti, se una promise è già ferma, vengono eseguiti immediatamente:

```
// una promise risolta immediatamente
let promise = new Promise(resolve => resolve("fatto!"));
  .catch(err => alert(err)); // <-- .catch handles the error object

promise.then(alert); // fatto! (viene mostrato in questo momento)
```

La cosa buona è: un handler `.then` è garantito per l'esecuzione sia che la promise prenda tempo o che si fermi immediatamente.

Ora, vediamo esempi più pratici di come le promise possano aiutarci a scrivere codice asincrono.

Esempio: `loadScript`

Abbiamo la funzione `loadScript` per caricare uno script dal capitolo precedente.

Ecco la variante basata sulle callback, giusto per ricordarcene:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for ${src}`));

  document.head.append(script);
}
```

Riscriviamola usando le Promise.

La nuova funzione `loadScript` non richiederà una callback. Invece, creerà e ritornerà un oggetto Promise che risolve quando il caricamento è completo. Il codice esterno può aggiungervi handler (subscribing functions) usando `.then`:

```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Errore di caricamento dello script per: ${src}`));

    document.head.append(script);
  });
}
```

Usage:

```
let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js");

promise.then(
  script => alert(`#${script.src} is loaded!`),
  error => alert(`Error: ${error.message}`)
);

promise.then(script => alert('Un altro handler per fare qualcosa\'altro'));
```

Possiamo immediatamente vedere alcuni benefit su pattern basato sulle callback:

Promises	Callbacks
Le promise ci permettono di fare le cose nell'ordine naturale. Prima, eseguiamo <code>loadScript(script)</code> , e poi (<code>.then</code>) scriviamo cosa fare con il risultato.	Dobbiamo avere una funzione <code>callback</code> a nostra disposizione quando chiamiamo <code>loadScript(script, callback)</code> . In altre parole, dobbiamo sapere cosa fare con il risultato prima che <code>loadScript</code> sia chiamato.
Possiamo chiamare <code>.then</code> su una Promise quante volte vogliamo. Ciascuna volta, stiamo aggiungendo un nuovo "fan", una nuova funzione iscritta (subscribing function), alla "lista degli abbonamenti (subscription list)". Maggiori informazioni a tal proposito nel prossimo capitolo: Concatenamento di promise (promise chaining) .	Ci può essere solo una callback.

Quindi le Promise ci offrono un flusso migliore e maggiore flessibilità. Ma c'è di più. Lo vedremo nei prossimi capitoli

✓ Esercizi

Ri-risolvere (re-resolve) una promise?

Qual è l'output del codice sotto?

```
let promise = new Promise(function(resolve, reject) {
  resolve(1);

  setTimeout(() => resolve(2), 1000);
});

promise.then(alert);
```

[Alla soluzione](#)

Delay with a promise

The built-in function `setTimeout` uses callbacks. Create a promise-based alternative.

The function `delay(ms)` should return a promise. That promise should resolve after `ms` milliseconds, so that we can add `.then` to it, like this:

```
function delay(ms) {
  // your code
}

delay(3000).then(() => alert('runs after 3 seconds'));
```

[Alla soluzione](#)

Animated circle with promise

Rewrite the `showCircle` function in the solution of the task [Animate il cerchio con callback](#) so that it returns a promise instead of accepting a callback.

The new usage:

```
showCircle(150, 150, 100).then(div => {
  div.classList.add('message-ball');
  div.append("Hello, world!");
});
```

Take the solution of the task [Animate il cerchio con callback](#) as the base.

[Alla soluzione](#)

Concatenamento di promise (promise chaining)

Ritorniamo al problema di cui abbiamo parlato in [Introduzione: callbacks](#): abbiamo una sequenza di task asincroni da essere completati uno dopo l'altro. Per esempio caricare script. Cosa possiamo fare per programmarla bene?

Le promise ci danno un po' di ricette per farlo.

In questo capitolo copriremo il concatenamento di promise (promise chaining).

Il concatenamento di promise ha questo aspetto:

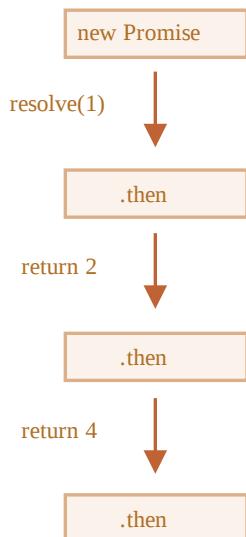
```
new Promise(function(resolve, reject) {  
  setTimeout(() => resolve(1), 1000); // (*)  
}).then(function(result) { // (**)  
  alert(result); // 1  
  return result * 2;  
}).then(function(result) { // (***)  
  alert(result); // 2  
  return result * 2;  
}).then(function(result) {  
  alert(result); // 4  
  return result * 2;  
});
```

L'idea è che il risultato viene passato attraverso la catena di gestori (handler) `.then`.

Il flusso dell'esempio è:

1. La promise iniziale è risolta (resolves) in 1 secondo `(*)`,
2. Poi (then) il `.then` viene chiamato `(**)`.
3. Il valore che ritorna è passato al prossimo gestore (handler) `.then` `(***)`
4. ...e così via.

Mano a mano che il risultato viene passato attraverso la catena di gestori (handler), possiamo vedere una sequenza di chiamate `alert`: `1 → 2 → 4`.



Tutto questo funziona, perché una chiamata a `promise.then` ritorna una promise, in questo modo possiamo chiamare il `.then` sulla promise ritornata.

Quando un gestore (handler) ritorna un valore, questo diventa il risultato della promise con il quale sarà chiamato il prossimo `.then`.

Un classico errore da newbie: tecnicamente possiamo anche aggiungere diversi `.then` ad una sola promise. Questo non è il concatenamento(chaining).

Per esempio:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
});

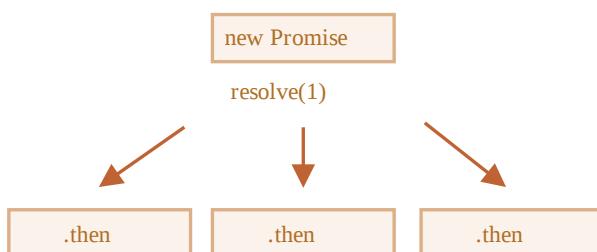
promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});
```

Quello che abbiamo fatto è giusto aggiungere diversi gestori (handler) ad una promise. I gestori (handler) si passano il risultato tra loro, al contrario lo processano indipendentemente.

Ecco una figura (da paragonare con il concatenamento di sopra):



Tutti i `.then` sulla stessa promise ricevono lo stesso risultato – il risultato della promise. Così nel codice sopra tutti gli `alert` mostrano lo stesso: `1`.

Nella pratica raramente avremo bisogno di molti gestori (handler) per la stessa promise. Il concatenamento (chaining) è usato molto più spesso.

Ritornare promise

Normalmente, il valore ritornato da un gestore (handler) `.then(handler)` è passato immediatamente a quello successivo. Ma esiste un'eccezione.

Se il valore ritornato è una promise, allora l'esecuzione è sospesa fino a quando la promise è ferma (settled). Dopo di ciò, il risultato della promise viene passato al prossimo gestore (handler) `.then`.

Per esempio:

```

new Promise(function(resolve, reject) {

    setTimeout(() => resolve(1), 1000);

}).then(function(result) {

    alert(result); // 1

    return new Promise((resolve, reject) => { /* (*) */
        setTimeout(() => resolve(result * 2), 1000);
    });
});

).then(function(result) { /* (**) */

    alert(result); // 2

    return new Promise((resolve, reject) => {
        setTimeout(() => resolve(result * 2), 1000);
    });
};

).then(function(result) {

    alert(result); // 4

});

```

Qui il primo `.then` mostra `1` e ritorna `new Promise(...)` nella linea `(*)`. Dopo un secondo la promise ritorata risolve (`resolves`), ed il risultato (l'argomento di `resolve`, che è `result*2`) viene passato al secondo gestore (`handler`) `.then` nella linea `(**)`. Infine mostra `2` e fa la stessa cosa.

Quindi l'output è ancora `1 → 2 → 4`, ma ora con un secondo di ritardo tra le chiamate `alert`.

Ritornare le promise ci permette di creare una catena di azioni asincrone.

Esempio: loadScript

Usiamo questa feature con `loadScript`, definita nel [capitolo precedente](#), per caricare gli script uno ad uno, in sequenza:

```

loadScript("/article/promise-chaining/one.js")
    .then(function(script) {
        return loadScript("/article/promise-chaining/two.js");
    })
    .then(function(script) {
        return loadScript("/article/promise-chaining/three.js");
    })
    .then(function(script) {
        // usiamo le funzioni dichiarate negli script
        // per mostrare che hanno effettivamente caricato
        one();
        two();
        three();
    });

```

Questo codice può essere reso un po' più corto con le funzioni a freccia:

```
loadScript("/article/promise-chaining/one.js")
  .then(script => loadScript("/article/promise-chaining/two.js"))
  .then(script => loadScript("/article/promise-chaining/three.js"))
  .then(script => {
    // scripts are loaded, we can use functions declared there
    one();
    two();
    three();
});
```

Qui ogni chiamata `loadScript` ritorna una promise, ed il prossimo `.then` viene eseguito quando la promise risolve (resolves). Poi inizia il caricamento del prossimo script. Così gli script vengono caricati uno dopo l'altro.

Possiamo aggiungere più azioni asincrone alla catena. È da notare che il codice rimane "piatto", cresce verso il basso, non verso destra. Non ci sono segni di "pyramid of doom".

È da notare che tecnicamente possiamo aggiungere `.then` direttamente ad ogni `loadScript`, come qui:

```
loadScript("/article/promise-chaining/one.js").then(script1 => {
  loadScript("/article/promise-chaining/two.js").then(script2 => {
    loadScript("/article/promise-chaining/three.js").then(script3 => {
      // this function has access to variables script1, script2 and script3
      one();
      two();
      three();
    });
  });
});
```

Questo codice fa lo stesso: carica 3 script in sequenza. Ma "cresce verso destra". Così abbiamo lo stesso problema che con le callback.

Le persone che iniziano ad usare le promise spesso non sono a conoscenza del concatenamento (chaining), così le scrivono in questo modo. In generale, è preferito il concatenamento (chaining).

A volte va bene scrivere `.then` direttamente, perché la funzione annidata abbia accesso allo scope esterno. Nell'esempio sopra la callback più annidata ha accesso a tutte le variabili `script1`, `script2`, `script3`. Ma è un'eccezione più che una regola.

Thenables

Per essere precisi, `.then` può ritornare un qualsiasi oggetto “thenable”, che verrà trattato nella stessa maniera di una promise.

Un oggetto “thenable” è un qualsiasi oggetto con un metodo `.then`.

L’idea è che librerie di terze parti possano implementare oggetti “promise compatibili” per conto loro. Possono avere un insieme esteso di metodi, ma anche essere compatibili con le promise native, poiché implementano `.then`.

Ecco un esempio di un oggetto “thenable”:

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve); // function() { native code }
    // resolve with this.num*2 after the 1 second
    setTimeout(() => resolve(this.num * 2), 1000); // (**)
  }
}

new Promise(resolve => resolve(1))
  .then(result => {
    return new Thenable(result); // (*)
  })
  .then(alert); // shows 2 after 1000ms
```

JavaScript controlla l’oggetto ritornato dall’handler `.then` nella linea `(*)`: Se ha un metodo chiamabile chiamato `then`, quindi chiama quel metodo passandogli come argomenti le funzioni native `resolve`, `reject` (in modo simile all’esecutore) ed aspetta fino a quando una delle due viene chiamata. Nell’esempio sopra `resolve(2)` è chiamata dopo un secondo `(**)`. Poi il risultato viene passato giù nella catena.

Questa feature permette di integrare oggetti custom con le catene di promise senza dover estendere `Promise`.

Esempio più grande: `fetch`

Nella programmazione frontend le promise sono spesso usate per le richieste di rete. Vediamone un esempio esteso.

Useremo il metodo `fetch` per caricare le informazioni sull’utente dal server remoto. Il metodo è abbastanza complesso, ha diversi parametri opzionali che studieremo in [separate chapters](#), ma l’utilizzo base è semplice:

```
let promise = fetch(url);
```

Questo fa una richiesta di rete all’`url` e ritorna una promise. La promise risolve (`resolves`) con un oggetto `response` appena il server remoto risponde con gli header, ma *prima che la risposta*

completa sia scaricata.

Per leggere la risposta completa, dovremo chiamare un metodo `response.text()`: questo metodo ritorna una promise che risolve (resolves) quando tutto il testo è scaricato dal server remoto, con questo come risultato.

Il codice sotto fa una richiesta ad `user.json` e carica il suo testo dal server:

```
fetch('/article/promise-chaining/user.json')
// il then sotto viene eseguito quando il server remoto risponde
.then(function(response) {
  // response.text() ritorna una nuova promise che risolve il testo completo della risposta
  // quando abbiamo finito di scaricarlo
  return response.text();
})
.then(function(text) {
  // ...ed ecco il contenuto del file remoto
  alert(text); // {"name": "iliakan", isAdmin: true}
});
```

C'è anche un metodo `response.json()` che legge i dati remoti e li parsa come JSON. Nel nostro caso è ancora più conveniente, quindi usiamolo.

Useremo anche le funzioni a freccia per brevità:

```
// come sopra, ma response.json() parsa il contenuto remoto come JSON
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => alert(user.name)); // iliakan, got user name
```

Ora facciamo qualcosa con l'utente caricato.

Per esempio possiamo fare un'altra richiesta a GitHub, caricare il profilo utente e mostrare l'avatar:

```
// Fa una richiesta per user.json
fetch('/article/promise-chaining/user.json')
  // Carica la risposta come json
  .then(response => response.json())
  // Fa una richiesta a GitHub
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  // Carica la risposta come json
  .then(response => response.json())
  // Mostra l'immagine dell'avatar (githubUser.avatar_url) per tre secondi (magari la2)
  .then(githubUser => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => img.remove(), 3000); // (*)
  });
});
```

Il codice funziona, possiamo vedere i dettagli nei commenti, ma dovrebbe essere abbastanza autodescrittivo. Tuttavia, c'è un potenziale errore, un errore tipico di chi inizia ad utilizzare le promise.

Guardiamo la linea `(*)`: come possiamo fare qualcosa *dopo* che l'avatar ha finito di essere mostrato e viene rimosso? Per esempio, ci piacerebbe mostrare un form per editare quell'utente o fare qualcos'altro. Allo stato attuale, non c'è modo.

Per mantenere la catena estensibile, dobbiamo ritornare una promise che si risolve (*resolves*) dopo che l'avatar finisce di mostrare.

Come qui:

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise(function(resolve, reject) { // (*)
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser); // (**)
    }, 3000);
  }))
  // viene eseguito dopo 3 secondi
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));
```

Ora subito dopo che `setTimeout` esegue `img.remove()`, chiama `resolve(githubUser)`, passando il controllo al prossimo `.then` nella catena e passando avanti i dati dell'utente.

Come regola, un'azione asincrona dovrebbe sempre ritornare una promise.

Questo rende possibile pianificare azioni successive. Anche se non abbiamo in piano di estendere la catena adesso, potremmo averne bisogno in seguito.

In fine, possiamo dividere il codice in funzioni riutilizzabili:

```
function loadJson(url) {
  return fetch(url)
    .then(response => response.json());
}

function loadGitHubUser(name) {
  return loadJson(`https://api.github.com/users/${name}`);
}

function showAvatar(githubUser) {
  return new Promise(function(resolve, reject) {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);
  })
}
```

```

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  });

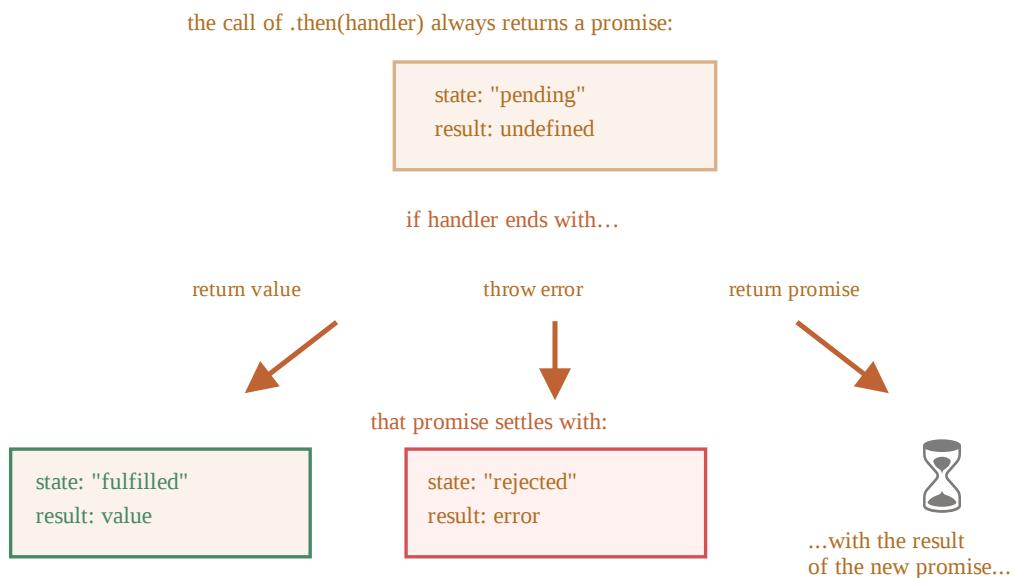
// Usiamole:
loadJson('/article/promise-chaining/user.json')
  .then(user => loadGithubUser(user.name))
  .then(showAvatar)
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));
// ...

```

Riassunto

Se un handler `.then` (o `catch/finally`, non importa) ritorna una promise, il resto della catena aspetta fino a quando la promise si ferma (settles). Quando lo fa, il suo risultato (o errore) è passato avanti.

Ecco un disegno esteso:



✓ Esercizi

Promise: then versus catch

Questi pezzi di codice sono uguali? In altre parole, si comportano nello stesso modo in ogni circostanza, per ogni funzione handler?

```
promise.then(f1).catch(f2);
```

Versus:

```
promise.then(f1, f2);
```

Gestione degli errori con le promise

Le azioni assincrone a volte possono fallire: in caso di errore la promise corrispondente viene respinta (rejected). Per esempio, `fetch` fallisce se il server remoto non è disponibile. Possiamo usare `.catch` per gestire gli errori (rejections).

Il concatenamento delle Promise è ottimo sotto questo aspetto. Quando una promise viene rifiutata (rejects), il controllo passa al gestore del rifiuto (rejection handler) più vicino nella catena. Questo è molto conveniente.

Per esempio, nel codice sotto l'URL è errato (no such server) e `.catch` gestisce l'errore:

```
fetch('https://no-such-server.blabla') // viene respinta (rejects)
  .then(response => response.json())
  .catch(err => alert(err)) // TypeError: failed to fetch (il testo può variare)
```

Oppure, forse, è tutto a posto con il server, ma la risposta non è JSON valido:

```
fetch('/') // fetch funziona bene adesso, il server risponde con successo
  .then(response => response.json()) // viene respinta (rejects): the page is HTML, not a valid JSON
  .catch(err => alert(err)) // SyntaxError: Unexpected token < in JSON at position 0
```

Il modo più facile di catturare (catch) è di mettere `.catch` alla fine della catena:

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise((resolve, reject) => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  }))
  .catch(error => alert(error.message));
```

Normalmente, `.catch` non viene eseguito, perché non ci sono errori. Ma se una qualsiasi delle promise sopra viene respinta (un errore di rete o JSON invalido o qualunque cosa), allora l'errore verrebbe catturato.

Try...catch implicito

Il codice di un esecutore (executor) e dei gestori (handlers) delle promise hanno un "try..catch invisible". Se si verifica un errore, viene catturato e gestito come un rigettamento (rejection).

Per esempio, questo codice:

```
new Promise((resolve, reject) => {
  throw new Error("Whoops!");
}).catch(alert); // Error: Whoops!
```

...Funziona esattamente come questo:

```
new Promise((resolve, reject) => {
  reject(new Error("Whoops!"));
}).catch(alert); // Error: Whoops!
```

Il "try..catch invisible" intorno all'esecutore (executor) cattura (catches) automaticamente l'errore e lo tratta come un rigettamento (rejection).

Questo accade non solo nell'esecutore (executor), ma anche nei suoi gestori (handlers). Se lanciamo (`throw`) dentro un gestore (handler) `.then`, questo significa una promise respinta (rejected), così il controllo salta al gestore (handler) degli errori più vicino.

Ecco un esempio:

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  throw new Error("Whoops!"); // respinge (rejects) la promise
}).catch(alert); // Error: Whoops!
```

Questo accade per tutti gli errori, non solo quelli causati dallo statement `throw`. Per esempio, un errore di programmazione:

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  blabla(); // non esiste la funzione
}).catch(alert); // ReferenceError: blabla is not defined
```

Il `.catch` finale non solo cattura (catches) i rigettamenti (rejections) esplicativi, ma anche gli errori occasionali nei gestori (handlers) .

Rethrowing

Come abbiamo già notato, `.catch` si comporta come `try..catch`. Possiamo avere tutti i gestori (handler) `.then` che vogliamo, e poi usare un solo `.catch` alla fine per gestire tutti gli errori al loro interno.

In un normale `try..catch` possiamo analizzare l'errore e magari rilanciarlo (`rethrow`) se non può essere gestito. È possibile fare lo stesso con le promise.

Se lanciamo `(throw)` dentro `.catch`, allora il controllo va al gestore (handler) più vicino. E se gestiamo l'errore e finiamo normalmente, allora continua al prossimo gestore (handler) `.then` per i casi di successo.

Nell'esempio sotto, `.catch` gestisce con successo l'errore:

```
// esecuzione: catch -> then
new Promise((resolve, reject) => {

    throw new Error("Whoops!");

}).catch(function(error) {

    alert("L'errore è gestito, continua normalmente");

}).then(() => alert("Il prossimo gestore (handler) per i casi di successo viene eseguito"));


```

Qui il blocco `.catch` finisce normalmente. Così il prossimo gestore (handler) `.then` viene chiamato.

Nell'esempio sotto vediamo l'altra situazione con `.catch`. Il gestore (handler) `(*)` cattura (catches) l'errore e non può gestirlo (e.g. sa solo come gestire `URIError`), quindi lo solleva (`throws`) di nuovo:

```
// esecuzione: catch -> catch -> then
new Promise((resolve, reject) => {

    throw new Error("Whoops!");

}).catch(function(error) { // (*) 

    if (error instanceof URIError) {
        // gestiscilo
    } else {
        alert("Non posso gestire l'errore");

        throw error; // lanciare questo o un altro errore ci fa saltare al prossimo catch
    }

}).then(function() {
    /* non viene mai eseguito */
}).catch(error => { // (**)

    alert(`Si è verificato un errore sconosciuto: ${error}`);
    // non ritornare nulla => l'esecuzione procede normalmente

});


```

Poi l'esecuzione passa dal primo `.catch` `(*)` al prossimo `(**)` giù per la catena.

Nella sezione sotto vedremo un pratico esempio di risollevamento (`rethrowing`).

Gestione degli errori di fetch

Miglioriamo la gestione degli errori per l'esempio del caricamento degli utenti.

La promise ritornata da `fetch` viene respinta (rejects) quando è impossibile fare una richiesta. Per esempio, un server remoto non è disponibile, o l'URL è malformato. Ma se il server remoto risponde con un errore 404, o anche un errore 500, allora è considerata una risposta valida.

Cosa accade se il server ritorna una pagina non JSON con un errore 500 nella linea (*)? Cosa accade se l'utente non esiste e GitHub ritorna una pagina con un errore 404 a (**)?

```
fetch('no-such-user.json') // (*)
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`)) // (**)
  .then(response => response.json())
  .catch(alert); // SyntaxError: Unexpected token < in JSON at position 0
// ...
```

Allo stato attuale, il codice prova comunque a caricare la risposta come JSON e muore con un errore di sintassi. Lo puoi vedere eseguendo l'esempio sopra, dato che il file `no-such-user.json` non esiste.

Questo non è buono, perché l'errore va semplicemente giù nella catena, senza dettagli: cosa è fallito e dove.

Quindi aggiungiamo un altro passo: dovremmo controllare la proprietà `response.status` che ha lo stato HTTP, e se non è 200, allora lanciare un errore.

```
class HttpError extends Error { // (1)
  constructor(response) {
    super(`#${response.status} for ${response.url}`);
    this.name = 'HttpError';
    this.response = response;
  }
}

function loadJson(url) { // (2)
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new HttpError(response);
      }
    })
}

loadJson('no-such-user.json') // (3)
  .catch(alert); // HttpError: 404 for .../no-such-user.json
```

1. Creiamo una classe custom per gli errori HTTP per distinguerli dagli altri tipi di errore. Inoltre, la nuova classe ha un costruttore che accetta l'oggetto `response` e lo salva nell'errore. Così il codice per la gestione degli errori sarà in grado di accedervi.

2. Dopo mettiamo insieme il codice per effettuare la richiesta e per gestire gli errori in una funzione che recupera l' url e tratta ogni stato non 200 come un errore. È conveniente, perchè spesso avremo bisogno di una logica simile.
3. Ora alert mostra un messaggio più utile e descrittivo.

Il bello di avere una nostra classe per gli errori è che possiamo facilmente verificarli nel nostro codice di gestione degli errori.

Per esempio, possiamo fare una richiesta, e poi se riceviamo un 404 – chiedere all'utente di modificare l'informazione.

Il codice sotto carica un utente con il nome da GitHub. Se non c'è l'utente, allora chiede il nome corretto:

```
function demoGithubUser() {
  let name = prompt("Inserire un nome", "iliakan");

  return loadJson(`https://api.github.com/users/${name}`)
    .then(user => {
      alert(`Full name: ${user.name}`);
      return user;
    })
    .catch(err => {
      if (err instanceof HttpError && err.response.status == 404) {
        alert("Utente inesistente, per favorlo nuovamente.");
        return demoGithubUser();
      } else {
        throw err; // (*)
      }
    });
}

demoGithubUser();
```

Notare che: .catch cattura tutti gli errori, ma “sa come gestire” solo HttpError 404. In questo caso particolare significa che non esiste l'utente, e .catch in questo caso riprova semplicemente.

Per altri errori, non ha idea di cosa possa andare storto. Magari un errore di programmazione o altro. Quindi semplicemente lo risoleva nella linea (*) .

Rigetti non gestiti (unhandled rejections)

Cosa accade se un errore non è gestito? Per esempio, dopo il risollevamento (*) nell'esempio sopra.

Oppure possiamo semplicemente dimenticarci di aggiungere un gestore (handler) dell'errore alla fine della catena, come qui:

```
new Promise(function() {
  noSuchFunction(); // Errore qui (non esiste la funzione)
})
  .then(() => {
```

```
// zero o molti handler di promise
}); // senza .catch alla fine!
```

Nel caso di un errore, lo stato della promise diventa “rejected”, e l'esecuzione dovrebbe saltare al gestore del respingimento (rejection handler). Ma negli esempi sopra non c'è questo gestore (handler). Quindi l'errore porta ad un “blocco”.

In pratica, proprio come con un normale errore non gestito, significa che qualcosa è andato terribilmente storto.

Cosa accade quando viene sollevato un errore e non viene gestito da `try..catch`? Lo script muore. Lo stesso accade con una promise rigettata che non viene gestita.

La maggior parte dei motori JavaScript tracciano queste situazioni e generano un errore globale in questo caso. Possiamo vederlo nella console.

Nel browser possiamo catturare (catch) questi errori usando `unhandledrejection`:

```
window.addEventListener('unhandledrejection', function(event) {
  // L'oggetto evento ha due proprietà speciali:
  alert(event.promise); // [object Promise] - la promise che ha causato l'errore
  alert(event.reason); // Error: Whoops! - L'oggetto errore non gestito
});

new Promise(function() {
  throw new Error("Whoops!");
}); // nessun catch per gestire l'errore
```

L'evento è parte dello [standard HTML ↗](#).

Se un errore si verifica, e non c'è nessun `.catch`, il gestore (handler) `unhandledrejection` viene lanciato, e riceve l'oggetto `event` con le informazioni riguardanti l'errore, in questo modo possiamo fare qualcosa.

Solitamente questi errori sono irrecuperabili, quindi la cosa migliore da fare è informare l'utente del problema e probabilmente riportare l'incidente al server.

In ambienti esterni al browser come Node.js ci sono altri modi simili di tracciare gli errori non gestiti.

Riepilogo

- `.catch` gestisce i respingimenti (rejections) delle promise di tutti i tipi: che sia una chiamata `reject()`, o un errore sollevato in un gestore (handler).
- Dovremmo mettere `.catch` esattamente nei posti in cui vogliamo gestire gli errori sapendo come gestirli. Il gestore (handler) dovrebbe analizzare gli errori (Le classi di errori ci sono di aiuto) e ri-sollevare (rethrow) quelli sconosciuti.
- È normale non usare `.catch` se non sappiamo come gestire gli errori (tutti gli errori sono irrecuperabili).
- In ogni caso dovremo avere i gestori (handler) dell'evento `unhandledrejection` (per il browser e quelli analoghi per gli altri ambienti), per tracciare gli errori non gestiti ed informarne l'utente (e probabilmente il nostro server), così che non accada mai che la nostra app “muoia e basta”.

Ed infine, se abbiamo l'indicatore di caricamento, allora `.finally` è un ottimo gestore (handler) per fermarlo quando il caricamento è completo:

```
function demoGithubUser() {
  let name = prompt("Enter a name?", "iliakan");

  document.body.style.opacity = 0.3; // (1) avvia l'indicatore

  return loadJson(`https://api.github.com/users/${name}`)
    .finally(() => { // (2) stop the indication
      document.body.style.opacity = '';
      return new Promise(resolve => setTimeout(resolve)); // (*)
    })
    .then(user => {
      alert(`Full name: ${user.name}`);
      return user;
    })
    .catch(err => {
      if (err instanceof HttpError && err.response.status == 404) {
        alert("Utente inesistente, per favorlo nuovamente.");
        return demoGithubUser();
      } else {
        throw err;
      }
    });
}

demoGithubUser();
```

Qui nella linea (1) indichiamo il caricamento oscurando il documento. Il metodo non conta, avremmo potuto usare qualunque altro tipo di indicazione.

Quando la promise è ferma (settled), che sia un fetch con successo o un errore, `finally` viene lanciato nella linea (2) ferma l'indicatore.

C'è un piccolo trucco per i browser (*) nel ritornare una promise con timeout zero da `finally`. Questo perché alcuni browser (come Chrome) hanno bisogno "di un po' di tempo" fuori dai gestori (handlers) per diegnare cambiamenti al documento. Questo assicura che l'indicazione è visivamente ferma prima di andare avanti nella catena.

✓ Esercizi

Error in setTimeout

Che cosa pensi? Il `.catch` sarà eseguito? Spiega la tua risposta.

```
new Promise(function(resolve, reject) {
  setTimeout(() => {
    throw new Error("Whoops!");
  }, 1000);
}).catch(alert);
```

[Alla soluzione](#)

Promise API

Esistono 6 metodi statici nella classe `Promise`. Qui copriremo rapidamente il loro casi d'uso.

Promise.all

Diciamo che vogliamo eseguire molte promise in parallelo, e aspettare che siano tutte pronte.

Per esempio, scaricare da diversi URL in parallelo e processare il contenuto quando abbiamo finito con tutti.

Ecco a cosa serve `Promise.all`.

La sintassi è:

```
let promise = Promise.all([...promises...]);
```

`Promise.all` accetta un array di promise (teoricamente si può usare qualsiasi iterabile, ma solitamente si usa un array) e ritorna una nuova promise.

La nuova promise si risolve quando tutte le promise elencate vengono risolte, e l'array dei loro risultati diventa il risultato finale.

Per esempio, il `Promise.all` sotto si ferma (settles) dopo 3 secondi, ed il suo risultato è un array `[1, 2, 3]`:

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000)) // 3
]).then(alert); // 1,2,3 quando le promise sono pronte: ogni promise contribuisce con un membro
```

È da notare che l'ordine relativo rimane lo stesso. Anche se la prima promise prendesse il tempo più lungo per risolversi (`resolve`), il suo risultato sarà sempre il primo nell'array dei risultati.

Un trucco comune consiste nel mappare array of di dati da lavorare in un array di promise, per poi avvolgerli (to wrap) in `Promise.all`.

Per esempio, se abbiamo un array di URL, possiamo scaricarli tutti così:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];

// mappiamo tutti gli url con la promise ritornata da fetch
let requests = urls.map(url => fetch(url));

/// Promise.all attende fino a quando tutti i job sono risolti (resolved)
Promise.all(requests)
  .then(responses => responses.forEach(
    response => alert(`${response.url}: ${response.status}`)
  ));
```

Un esempio migliore in cui scarichiamo informazioni utente per un array di utenti di GitHub in base al loro nome (potremmo scaricare una matrice di merci in base ai rispettivi id, la logica è la stessa)

```
let names = ['iliakan', 'remy', 'jeresig'];

let requests = names.map(name => fetch(`https://api.github.com/users/${name}`));

Promise.all(requests)
  .then(responses => {
    // tutte le risposte sono pronte, possiamo mostrare i loro codici di stato HTTP
    for(let response of responses) {
      alert(`${response.url}: ${response.status}`); // mostra 200 per ogni url
    }

    return responses;
})
// mappa l'array di risposte in un array di response.json() per leggere il loro contenuto
  .then(responses => Promise.all(responses.map(r => r.json())))
// è stato fatto il parsing JSON di tutte le risposte JSON: "users" è l'array con i risultati
  .then(users => users.forEach(user => alert(user.name)));
```

Se una qualsiasi delle promise è respinta (rejected), `Promise.all` viene immediatamente respinta (rejects) con l'errore.

Per esempio:

```
Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).catch(alert); // Error: Whoops!
```

Qui la seconda promise viene respinta (rejects) in due secondi. Questo porta al rigetto immediato di `Promise.all`, così `.catch` viene eseguito: l'errore del rigetto diventa il risultato di tutto `Promise.all`.

⚠️ In caso di errore, le altre promise vengono ignorate

Se una promise è respinta (rejects), `Promise.all` è immediatamente respinto, dimenticando completamente delle altre nella lista. I loro risultati sono ignorati.

Per esempio, se ci sono molte chiamate `fetch`, come nell'esempio sopra, ed una di esse fallisce, le altre continueranno ad essere eseguite, ma `Promise.all` le ignorerà. Probabilmente poi si fermeranno (settle), ma il loro risultato sarà ignorato.

`Promise.all` non fa niente per cancellarle, perché nelle promise non esiste il concetto di "cancellazione". In [un altro capitolo](#) copriremo `AbortController` il cui scopo è aiutarci con questo, ma non è una parte delle API Promise.

i `Promise.all(...)` accetta oggetti non-promise in un `iterable`

Normalmente, `Promise.all(...)` accetta un iterable (nella maggior parte dei casi un array) di promises. Ma se uno qualsiasi di questi oggetti non è una promise, viene “avvolto” (wrapped) in `Promise.resolve`.

Per esempio, qui i risultati sono `[1, 2, 3]`:

```
Promise.all([
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(1), 1000)
  }),
  2, // trattato come Promise.resolve(2)
  3 //trattato come Promise.resolve(3)
]).then(alert); // 1, 2, 3
```

Così siamo in grado di passare valori non-promise a `Promise.all` quando conveniente.

Promise.allSettled

⚠️ Aggiunta di recente

Questa funzionalità è stata aggiunta di recente al linguaggio. Vecchi browsers potrebbero richiedere polyfills.

`Promise.all` viene respinto interamente se una sola delle promise viene respinta. Questo è buono in alcuni casi, quando abbiamo bisogno di *tutti* i risultati per proseguire:

```
Promise.all([
  fetch('/template.html'),
  fetch('/style.css'),
  fetch('/data.json')
]).then(render); // il metodo render ha bisogno di tutti i risultati
```

`Promise.allSettled` aspetta che tutte le promise siano ferme (settled): anche se una viene respinta (rejects), aspetta per le altre. L'array risultante ha:

- `{status:"fulfilled", value:result}` per le risposte con successo,
- `{status:"rejected", reason:error}` per gli errori.

Per esempio, ci piacerebbe scaricare le informazioni su diversi utenti. Anche se una richiesta fallisce, siamo interessati alle altre .

Usiamo `Promise.allSettled`:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://no-such-url'
];
```

```

Promise.allSettled(urls.map(url => fetch(url)))
  .then(results => { // (*)
    results.forEach((result, num) => {
      if (result.status == "fulfilled") {
        alert(`#${urls[num]}: ${result.value.status}`);
      }
      if (result.status == "rejected") {
        alert(`#${urls[num]}: ${result.reason}`);
      }
    });
  });

```

I risultati (`results`) nella linea (*) sopra saranno:

```

[
  {status: 'fulfilled', value: ...response...},
  {status: 'fulfilled', value: ...response...},
  {status: 'rejected', reason: ...error object...}
]

```

Così, per ogni promise otteniamo il suo stato e `valore/ragione`.

Polyfill

Se il browser non supporta `Promise.allSettled`, è facile usare un polyfill:

```

if (!Promise.allSettled) {
  const rejectHandler = reason => ({ status: 'rejected', reason });

  const resolveHandler = value => ({ status: 'fulfilled', value });

  Promise.allSettled = function (promises) {
    const convertedPromises = promises.map(p => Promise.resolve(p).then(resolveHandler, rejectHandler));
    return Promise.all(convertedPromises);
  };
}

```

In questo codice, `promises.map` prende i valori in ingresso, li “trasforma” in promises (nel caso in cui sia stata passata una non-promise) con `p => Promise.resolve(p)`, e poi vi aggiunge il gestore (handler) `.then`.

Il gestore handler “trasforma” un risultato positivo `v` in `{state:'fulfilled', value:v}`, ed un errore `r` in `{state:'rejected', reason:r}`. Questo è esattamente il formato di `Promise.allSettled`.

Poi possiamo usare il risultato di `Promise.allSettled` per ottenere i risultati di *tutte* le promise date, anche se alcune venissero respinte.

Promise.race

Similmente a `Promise.all`, accetta un iterabile di promise, ma invece di attendere che siano tutte finite, aspetta per il primo risultato (o errore), e va avanti con quello.

La sintassi è:

```
let promise = Promise.race(iterable);
```

For instance, here the result will be 1 :

```
Promise.race([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

Così, il primo risultato/errore diventa il risultato di tutto `Promise.race`. Quando la prima promise ferma (settled) “vince la gara” (wins the race), tutti i risultati/errori successivi sono ignorati.

Promise.any

Similar to `Promise.race`, but waits only for the first fulfilled promise and gets its result. If all of the given promises are rejected, then the returned promise is rejected with `AggregateError` – a special error object that stores all promise errors in its `errors` property.

The syntax is:

```
let promise = Promise.any(iterable);
```

For instance, here the result will be 1 :

```
Promise.any([
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 1000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

The first promise here was fastest, but it was rejected, so the second promise became the result. After the first fulfilled promise “wins the race”, all further results are ignored.

Here's an example when all promises fail:

```
Promise.any([
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Ouch!")), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Error!")), 2000))
]).catch(error => {
  console.log(error.constructor.name); // AggregateError
  console.log(error.errors[0]); // Error: Ouch!
  console.log(error.errors[1]); // Error: Error
});
```

As you can see, error objects for failed promises are available in the `errors` property of the `AggregateError` object.

Promise.resolve/reject

I metodi `Promise.resolve` e `Promise.reject` vengono utilizzati raramente nel codice moderno, poiché la sintassi `async/await` (che studieremo [più avanti](#)) li rende obsoleti.

Li studiamo per completezza e per quelli che non possono utilizzare `async/await` per qualche ragione.

Promise.resolve

`Promise.resolve(value)` risolve una Promise con il risultato `value`.

Come nell'esempio:

```
let promise = new Promise(resolve => resolve(value));
```

Il metodo viene utilizzato per compatibilità, quando ci si aspetta che una funzione ritorni una promise.

Ad esempio, la funzione `loadCached` sotto, analizza un URL e ne memorizza (sulla cache) il suo contenuto. Per le future chiamate allo stesso URL verrà immediatamente ritornato il contenuto dalla cache, ma utilizzando `Promise.resolve` per renderlo una promise, in questo modo il valore ritornato sarà sempre una promise:

```
let cache = new Map();

function loadCached(url) {
  if (cache.has(url)) {
    return Promise.resolve(cache.get(url)); // (*)
  }

  return fetch(url)
    .then(response => response.text())
    .then(text => {
      cache.set(url, text);
      return text;
    });
}
```

Possiamo scrivere `loadCached(url).then(...)`, perché ci viene garantito che la funzione ritorni una promise. Possiamo sempre utilizzare `.then` dopo `loadCached`. Questo è lo scopo di `Promise.resolve` nella riga `(*)`.

Promise.reject

`Promise.reject(error)` rifiuta una promise con `error`.

Come nell'esempio:

```
let promise = new Promise((resolve, reject) => reject(error));
```

Nella pratica, questo metodo non viene quasi mai utilizzato.

Riepilogo

There are 6 static methods of `Promise` class:

1. `Promise.all(promises)` – aspetta che tutte le promise siano risolte e ritorna array un array dei loro risultati. Se una qualsiasi delle promise date viene respinta, allora diventa l'errore di `Promise.all`, e tutti gli altri risultati sono ignorati.
2. `Promise.allSettled(promises)` (un nuovo metodo) – aspetta che tutte le promises vengano risolte o respinte e ritorna un array dei loro risultati come oggetti con questa forma:
 - `state: 'fulfilled'` or `'rejected'`
 - `value` (se risolta (`fulfilled`) o `reason` (se respinta (`rejected`)).
3. `Promise.race(promises)` – aspetta che la prima promise sia ferma (settle), ed il suo risultato/errore diventa il risultato.
4. `Promise.any(promises)` (metodo aggiunto di recente) – aspetta che la prima promise venga risolta e restituisca il risultato. Se tutte le promises vengono respinte, `AggregateError` ↗ diventa l'errore di `Promise.any`.
5. `Promise.resolve(value)` – crea una promise risolta (resolved) con il valore dato.
6. `Promise.reject(error)` – crea una promise respinta (rejected) con il valore dato.

Di tutti questi, il più comunemente utilizzato è `Promise.all`.

Promisification

Promisification – è una parola lunga per una trasformazione semplice. È la conversione di una funzione che accetta una callback in una funzione che ritorna una promise.

Per essere più precisi, creiamo una funzione wrapper che fa lo stesso, chiamando internamente quella originale, ma ritornando una promise.

Queste trasformazioni sono spesso necessarie nella vita reale, dato che molte funzioni e librerie sono basate su callback. Ma le promise sono più pratiche. Per questo motivo ha senso trasformarle in promise.

Per esempio, abbiamo `loadScript(src, callback)` dal capitolo [Introduzione: callbacks](#).

```
function loadScript(src, callback) {  
  let script = document.createElement('script');  
  script.src = src;  
  
  script.onload = () => callback(null, script);  
  script.onerror = () => callback(new Error(`Errore caricamento script per ${src}`));  
  
  document.head.append(script);  
}  
  
// utilizzo:  
// loadScript('path/script.js', (err, script) => {...})
```

Trasformiamolo in una promise. La nuova funzione `loadScriptPromise(src)` farà lo stesso, ma accetta solo `src` (senza callback) e ritorna una promise.

Here it is:

```
let loadScriptPromise = function(src) {
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {
      if (err) reject(err);
      else resolve(script);
    });
  });
};

// uso:
// loadScriptPromise('path/script.js').then(...)
```

Ora `loadScriptPromise` si adatta bene al nostro codice basato sulle promise.

Come possiamo vedere, delega tutto il lavoro alla `loadScript` originale, passando la sua callback che si traduce nel `resolve/reject` della promise.

Dato che abbiamo bisogno di trasformare in (promisify) molte funzione, ha senso usare un helper.

Questo è molto semplice – `promisify(f)` sotto prende una funzione da trasformare in promise `f` e ritorna una funzione wrapper.

Quel wrapper fa la stessa cosa del codice sopra: ritorna una promise e passa la chiamata alla `f` originale, tracciando il risultato in una sua callback:

```
function promisify(f) {
  return function (...args) { // ritorna una funzione wrapper
    return new Promise((resolve, reject) => {
      function callback(err, result) { // la nostra callback f
        if (err) {
          reject(err);
        } else {
          resolve(result);
        }
      }

      args.push(callback); // aggiunge la nostra callback custom alla fine degli argomenti

      f.call(this, ...args); // chiama la funzione originale
    });
  };
}

// uso:
let loadScriptPromise = promisify(loadScript);
loadScriptPromise(...).then(...);
```

Qui diamo per scontato che la funzione originale aspetti una callback con due argomenti `(err, result)`. Questo è quello che troveremo più spesso. Poi la nostra callback custom è

esattamente nel formato corretto, e `promisify` funziona perfettamente per questo caso.

Ma cosa succederebbe se `f` aspettasse una callback con più argomenti `callback(err, res1, res2)`?

Ecco una modifica di `promisify` che ritorna un array di diversi risultati della callback:

```
// promisify(f, true) per avere un array di risultati
function promisify(f, manyArgs = false) {
  return function (...args) {
    return new Promise((resolve, reject) => {
      function callback(err, ...results) { // la nostra callback custom per f
        if (err) {
          reject(err);
        } else {
          // risolve con tutti i risultati della callback se manyArgs è specificato
          resolve(manyArgs ? results : results[0]);
        }
      }
      args.push(callback);

      f.call(this, ...args);
    });
  };
}

// usage:
f = promisify(f, true);
f(...).then(arrayOfResults => ..., err => ...);
```

In alcuni casi, `err` può essere del tutto assente: `callback(result)`, o c'è qualcosa di particolare nel formato della callback, allora possiamo trasformare in promise (`promisify`) queste funzioni senza usare un helper, manualmente.

Ci sono anche moduli con delle funzioni per trasformare in promise un po' più flessibili, ad esempio [es6-promisify](#). In Node.js è presente una funzione `util.promisify`.

i Da notare:

La trasformazione in promise (promisification) è un ottimo approccio, specialmente quando si utilizza `async/await` (nel prossimo capitolo), ma non è un sostituto totale per le callback.

Ricorda, una promise può avere un solo risultato, ma una callback può tecnicamente essere chiamata più volte.

Così la trasformazione in (promisification) è intesa solo per le funzioni che chiameranno la callback una volta sola. Le chiamate successive saranno ignorate.

Microtasks

I gestori delle Promise `.then/.catch/.finally` sono sempre asincroni.

Anche quando una Promise è immediatamente risolta, il codice sulle linee sotto `.then/.catch/.finally` verrà sempre eseguito prima dei gestori.

Ecco una dimostrazione:

```
let promise = Promise.resolve();

promise.then(() => alert("promise completa"));

alert("codice finito"); // questo alert viene mostrato prima
```

Se lo esegui, vedrai prima `codice finito`, in seguito `promise done`.

Questo è strano, perché la Promise è chiaramente completa dall'inizio.

Perché quindi il `.then` viene eseguito dopo? Cosa succede?

Coda dei Microtask (Microtasks Queue)

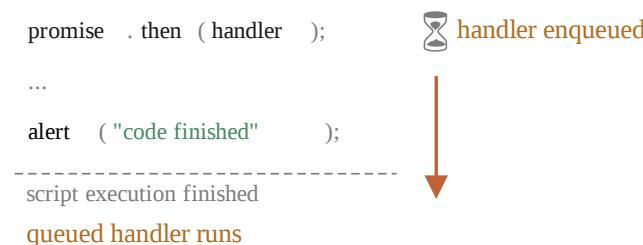
I task asincroni hanno bisogno di una gestione appropriata. Per questo motivo, lo standard specifica una coda interna `PromiseJobs`, più spesso riferita come “coda dei microtask” (microtask queue) (termine di v8).

Come detto nella [specifica ↗](#):

- La coda è primo-dentro-primo-fuori: i task messi in coda per primi sono eseguiti per primi.
- L'esecuzione di un task è iniziata solo quando nient'altro è in esecuzione.

Oppure, per dirla in modo semplice, quando una promise è pronta, i suoi gestori `.then/catch/finally` sono messi nella coda. Non vengono ancora eseguiti. Il motore JavaScript prende un task dalla coda e lo esegue, quando diventa libero dal codice corrente.

Questo è il motivo per cui “codice finito” nell'esempio sopra viene mostrato prima.



I gestori delle promise passano sempre da quella coda interna.

Se c'è una catena con diversi `.then/catch/finally`, allora ognuno di essi viene eseguito in modo asincrono. Cioè, viene prima messo in coda ed eseguito quando il codice corrente è completo e i gestori messi in coda precedentemente sono finiti.

Che cosa succede se per noi l'ordine è importante? Come possiamo far funzionare `code finished` dopo `promise done`?

Facile, basta metterlo in coda con `.then`:

```
Promise.resolve()
  .then(() => alert("promise done!"))
  .then(() => alert("code finished"));
```

Ora l'ordine è come inteso.

Rigetto non gestito (Unhandled rejection)

Ricordi l'evento “unhandledrejection” dal capitolo [Gestione degli errori con le promise?](#)

Ora possiamo vedere esattamente come JavaScript viene a conoscenza che c’è stato un respingimento non gestito (unhandled rejection)

“Unhandled rejection” avviene quando un errore di una promise non è gestito alla fine della coda dei microtask

Normalmente, se ci aspettiamo un errore, aggiungiamo `.catch` alla catena delle promise per gestirlo:

```
let promise = Promise.reject(new Error("Promise Fallita!"));
promise.catch(err => alert('catturato'));

// non viene eseguito: errore gestito
window.addEventListener('unhandledrejection', event => alert(event.reason));
```

...Ma se ci dimentichiamo di aggiungere `.catch`, allora, dopo che la coda dei microtask è vuota, il motore innesca l’evento:

```
let promise = Promise.reject(new Error("Promise Fallita!"));

// Promise Fallita!
window.addEventListener('unhandledrejection', event => alert(event.reason));
```

Cosa succede se gestiamo l’errore dopo? Come qui:

```
let promise = Promise.reject(new Error("Promise Fallita!"));
setTimeout(() => promise.catch(err => alert('caught')), 1000);

// Error: Promise Fallita!
window.addEventListener('unhandledrejection', event => alert(event.reason));
```

Ora il respingimento non gestito appare di nuovo. Perché? `unhandledrejection` viene innescato quando la coda dei microtask è completa. Il motore esamina le promise e, se qualcuna di esse è in stato “rejected”, allora l’evento è generato.

Nell’esempio, il `.catch` aggiunto da `setTimeout` viene eseguito, ovviamente lo fa, ma dopo, quando `unhandledrejection` è già avvenuto.

Se non fossimo a conoscenza della coda dei microtask, potremmo chiederci: “Perché il gestore di `unhandledrejection` viene eseguito? Abbiamo catturato l’errore!”.

Ma ora sappiamo che `unhandledrejection` è generato quando la coda dei microtask è completa: il motore esamina le promise e, se una di esse è in stato “rejected”, allora l’evento viene innescato.

Nell'esempio sopra, anche il `.catch` aggiunto da `setTimeout` viene innescato, ma dopo, quando `UnhandledRejection` è già avvenuto, quindi questo non cambia niente.

Riepilogo

La gestione delle promise è sempre asincrona, dato che tutte le azioni delle promise passano attraverso la coda “promise jobs”, anche chiamata “microtask queue” (termine di v8).

Così, i gestori `.then/catch/finally` sono sempre chiamati dopo che il codice corrente è finito.

Se abbiamo bisogno della certezza che un pezzo di codice sia eseguito dopo `.then/catch/finally`, possiamo aggiungerlo ad una chiamata `.then` in catena.

Nella maggior parte dei motori JavaScript, inclusi i browser e Node.js, il concetto di microtask è strettamente legato al “loop degli event” (event loop) ed ai “macrotasks”. Dato che questi non hanno una relazione diretta con le promise, sono coperti in un'altra parte del tutorial, nel capitolo [Event loop: microtasks e macrotasks](#).

Async/await

Esiste una sintassi speciale per lavorare con le promise in un modo più comodo, chiamata `async/await`. È sorprendentemente facile da capire e usare.

Async functions

Iniziamo con la parola chiave `async`. Può essere messa prima di una funzione, come nell'esempio sotto:

```
async function f() {  
  return 1;  
}
```

La parola “`async`” prima di una funzione significa una cosa semplice: la funzione ritorna sempre una promise. Gli altri valori sono “avvolti” wrapped in una promise risolta automaticamente.

Per esempio, questa funzione ritorna una promise risolta con il risultato di `1`, proviamola:

```
async function f() {  
  return 1;  
}  
  
f().then(alert); // 1
```

...Possiamo anche ritornare esplicitamente una promise, sarebbe lo stesso:

```
async function f() {  
  return Promise.resolve(1);  
}
```

```
f().then(alert); // 1
```

Così, `async` assicura che la funzione ritorni una promise, e “avvolge” (wraps) le non-promise al suo interno. Abbastanza semplice, vero? Ma non sono quello. C’è un’altra parola chiave, `await`, che funziona solo nelle funzioni `async`, ed è piuttosto cool.

Await

La sintassi:

```
// funziona solo all'interno delle funzioni async
let value = await promise;
```

La parola chiave `await` fa attendere JavaScript fino a quando la promise è ferma (settles) e ritorna il suo risultato.

Ecco un esempio con una promise che si risolve in un secondo:

```
async function f() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("fatto!"), 1000)
  });

  let result = await promise; // attende fino a quando la promise si risolve (*)

  alert(result); // "fatto!"
}

f();
```

L’esecuzione della funzione “va in pausa” alla linea `(*)` e riprende quando la promise si ferma (settles), con `result` che diventa il suo risultato. Così il codice sopra mostra “fatto!” in un secondo.

Enfatizziamo: `await` fa letteralmente attendere JavaScript fino a quando la promise si ferma, poi va avanti con il risultato. Questo non costa alcuna risorsa della CPU, perché il motore può fare altri lavori nel frattempo: eseguire altri script, gestire eventi etc.

È solo una sintassi più elegante per ottenere il risultato della promise `promise.then`, più facile da leggere e da scrivere.

Non possiamo usare `await` nelle normali funzioni

Se proviamo ad utilizzare `await` in una funzione non asincrona, otterremmo un errore di sintassi:

```
function f() {
  let promise = Promise.resolve(1);
  let result = await promise; // Syntax error
}
```

Avremo questo errore se non mettiamo `async` prima di una funzione. Come detto, `await` funziona solo all'interno di una funzione asincrona (`async function`).

Prendiamo l'esempio `showAvatar()` dal capitolo [Concatenamento di promise \(promise chaining\)](#) e riscriviamolo usando `async/await`:

1. Avremo bisogno di sostituire le chiamate `.then` con `await`.
2. Inoltre dovremo rendere la funzione asincrona (`async`) per farli lavorare.

```
async function showAvatar() {

  // legge il nostro JSON
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();

  // legge l'utente GitHub
  let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);
  let githubUser = await githubResponse.json();

  // show the avatar
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  // wait 3 seconds
  await new Promise((resolve, reject) => setTimeout(resolve, 3000));

  img.remove();

  return githubUser;
}

showAvatar();
```

Piuttosto pulito e facile da leggere, vero? Molto meglio che prima.

i `await` Non funzionerà nel codice di livello più alto

Le persone che hanno appena iniziato ad utilizzare `await` tendono a dimenticare il fatto che non possiamo utilizzare `await` nel codice di livello più alto. Per esempio, questo non funzionerà:

For instance:

```
// we assume this code runs at top level, inside a module
let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();

console.log(user);
```

Possiamo “avvolgerlo” wrap in una funzione async anonima, come qui:

```
(async () => {
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();
  ...
})();
```

i `await` accetta i “thenables”

Come `promise.then`, `await` permette di usare gli oggetti “thenable” (quelli con un metodo `then`). L’idea è che un oggetto di terze parti possa non essere una promise, ma essere promise-compatibile: se supporta `.then`, è abbastanza per usarlo con `await`.

Ecco una dimostrazione: la classe `Thenable` class, l’`await` sotto accetta:

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve);
    // risolve con this.num*2 dopo 1000ms
    setTimeout(() => resolve(this.num * 2), 1000); // (*)
  }
}

async function f() {
  // attende per 1 secondo, poi il risultato diventa 2
  let result = await new Thenable(1);
  alert(result);
}

f();
```

Se `await` riceve un oggetto non-promise con `.then`, chiama quel metodo passando le funzioni native `resolve`, `reject` come argomenti. Poi `await` fino a quando una delle due viene chiamata (nell’esempio sopra avviene nella linea `(*)`) e poi procede con il risultato.

i Methods `async` delle classi

Per dichiarare un metodo `async` di una classe, basta precederlo con `async`:

```
class Waiter {
  async wait() {
    return await Promise.resolve(1);
  }
}

new Waiter()
  .wait()
  .then(alert); // 1 (this is the same as (result => alert(result)))
```

Il significato è lo stesso: assicura che il valore ritornato sia una promise ed abilita `await`.

Gestione degli errori

Se una promise si risolve normalmente, allora `await promise` ritorna il risultato. Ma nel caso di un rigetto (rejection), solleva l’errore, proprio come se in quella linea ci fosse stato un `throw`.

Questo codice:

```
async function f() {
  await Promise.reject(new Error("Whoops!"));
}
```

...È lo stesso di questo:

```
async function f() {
  throw new Error("Whoops!");
}
```

Nelle situazioni reali, la promise può prendere del tempo prima di venire rigettata. In quel caso ci sarà un ritardo prima che `await` sollevi un errore.

Possiamo catturare l'errore utilizzando `try..catch`, allo stesso modo che con un regolare `throw`:

```
async function f() {

  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}

f();
```

In caso di errore, il controllo salta al blocco `catch`. Possiamo anche “avvolgere” (wrap) più linee:

```
async function f() {

  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // cattura gli errori sia in fetch che in response.json
    alert(err);
  }
}

f();
```

Se non abbiamo `try..catch`, allora la promise generata dalla chiamata della funzione `async f()` viene rigettata. Possiamo aggiungere `.catch` per gestirla:

```
async function f() {
  let response = await fetch('http://no-such-url');
```

```
}
```

// f() diventa una promise rigettata
f().catch(alert); // TypeError: failed to fetch // (*)

Se ci dimentichiamo di aggiungere `.catch` qui, allora otterremo un errore di una promise non gestito (visibile nella console). Possiamo catturare usando un gestore di eventi globale come descritto nel capitolo [Gestione degli errori con le promise](#).

i `async/await and promise.then/catch`

Quando usiamo `async/await`, raramente abbiamo bisogno di `.then`, perché `await` gestisce l'attesa per noi. E possiamo usare un normale `try..catch` invece di `.catch`. Questo è spesso (non sempre) più conveniente.

Ma al livello più alto del codice, quando siamo fuori da qualunque funzione `async`, non siamo sintatticamente in grado di usare `await`, così è una pratica normale usare `.then/catch` per gestire il risultato finale degli errori che “cadono attraverso” (falling-through).

Come nella linea `(*)` dell'esempio sopra.

i `async/await funziona bene con Promise.all`

Quando dobbiamo attendere per più promise, possiamo avvolgerle (wrap) in `Promise.all` e poi attendere (`await`):

```
// attende per l'array dei risultati
let results = await Promise.all([
  fetch(url1),
  fetch(url2),
  ...
]);
```

In caso di errore, si propaga come di solito: dalla promise fallita a `Promise.all`, poi diventa una eccezione che possiamo catturare usando `try..catch` attorno alla chiamata.

Riepilogo

La parola chiave `async` prima di una funzione ha due effetti:

1. Fa sempre ritornare una promise.
2. Permette di usare `await` al suo interno.

La parola chiave `await` prima di una promise fa attendere JavaScript fino a quando la promise è ferma, e poi:

1. Se c'è un errore, l'eccezione è generata, come se `throw error` fosse stato chiamato in quel punto.
2. Altrimenti, ritorna il risultato.

Insieme forniscono una ottima struttura per scrivere codice asincrono che sia facile sia da leggere che da scrivere.

Con `async/await` raramente avremo bisogno di scrivere `promise.then/catch`, ma comunque non dovremo dimenticare che sono sempre basate su promise, perché a volte (e.s. nello scope più esterno) dovremo usare questi metodi. Inoltre `Promise.all` è una buona cosa per attendere per più task contemporaneamente.

✓ Esercizi

Rewrite using `async/await`

Rewrite this example code from the chapter [Concatenamento di promise \(promise chaining\)](#) using `async/await` instead of `.then/catch`:

```
function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new Error(response.status);
      }
    });
}

loadJson('no-such-user.json')
  .catch(alert); // Error: 404
```

[Alla soluzione](#)

Riscrivere "rethrow" con `async/await`

Sotto puoi trovare l'esempio "rethrow". Riscriverlo usando `async/await` invece di `.then/catch`.

Sbarazzarsi della ricorsione a favore di un ciclo in `demoGithubUser`: con `async/await` diventa facile da fare.

```
class HttpError extends Error {
  constructor(response) {
    super(`#${response.status} for ${response.url}`);
    this.name = 'HttpError';
    this.response = response;
  }
}

function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
```

```

        throw new HttpError(response);
    }
});
}

// Ask for a user name until github returns a valid user
function demoGithubUser() {
    let name = prompt("Enter a name?", "iliakan");

    return loadJson(`https://api.github.com/users/${name}`)
        .then(user => {
            alert(`Full name: ${user.name}`);
            return user;
        })
        .catch(err => {
            if (err instanceof HttpError && err.response.status == 404) {
                alert("No such user, please reenter.");
                return demoGithubUser();
            } else {
                throw err;
            }
        });
}

demoGithubUser();

```

[Alla soluzione](#)

Call `async` from non-`async`

Abbiamo una funzione “regolare”. Come chiamare `async` da questa ed usare il suo risultato?

```

async function wait() {
    await new Promise(resolve => setTimeout(resolve, 1000));

    return 10;
}

function f() {
    // ...cosa bisogna scrivere qui?
    // dobbiamo chiamare async wait() ed aspettare per ricevere 10
    // ricorda, non possiamo usare "await"
}

```

P.S. Il task è tecnicamente molto semplice, ma la domanda è piuttosto comune per gli sviluppatori nuovi ad `async/await`.

[Alla soluzione](#)

Generators, iterazioni avanzate I generatori

Le funzioni ritornano normalmente un solo valore (a volte non ritornano nulla).

I generatori possono ritornare (“yield”) valori multipli, uno dopo l’altro, ogni volta che vengono invocati. Sono, di fatto, lo strumento ideale da utilizzare con gli [iteratori](#), dal momento che ci consentono di creare flussi di dati con facilità.

Le funzioni generatrici

Per creare un generatore, abbiamo bisogno di uno specifico costrutto sintattico: `function*`, chiamato, appunto, “funzione generatrice”.

Ecco un esempio:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

Le funzioni generatrici si comportano diversamente rispetto alle normali funzioni. Quando una generatrice viene invocata, di fatto, il codice al suo interno non viene eseguito, ma ritorna uno speciale oggetto, chiamato “oggetto generatore”, che ne consente di gestire l’esecuzione.

Dai un’occhiata qua:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

// "la funzione generatrice" crea un "oggetto generatore"
let generator = generateSequence();
alert(generator); // [object Generator]
```

L’esecuzione del codice della funzione non è ancora iniziata:

```
function* generateSequence() { ←
  yield 1;
  yield 2;
  return 3;
}
```

Il metodo principale di un oggetto generatore è `next()`. Quando invocato, esegue le istruzioni in esso contenute fino alla prossima istruzione `yield <valore>` (`valore` può essere omesso, in tal caso sarà `undefined`). A questo punto l’esecuzione si arresta e il `valore` viene “ceduto” al codice esterno.

Il risultato dell’esecuzione di `next()` è sempre un oggetto con due proprietà:

- `value` : il valore che viene “ceduto”.
- `done` : `true`, se il codice della funzione è stato eseguito completamente, altrimenti, `false`.

Ad esempio, qui andiamo a creare un generatore e otteniamo il primo valore “ceduto”:

```
function* generateSequence() {  
  yield 1;  
  yield 2;  
  return 3;  
}  
  
let generator = generateSequence();  
  
let one = generator.next();  
  
alert(JSON.stringify(one)); // {value: 1, done: false}
```

A questo punto, abbiamo ottenuto solo il primo valore e l'esecuzione della funzione ha raggiunto la seconda riga:

```
function* generateSequence() {  
  yield 1; ← {value: 1, done: false}  
  yield 2;  
  return 3;  
}
```

Invochiamo ancora `generator.next()`. L'esecuzione del codice riprenderà da dove si era fermata, fino a restituire il valore del prossimo `yield`:

```
let two = generator.next();  
  
alert(JSON.stringify(two)); // {value: 2, done: false}
```

```
function* generateSequence() {  
  yield 1; ← {value: 2, done: false}  
  yield 2;  
  return 3;  
}
```

Per finire, invocandolo nuovamente (`generator.next()`), si raggiungerà l'istruzione `return` che terminerà la funzione:

```
let three = generator.next();  
  
alert(JSON.stringify(three)); // {value: 3, done: true}
```

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
} → {value: 3, done: true}
```

A questo punto il generatore ha terminato. Possiamo vederlo dal risultato finale `done: true` e `value: 3`.

Effettuare nuove chiamate a `generator.next()` non avrebbe più senso. Se lo facciamo, otterremmo sempre lo stesso oggetto: `{done: true}`.

i `function* f(...)` or `function *f(...)` ?

Entrambe le sintassi sono corrette.

La prima, tuttavia, è la più utilizzata, dal momento che è l'asterisco `*` a indicare che la funzione è una generatrice, non il nome. Per questo motivo ha più senso accoppiare l'asterisco con la parola chiave `function`.

I generatori sono iteratori

Come probabilmente avrai intuito dalla presenza del metodo `next()`, i generatori sono **iterabili**.

Possiamo eseguire cicli sui valori ritornati utilizzando `for .. of`:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, then 2
}
```

Non trovi che sia molto più leggibile di `.next().value`?

...Nota bene: l'esempio precedente produrrà come risultato `1` e `2`, nulla più. Il `3` non verrà preso in considerazione!

La spiegazione di questo comportamento sta nel fatto che `for .. of` ignora l'ultimo `value` non appena la proprietà è `done: true`. Per questo motivo, se vogliamo che tutti i valori siano mostrati da `for .. of`, dobbiamo ritornarli tramite `yield`:

```
function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
```

```

}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, poi 2, poi 3
}

```

Dal momento che i generatori sono iteratori, possiamo usufruire di tutte le funzionalità che ne derivano, per esempio lo “spread operator”.

```

function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
}

let sequence = [0, ...generateSequence()];

alert(sequence); // 0, 1, 2, 3

```

Nell'esempio precedente `...generateSequence()` converte l'oggetto generatore iteratore in un array di elementi (puoi approfondire l'argomento relativo allo spread operator nel capitolo Articolo "rest-parameters-spread-operator" non trovato)

Usare i generatori con gli iteratori

Tempo fa, nel capitolo [Iteratori](#) abbiamo creato un oggetto iteratore `range` che ritorna valori `from..to` (da...a).

Ecco l'esempio, per rinfrescarci la memoria:

```

let range = {
  from: 1,
  to: 5,

  // for..of invoca questo metodo solo all'inizio
  [Symbol.iterator]() {
    // ...ritorna l'oggetto iteratore:
    // da qui in poi, for..of utilizzera; solo quell'oggetto per ottenere i valori success
    return {
      current: this.from,
      last: this.to,

      // next() viene invocata ad ogni iterazione fino alla fine del ciclo for..of
      next() {
        // dovrebbe ritornare il valore sotto forma di un oggett {done:..., value:...}
        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      },
    };
  },
};

```

```
};

// l'iterazione su range ritorna i numeri compresi tra range.from e range.to
alert([...range]); // 1,2,3,4,5
```

Possiamo usare una funzione generatrice come iteratore assegnandola a `Symbol.iterator`.

Ecco qui lo stesso `range`, ma in una forma più compatta:

```
let range = {
  from: 1,
  to: 5,

  *[Symbol.iterator]() {
    // forma abbreviata di [Symbol.iterator]: function()
    for (let value = this.from; value <= this.to; value++) {
      yield value;
    }
  },
};

alert([...range]); // 1,2,3,4,5
```

Il funzionamento è invariato dal momento che `range[Symbol.iterator]()` ora ritorna un generatore che è esattamente quello che `for...of` si aspetta:

- ha un metodo `.next()`
- il quale ritorna valori nella forma `{value: ..., done: true/false}`

Questa non è una coincidenza, ovviamente. I generatori sono stati aggiunti al linguaggio JavaScript tenendo a mente gli iteratori, per implementarli più facilmente.

La variante con i generatori è molto più concisa del codice originale di `range` ma mantiene le funzionalità invariate.

I generatori potrebbero generare valori per sempre

Negli esempi precedenti abbiamo generato sequenze finite ma possiamo anche creare generatori che restituiscono valori infinitamente. Per esempio, una sequenza infinita di numeri pseudo-casuali.

Ciò richiederebbe sicuramente un `break` (o un `return`) nel `for...of` che utilizziamo per iterare su tale generatore, altrimenti il ciclo si ripeterebbe all'infinito bloccando l'esecuzione dell'applicazione.

Composizione di generatori

La composizione dei generatori è una caratteristica particolare dei generatori che consente di “innestarli” l'uno nell'altro, in modo trasparente.

Per esempio, data una funzione che genera una sequenza di numeri:

```
function* generateSequence(start, end) {
```

```
for (let i = start; i <= end; i++) yield i;
}
```

Vogliamo usarla per generare una sequenza più complessa:

- per prime le cifre `0..9` (con i codici carattere 48...57),
- seguite dalle lettere dell'alfabeto `a..z` (codici carattere 65...90)
- seguite dalle lettere maiuscole `A..Z` (codici carattere 97...122)

Possiamo usare questa sequenza, ad esempio, per creare password selezionandone i caratteri (potremmo anche aggiungere caratteri sintattici), ma generiamo la sequenza per ora.

In una normale funzione, per combinare i risultati di più funzioni, dapprima invochiamo le funzioni, ne memorizziamo i risultati e, infine, li combiniamo.

Usando i generatori, c'è una sintassi speciale di `yield*` per "innestare" (comporre) un generatore all'interno di un altro.

Il generatore composto:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}

function* generatePasswordCodes() {

  // 0..9
  yield* generateSequence(48, 57);

  // A..Z
  yield* generateSequence(65, 90);

  // a..z
  yield* generateSequence(97, 122);

}

let str = '';

for(let code of generatePasswordCodes()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9A..Za..z
```

La direttiva `yield*` delega l'esecuzione a un altro generatore. Con il termine *delega* si intende che `yield* gen` itera sui valori del generatore `gen` e, in modo trasparente, inoltra i valori che restituisce verso l'esterno. Come se i valori fossero restituiti dal generatore più esterno.

Il risultato è lo stesso che otterremmo mettendo in sequenza il codice dei generatori annidati:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}
```

```

function* generateAlphaNum() {

  // yield* generateSequence(48, 57);
  for (let i = 48; i <= 57; i++) yield i;

  // yield* generateSequence(65, 90);
  for (let i = 65; i <= 90; i++) yield i;

  // yield* generateSequence(97, 122);
  for (let i = 97; i <= 122; i++) yield i;

}

let str = '';

for(let code of generateAlphaNum()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9A..Za..z

```

La composizione dei generatori è un modo naturale di immettere il flusso di un generatore all'interno di un altro. Non viene utilizzata memoria aggiuntiva per memorizzare i valori intermedi.

“yield” è una via a doppio senso

Finora i generatori sono assimilati agli iteratori, con una sintassi speciale per generare valori ma, di fatto, sono molto più potenti e flessibili.

Questo perché `yield` è una via a doppio senso: non solo ritorna il risultato verso l'esterno ma provvede anche a passare il valore all'interno del generatore.

Per fare questo, dovremmo invocare `generator.next(arg)` con un argomento. Tale argomento diventerà il risultato di `yield`.

Vediamo un esempio:

```

function* gen() {
  // Passa una domanda al codice esterno e attende una risposta
  let result = yield "2 + 2 = ?"; // (*)

  alert(result);

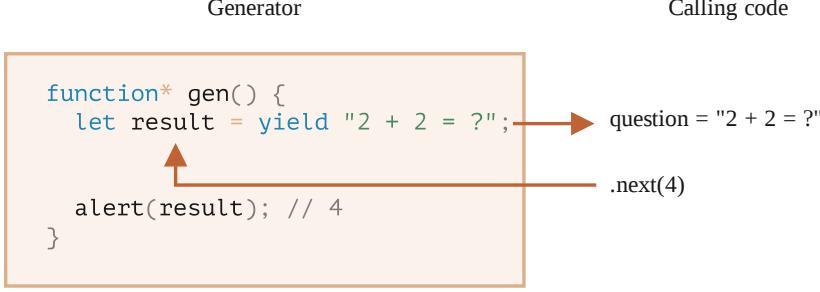
}

let generator = gen();

let question = generator.next().value; // <-- yield ritorna il valore

generator.next(4); // --> pass il risultato al generatore

```



1. La prima invocazione di `generator.next()` è sempre senza argomento. Inizia l'esecuzione e ritorna il risultato della prima `yield "2+2=?"`. A questo punto il generatore si arresta (si ferma su quella riga).
2. Di seguito, come mostrato nella figura sopra, il risultato di `yield` va dentro alla variabile `question` nel codice all'esterno.
3. Quando eseguiamo `generator.next(4)`, il generatore riprende l'esecuzione e `4` finisce nel risultato: `let result = 4`.

Si noti che il codice all'esterno non deve per forza invocare `next(4)` immediatamente. Potrebbe impiegare del tempo, ma questo non è un problema: il generatore attenderà.

Per esempio:

```
// ripristina il generatore dopo un certo lasso di tempo
setTimeout(() => generator.next(4), 1000);
```

Come possiamo vedere, a differenza delle normali funzioni, un generatore e il codice che lo invoca possono scambiarsi risultati passando valori a `next/yield`.

Per rendere il tutto ancora più evidente, ecco un altro esempio, con più chiamate:

```

function* gen() {
  let ask1 = yield "2 + 2 = ?";

  alert(ask1); // 4

  let ask2 = yield "3 * 3 = ?";

  alert(ask2); // 9
}

let generator = gen();

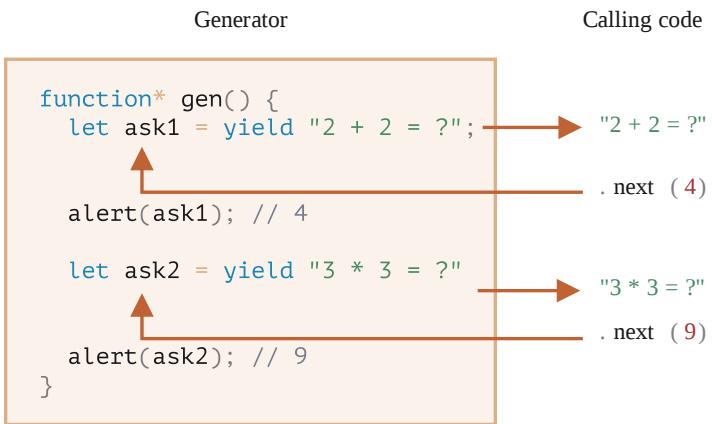
alert(generator.next().value); // "2 + 2 = ?"

alert(generator.next(4).value); // "3 * 3 = ?"

alert(generator.next(9).done); // true

```

Illustrazione dell'esecuzione:



1. La prima `.next()` inizia l'esecuzione... raggiunge la prima `yield`.
2. Il risultato viene ritornato al codice esterno.
3. La seconda `.next(4)` passa `4` al generatore come risultato della prima `yield` e riprende l'esecuzione.
4. ...la seconda `yield` viene raggiunta e diventa il risultato della chiamata al generatore.
5. La terza `next(9)` passa `9` nel generatore come risultato della seconda `yield` e riprende l'esecuzione che raggiunge la fine della funzione, dunque, `done: true`.

È un po' come una partita a “ping-pong”. Ogni `next(value)` (escluso il primo), passa un valore al generatore. Questo valore diventa il risultato della `yield` corrente per poi ritornare il risultato della `yield` successiva.

generator.throw

Come abbiamo visto negli esempi precedenti, il codice esterno può passare un valore all'interno del generatore, come risultato della `yield`.

...ma può anche passargli (`throw`) un errore. Ciò è naturale, dal momento che un errore è comunque un risultato.

Per passare un errore all'interno di una `yield`, dovremmo invocare `generator.throw(err)`. In questo caso, tale `err` risulta generato dalla riga contenente tale `yield`.

Nel prossimo esempio, la `yield` di `"2 + 2 = ?"` genera un errore:

```

function* gen() {
  try {
    let result = yield "2 + 2 = ?"; // (1)

    alert("The execution does not reach here, because the exception is thrown above");
  } catch(e) {
    alert(e); // shows the error
  }
}

let generator = gen();

let question = generator.next().value;

```

```
generator.throw(new Error("The answer is not found in my database")); // (2)
```

L'errore, lanciato all'interno del generatore alla riga (2) genera un'eccezione alla riga (1) in corrispondenza della `yield`. Nell'esempio precedente `try..catch` gestisce l'errore e lo mostra.

Se non gestiamo l'errore, come accadrebbe con qualsiasi eccezione, quest'ultima andrebbe a causare un errore nel codice esterno.

La riga corrente del codice esterno è quella che contiene `generator.throw`, identificata da (2). Possiamo, pertanto, gestire l'eccezione qui, come nell'esempio:

```
function* generate() {
  let result = yield "2 + 2 = ?"; // Errore in questa riga
}

let generator = generate();

let question = generator.next().value;

try {
  generator.throw(new Error("The answer is not found in my database"));
} catch(e) {
  alert(e); // mostra l'errore
}
```

Se non gestiamo l'errore qui, come al solito, questo risalirà fino al codice più esterno, se esistente, altrimenti farà fallire lo script.

Riepilogo

- I generatori vengono creati tramite funzioni generatrici `function* f(...){...}`.
- Solo nei generatori può esistere un operatore `yield`.
- Il codice esterno e il generatore possono intercambiare risultato tramite chiamate a `next/yield`.

Nel JavaScript moderno, i generatori vengono usati raramente ma a volte possono essere utili, dal momento che la loro capacità di intercambiare dati con il codice all'esterno è alquanto unica. Sicuramente, sono un ottimo modo per creare degli iteratori.

Nel prossimo capitolo impareremo a usare i generatori asincroni, utilizzati per leggere flussi di dati generati in modo asincrono (per esempio, dati paginati ottenuti dalla rete) nei cicli `for await ... of`.

Questo è un caso d'uso molto importante, dal momento che nella programmazione web ci troviamo spesso a manipolare flussi di dati.

✓ Esercizi

Pseudo-random generator

There are many areas where we need random data.

One of them is testing. We may need random data: text, numbers, etc. to test things out well.

In JavaScript, we could use `Math.random()`. But if something goes wrong, we'd like to be able to repeat the test, using exactly the same data.

For that, so called “seeded pseudo-random generators” are used. They take a “seed”, the first value, and then generate the next ones using a formula so that the same seed yields the same sequence, and hence the whole flow is easily reproducible. We only need to remember the seed to repeat it.

An example of such formula, that generates somewhat uniformly distributed values:

```
next = previous * 16807 % 2147483647
```

If we use `1` as the seed, the values will be:

1. 16807
2. 282475249
3. 1622650073
4. ...and so on...

The task is to create a generator function `pseudoRandom(seed)` that takes `seed` and creates the generator with this formula.

Usage example:

```
let generator = pseudoRandom(1);

alert(generator.next().value); // 16807
alert(generator.next().value); // 282475249
alert(generator.next().value); // 1622650073
```

[Apri una sandbox con i test.](#) ↗

[Alla soluzione](#)

Iteratori e generatori asincroni

Gli iteratori asincroni consentono di iterare su dati che arrivano in modo asincrono, a richiesta. Per esempio, quando eseguiamo una serie di download parziali dalla rete. I generatori asincroni ci consentono di semplificare questo processo.

Vediamo prima un semplice esempio per prendere confidenza con la sintassi, dopodiché analizzeremo un caso d'uso reale.

Iteratori asincroni

Gli iteratori asincroni sono simili ai comuni iteratori, con alcune differenze sintattiche.

Gli oggetti iteratori “comuni”, come abbiamo detto nel capitolo [iteratori](#), si presentano in questo modo:

```
let range = {
  from: 1,
  to: 5,

  // for..of invoca questo metodo una sola volta all'inizio dell'esecuzione

  [Symbol.iterator]() { // called once, in the beginning of for..of

    // ...ritorna l'oggetto iteratore:
    // dopodich&eacute;, for..of interagisce solo con questo oggetto,
    // chiedendogli i valori successivi tramite il metodo next()

    return {
      current: this.from,
      last: this.to,

      // next() viene invocato ad ogni iterazione dal ciclo for..of
      next() { // (2)
        // dovrebbe ritornare il valore come un oggetto {done:..., value:...}
        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      }
    };
  }
};

for(let value of range) {
  alert(value); // 1 poi 2, poi 3, poi 4, poi 5
}
```

Se necessario, rileggersi il [capitolo sugli iteratori](#) per avere maggiori dettagli circa gli iteratori comuni.

Per rendere l'oggetto iteratore asincrono:

1. Dobbiamo usare `Symbol.asyncIterator` anziché `Symbol.iterator`.
2. `next()` dovrebbe ritornare una promise.
3. Per iterare sui valori di tale oggetto, dobbiamo usare un ciclo del tipo: `for await (let item of iterable)`.

Rendiamo l'oggetto `range` iterabile, come nell'esempio precedente ma, questa volta, ritornerà i valori in modo asincrono, uno ogni secondo:

```
let range = {
  from: 1,
  to: 5,

  // for await..of invoca questo metodo una sola volta all'inizio dell'esecuzione
```

```

[Symbol.asyncIterator]() { // (1)
    // ...ritorna l'oggetto iteratore:
    // dopodich&egrave;, for await..of interagisce solo con questo oggetto,
    // chiedendogli i valori successivi tramite il metodo next()

    return {
        current: this.from,
        last: this.to,

        // next() viene invocato ad ogni iterazione dal ciclo for await..of
        async next() { // (2)
            // dovrebbe ritornare il valore come un oggetto {done:..., value:...}
            // (automaticamente racchiuso in una promise dal momento che siamo un metodo 'async')

            // possiamo utilizzare await all'interno per eseguire codice asincrono:

            await new Promise(resolve => setTimeout(resolve, 1000)); // (3)

        }
    };
}

(async () => {

    for await (let value of range) { // (4)
        alert(value); // 1,2,3,4,5
    }

})()

```

Possiamo notare che la struttura è simile a quella dei comuni iteratori:

1. Per rendere un oggetto iterabile in modo asincrono, esso deve contenere un metodo `Symbol.asyncIterator` (1).
2. Questo metodo deve ritornare un oggetto contenente il metodo `next()`, che ritorna a sua volta una promise (2).
3. Il metodo `next()` non deve necessariamente essere `async`; può essere un metodo normale che ritorna una promise, anche se `async` ci consentirebbe di utilizzare `await`, che può tornarci utile. Nell'esempio, abbiamo utilizzato un ritardo di un secondo (3).
4. Per iterare dobbiamo utilizzare il ciclo `for await(let value of range)` (4), si tratta di aggiungere “`await`” dopo il “`for`”. Questo ciclo invoca il metodo `range[Symbol.asyncIterator]()` una sola volta, dopodiché il metodo invocherà `next()` per ottenere i valori.

Ecco una semplice tabella di riepilogo:

	Iteratori	Iteratori asincroni
Metodo dell'oggetto che restituisce l'iteratore	<code>Symbol.iterator</code>	<code>Symbol.asyncIterator</code>

	Iteratori	Iteratori asincroni
Il valore ritornato da <code>next()</code> è ciclo da utilizzare	<code>qualsiasi valore</code> <code>for..of</code>	<code>Promise</code> <code>for await..of</code>

Lo spread operator '...' non funziona in modo asincrono

Le funzionalità offerte dai comuni iteratori (sincroni) non sono disponibili per gli iteratori asincroni.

Per esempio, lo spread operator non può essere utilizzato:

```
alert([...range]); // Errore, non c'è Symbol.iterator
```

Questo è prevedibile, dal momento che lo spread operator ha bisogno di `Symbol.iterator` anziché `Symbol.asyncIterator`. Lo stesso vale per `for..of` (senza `await`).

Generatori asincroni

Come già sappiamo, JavaScript supporta anche i cosiddetti generatori, che sono anche iteratori.

Ricordiamo l'esempio del generatore di una sequenza di numeri da `start` a `end`, nel capitolo [generatori](#):

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) {
    yield i;
  }

  for (let value of generateSequence(1, 5)) {
    alert(value); // 1, poi 2, poi 3, poi 4, poi 5
  }
}
```

Nei normali generatori non possiamo usare `await`. Tutti i valori devono essere ritornati in modo sincrono: non c'è modo di ritornare valori "futuri" utilizzando il ciclo `for..of`, dal momento che si tratta di un costrutto di tipo sincrono.

Cosa fare se avessimo bisogno di usare `await` all'interno di un generatore? Per eseguire, ad esempio, una richiesta dalla rete?

Non c'è problema, sarà sufficiente anteporre la parola chiave `async`, come nell'esempio seguente:

```
async function* generateSequence(start, end) {

  for (let i = start; i <= end; i++) {

    // Wow, can use await!
    await new Promise(resolve => setTimeout(resolve, 1000));
  }
}
```

```

    yield i;
}

}

(async () => {

  let generator = generateSequence(1, 5);
  for await (let value of generator) {
    alert(value); // 1, poi 2, poi 3, poi 4, poi 5
  }
})();

```

In questo modo abbiamo ottenuto il generatore asincrono, che possiamo usare nelle iterazioni con il ciclo `for await...of`.

E' molto semplice. Aggiungiamo la parola chiave `async` ed ecco che il generatore può utilizzare `await` al suo interno e trarre vantaggio delle promise, così come di tutte le altre funzioni asincrone.

Tecnicamente, un'altra importante caratteristica dei generatori asincroni è che anche il relativo metodo `generator.next()` diventa asincrono, ritornando delle promise.

Con un generatore normale utilizzeremmo `result = generator.next()` per ottenere i valori ritornati. Con i generatori asincroni dobbiamo, invece, aggiungere `await`, come nell'esempio:

```
result = await generator.next(); // result = {value: ..., done: true/false}
```

That's why `async` generators work with `for await...of`.

```
## Iteratori asincroni
```

Come già sappiamo, per rendere un semplice oggetto un oggetto iteratore, dobbiamo aggiungere:

```
```js
let range = {
 from: 1,
 to: 5,
 [Symbol.iterator]() {
 return <oggetto che abbia un metodo next() per trasformare l'oggetto in un iteratore>
 }
}
```

```

Un approccio comune è quello di far ritornare a `Symbol.iterator` un generatore anziché una funzione.

Ricordiamo di seguito un esempio dal capitolo [](info:generators):

```
```js run
let range = {
 from: 1,
 to: 5,
```

```

*[Symbol.iterator]() {
 // sintassi compatta di [Symbol.iterator]: function*()
 for (let value = this.from; value <= this.to; value++) {
 yield value;
 }
},
};

for (let value of range) {
 alert(value); // 1, poi 2, poi 3, poi 4, poi 5
}
```

```

In questo esempio l'oggetto `range` è un iteratore e il generatore `*[Symbol.iterator]` lo è.

Se volessimo aggiungere delle funzionalità asincrone al generatore, dovremmo sostituire `yield` con `return`:

```

```js run
let range = {
 from: 1,
 to: 5,

 // this line is same as [Symbol.asyncIterator]: async function*() {
 async *[Symbol.asyncIterator]() { // come per [Symbol.iterator]: async function*()
 for(let value = this.from; value <= this.to; value++) {

 // mettiamo una pausa tra i valori ritornati, aspettando un secondo
 await new Promise(resolve => setTimeout(resolve, 1000));

 yield value;
 }
 }
};

(async () => {

```

```

 for await (let value of range) {
 alert(value); // 1, poi 2, poi 3, poi 4, poi 5
 }
})();
```

```

Adesso i valori verranno ritornati con un ritardo di 1 secondo tra l'uno e l'altro.

Esempio reale

Finora abbiamo visto esempi molto semplici, tanto per prendere confidenza. Vediamo ora un esempio più complesso.

Ci sono molti servizi online che restituiscono dati paginati. Per esempio, quando abbiamo bisogno

E' un modello molto comune, non solo per gli utenti, ma per qualsiasi cosa. Ad esempio, GitHub consente di paginare i commit di un repository.

- Eseguiamo una richiesta alla URL nella forma `https://api.github.com/repos/<repo>/commits`.
- Il server risponde con un JSON di 30 commit e ci ritorna anche un link alla pagina successiva.
- Dopodiché, possiamo usare tale link per la richiesta successiva, ottenendo le commit successive.

Ci piacerebbe, tuttavia, avere una API più semplice: un oggetto iteratore per le commit, che

```

```js
let repo = "javascript-tutorial/en.javascript.info"; // repository GitHub dal quale ottenere le commit
```

```

```
```js
for await (let commit of fetchCommits("username/repository")) {
 // process commit
}
```

```

Ci piacerebbe un'invocazione, come ad esempio `fetchCommits(repo)` per ottenere le commit, che e Grazie ai generatori asincroni diventa piuttosto semplice da implementare:

```
```js
async function* fetchCommits(repo) {
 let url = `https://api.github.com/repos/${repo}/commits`;

 while (url) {
 const response = await fetch(url, {
 // (1)
 headers: { "User-Agent": "Our script" }, // github richiede un header user-agent
 });

 const body = await response.json(); // (2) la risposta è un JSON (array di commit)

 // (3) la URL della pagina successiva è negli header, dunque dobbiamo estrarla
 let nextPage = response.headers.get("Link").match(/<(.*)>; rel="next"/);
 nextPage = nextPage && nextPage[1];

 url = nextPage;

 for (let commit of body) {
 // (4) restituisce (yield) le commit una ad una fino alla fine della pagina
 yield commit;
 }
 }
}
```

```

1. Utilizziamo il metodo [fetch](info:fetch) del browser per ottenere i dati dalla URL remota. Q
2. Il risultato di fetch viene interpretato come un JSON, altra caratteristica del metodo `fetch`
3. Dovremmo, quindi, ottenere la URL alla pagina successiva dal header `Link` della risposta. Si
4. Infine, ritorniamo tutte le commit ricevute tramite `yield` e, una volta terminate, la succes

Un esempio di utilizzo (visualizza gli autori delle commit nella console):

```
```js run
(async () => {
 let count = 0;

 for await (const commit of fetchCommits(
 "javascript-tutorial/en.javascript.info"
)) {
 console.log(commit.author.login);

 if (++count == 100) {
 // let's stop at 100 commits
 break;
 }
 }
})();

// Note: If you are running this in an external sandbox, you'll need to paste here the function

```

Questo è esattamente quello che volevamo. I meccanismi interni delle richieste paginate sono:

```
Riepilogo
```

I normali iteratori e generatori funzionano bene con dati che non richiedono tempo per essere generati.

Quando i dati ci arrivano in modo asincrono, con dei ritardi, possiamo usare iteratori e generatori.

Differenze sintattiche tra iteratori sincroni e asincroni:

	Iteratori	Iteratori asincroni
-----	-----	-----
Metodo che ci fornisce l'iteratore   `Symbol.iterator`   `Symbol.asyncIterator`	valore ritornato da `next()`   qualsiasi valore   `Promise`	

Differenze sintattiche tra generatori asincroni e sincroni:

	Generators	Async generators
-----	-----	-----
Dichiarazione   `function*`   `async function*`	`generator.next()` ritorna...   `{value:..., done: true/false}`   `Promise` che risolve ritornando	

Nello sviluppo web incontriamo spesso flussi di dati che vengono ritornati "in gruppi". Per esempio:

Possiamo usare i generatori asincroni per processare questo tipo di dati, ma vale anche la pena di saperne di più.

Le Streams API non fanno parte del linguaggio JavaScript standard.

## Moduli

### Moduli, introduzione

Quando la nostra applicazione cresce di dimensione, vogliamo dividerla in diversi file, chiamati "moduli"(modules). Un modulo solitamente contiene una classe o una libreria di funzioni.

Per molto tempo JavaScript è esistito senza una vera sintassi per i moduli nel linguaggio. Questo non era un problema, dato che inizialmente gli script erano piccoli e semplici, e quindi non c'era esigenza.

Ma gli script man mano diventarono più grandi e complessi, di conseguenza la comunità inventò vari sistemi per organizzare il codice in moduli, come librerie speciali che gestivano il caricamento di moduli su richiesta.

Per esempio:

- [AMD](#) – uno dei più vecchi sistemi per la gestione di moduli, inizialmente implementato dalla libreria [require.js](#).
- [CommonJS](#) – il sistema per la gestione di moduli creato per node.js server.
- [UMD](#) – un'altro sistema di gestione di moduli, che è stato suggerito come metodo universale, compatibile sia con AMD sia con CommonJS.

Ormai tutti questi sistemi vengono lentamente abbandonati, anche se ancora possono essere trovati in vecchi script.

Il sistema per la gestione dei moduli nel linguaggio è stato standardizzato nel 2015, e si è gradualmente evoluto da quel momento in poi. Ora è supportato da tutti i browser principali e all'interno di node.js, da adesso in poi sarà questo il sistema che studieremo.

## Che cos'è un modulo?

Un modulo è semplicemente un file. Uno script è un modulo.

I moduli possono caricarsi a vicenda e utilizzare speciali direttive `export` e `import` per scambiarsi funzionalità, chiamando le funzioni da un modulo all'altro:

- `export` contrassegna variabili e funzioni che devono essere accessibili dall'esterno del modulo.
- `import` permette d'importare funzionalità da altri moduli.

Ad esempio, se abbiamo un file `sayHi.js` possiamo rendere utilizzabile all'esterno la funzione(esportarla) in questo modo:

```
// sayHi.js
export function sayHi(user) {
 alert(`Ciao, ${user}!`);
}
```

...Successivamente un'altro file può importarla e usarla in questo modo:

```
// main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('John'); // Ciao, John!
```

La direttiva `import` carica il modulo presente al percorso `./sayHi.js`, relativamente al file corrente, e assegna la funzione esportata `sayHi` alla variabile corrispondente.

Ora proviamo ad utilizzare l'esempio all'interno del browser.

Dato che i moduli utilizzano parole chiavi e funzionalità speciali, dobbiamo comunicare al browser che lo script deve essere trattato come un modulo, utilizzando l'attributo `<script type="module">`.

In questo modo:

[https://plnkr.co/edit/GIIJjVb4RKboN60i?p=preview ↗](https://plnkr.co/edit/GIIJjVb4RKboN60i?p=preview)

Il browser recupera ed elabora automaticamente il modulo importato (e i suoi import se necessario), e infine esegue lo script.

## Modules work only via HTTP(s), not in local files

Se provate ad aprire una pagina web in locale, tramite il protocollo `file://`, scoprirete che le direttive `import/export` non funzionano. Per questo vanno utilizzati dei web-server locali come [static-server ↗](#) oppure utilizzando la funzionalità “live server” dell’editor di codice, come quello di VS Code [Live Server Extension ↗](#) per testare i moduli.

## Funzionalità principali dei moduli

Cosa c’è di diverso nei moduli rispetto ai “normali” script?

Ci sono delle funzionalità aggiunte, valide sia per codice JavaScript all’interno dei browser sia per quello eseguito lato server.

### Hanno sempre “use strict”

I moduli lavorano sempre in *strict mode*, automaticamente. Ad esempio, assegnare un valore ad una variabile non dichiarata genera un’errore.

```
<script type="module">
 a = 5; // error
</script>
```

### Visibilità delle variabili (scope) all’interno dei moduli

Ogni modulo ha la propria visibilità delle variabili di massimo livello. In altre parole le variabili dichiarate a livello maggiore all’interno di un modulo non sono visibili negli altri script.

In the example below, two scripts are imported, and `hello.js` tries to use `user` variable declared in `user.js`. It fails, because it’s a separate module (you’ll see the error in the console):

[https://plnkr.co/edit/AX16IKOOHnTNI3Qe?p=preview ↗](https://plnkr.co/edit/AX16IKOOHnTNI3Qe?p=preview)

I moduli dovrebbero eseguire l’`export` di ciò che vogliono che sia accessibile dall’esterno e l’`import` ciò di cui hanno bisogno.

- `user.js` dovrebbe esportare la variabile `user`.
- `hello.js` dovrebbe importarla dal modulo `user.js`.

In altre parole, con i moduli usiamo import/export invece di affidarci alle variabili globali.

Questa è la versione corretta:

[https://plnkr.co/edit/AfULNYDqLH9SxqH1?p=preview ↗](https://plnkr.co/edit/AfULNYDqLH9SxqH1?p=preview)

All’interno del browser, se parliamo di pagine HTML, esiste uno scope indipendente all’interno di ogni `<script type="module">`.

Ecco due script sulla stessa pagina, entrambi `type="module"`. Non vedono le variabili di primo livello l’uno dell’altro:

```
<script type="module">
 // Le variabili saranno visibili solo all'interno di questo modulo
 let user = "John";
```

```
</script>

<script type="module">
 alert(user); // Errore: la variabile user non è definita
</script>
```

**i** **Da notare:**

Nel browser, possiamo creare una variabile globale assegnandola esplicitamente ad una proprietà di `window`, ad esempio `window.user = "John"`.

Così sarà accessibile a tutti gli scripts, sia con `type="module"` che senza.

Detto questo, creare questo genere di variabili è una cattiva pratica, cercate di evitarlo.

## Un modulo viene eseguito solo la prima volta che viene importato

Se lo stesso modulo viene importato in vari altri moduli, il suo codice viene eseguito solo una volta, durante il primo import. Successivamente tutti i suoi exports vengono distribuiti agli altri moduli che la importano.

La valutazione una tantum ha conseguenze importanti di cui dovremmo essere consapevoli.

Vediamo degli esempi.

Prima di tutto, se eseguire un modulo ha altri effetti, come far apparire un messaggio, importare quel modulo più volte lo farà apparire solamente una volta, la prima:

```
// alert.js
alert("Il modulo è stato eseguito!");

// Importiamo lo stesso modulo in file diversi

// 1.js
import `./alert.js`; // Il modulo è stato eseguito!

// 2.js
import `./alert.js`; // (non appare nulla)
```

La seconda importazione non mostra nulla, perché il modulo è già stato valutato.

C'è una regola: il codice del modulo di primo livello dovrebbe essere usato per l'inizializzazione, la creazione di strutture dati interne specifiche del modulo. Se abbiamo bisogno di rendere qualcosa richiamabile più volte, dovremmo esportarlo come una funzione, come abbiamo fatto con `sayHi` sopra.

Vediamo ora un esempio più complesso.

Prendiamo in considerazione un modulo che esporta un oggetto:

```
// admin.js
export let admin = {
 name: "John"
};
```

Nel momento che questo modulo viene importato in più file viene comunque eseguito una sola volta, l'oggetto `admin` viene creato e poi passato a tutti i moduli che lo hanno importato.

Tutti quindi ottengono esattamente lo stesso oggetto `admin`:

```
// 1.js
import {admin} from './admin.js';
admin.name = "Pete";

// 2.js
import {admin} from './admin.js';
alert(admin.name); // Pete

// Entrambi 1.js e 2.js si riferiscono allo stesso oggetto
// I cambiamenti fatti in 1.js sono visibili in 2.js
```

Come puoi vedere, quando `1.js` cambia la proprietà `name` nell'`admin` importato, allora anche `2.js` può vedere il nuovo `admin.name`.

Questo è il motivo per cui il modulo viene eseguito solo una volta. Le esportazioni vengono generate e quindi condivise tra gli importatori, quindi se qualcosa cambia l'oggetto `admin`, gli altri moduli lo vedranno.

**Questo comportamento in realtà è molto utile, perché ci permette di *configurare* i moduli.**

In altre parole, un modulo può fornire una funzionalità generica che necessita di una configurazione. Per esempio, l'autenticazione necessita di credenziali. Quindi può esportare un oggetto di configurazione aspettandosi che il codice esterno gli venga assegnato.

Ecco lo schema classico:

1. Un modulo esporta alcuni strumenti di configurazione, ad es. un oggetto di configurazione.
2. Alla prima importazione lo inizializziamo, impostando le sue proprietà. Potrebbe farlo lo script di livello più alto.
3. Ulteriori importazioni utilizzano il modulo.

Per fare un esempio, il modulo `admin.js` può fornire alcune funzionalità (ad esempio l'autenticazione), ma si aspetta di ricevere le credenziali all'interno dell'oggetto `config` dall'esterno:

```
// admin.js
export let config = { };

export function sayHi() {
 alert(`Sono pronto, ${config.user}!`);
}
```

Qui, `admin.js` esporta l'oggetto `config` (inizialmente vuoto, ma potrebbe anche avere proprietà predefinite).

Quindi in `init.js`, il primo script della nostra applicazione, importiamo `config` ed impostiamo `config.user`:

```
// init.js
import {config} from './admin.js';
config.user = "Pete";
```

...Ora il modulo `admin.js` è configurato.

Le successive importazioni chiamarlo, e verrà mostrato correttamente lo user corrente:

```
// another.js
import {sayHi} from './admin.js';

sayHi(); // Sono pronto, Pete!
```

## import.meta

L'oggetto `import.meta` contiene le informazioni riguardanti il modulo corrente.

Il suo contenuto dipende dall'ambiente di esecuzione. Nel browser, contiene l'URL dello script o dell'attuale pagina web se inserito all'interno dell'HTML:

```
<script type="module">
 alert(import.meta.url); // script URL
 // per gli inline script è l'URL della pagina corrente
</script>
```

## All'interno di un modulo, “this” non è definito (`undefined`)

Questa è una funzionalità minore, ma per completezza dobbiamo menzionarla.

In un modulo, Il `this` di livello maggiore non è definito (`undefined`).

Facciamo il confronto con uno script che non è un modulo, dove `this` è un oggetto globale.

```
<script>
 alert(this); // window
</script>

<script type="module">
 alert(this); // undefined
</script>
```

## Funzionalità specifiche nel browser

Ci sono diverse funzionalità specifiche dei moduli utilizzati all'interno del browser con `type="module"`.

Potresti voler saltare questa sezione se stai leggendo per la prima volta , oppure se non hai intenzione di usare JavaScript all'interno del browser.

### I moduli sono caricati in modo differente

I moduli vengono sempre reputati script differenti, stesso effetto dell'attributo `defer` (descritto nel capitolo [Scripts: async, defer](#)) sia per gli script esterni che per quelli interni.

In altre parole:

- Il download di un modulo esterno `<script type="module" src="...>` non blocca l'elaborazione dell'HTML, vengono caricati in parallelo insieme alle altre risorse.
- I moduli attendono fino al momento in cui l'HTML è pronto (anche se sono molto piccoli e possono essere elaborati più velocemente dell'HTML), e poi vengono eseguiti.
- L'ordine relativo degli script viene mantenuto: gli script che appaiono prima nel documento vengono eseguiti per primi.

Come conseguenza, i moduli “vedono” sempre la pagina HTML completamente caricata, inclusi gli elementi sotto di essi.

Ad esempio:

```
<script type="module">
 alert(typeof button); // Object: lo script può 'vedere' il bottone sottostante
 // dato che il modulo viene caricato in modo differente, viene eseguito solo dopo che l'intera
</script>
```

Confrontiamo lo script normale:

```
<script>
 alert(typeof button); // Error: button is undefined, lo script non riesce a vedere il bottone
 // Gli script normali vengono eseguiti immediatamente, prima che il resto della pagina venga processata
</script>

<button id="button">Button</button>
```

Da notare: il secondo script viene eseguito per primo! Infatti vedremo prima `undefined`, e dopo `object`.

Questo accade proprio perché i moduli sono differiti, e quindi attendono che tutto il documento venga processato, al contrario, gli script normali vengono eseguiti immediatamente e di conseguenza vediamo l'output del secondo script per primo.

Quando utilizziamo i moduli, dobbiamo porre attenzione al fatto che la pagina HTML appare mentre viene caricata, e i moduli JavaScript vengono eseguiti successivamente al caricamento, di conseguenza l'utente potrebbe vedere la pagina *prima* che l'applicazione JavaScript sia pronta. Alcune funzionalità potrebbero in questo modo non funzionare immediatamente. Per questo motivo è opportuno inserire degli indicatori di caricamento, o comunque assicurarsi che i visitatori non vengano confusi da questi possibili comportamenti.

### Async funziona sui moduli scritti inline

Per gli script normali l'attributo `async` funziona solamente sugli script esterni, Gli script caricati in modo asincrono (Async) vengono eseguiti immediatamente e indipendentemente dagli altri script e del documento HTML.

Per i moduli `async` può essere utilizzato sempre.

Ad esempio, lo script seguente è dichiarato asincrono, e quindi non aspetta nulla e viene eseguito.

Esegue l'import (recupera `./analytics.js`) e procede quando è pronto, anche se il documento HTML non è completo, o se gli altri script sono ancora in attesa.

Questo comportamento è ottimo per le funzionalità che non dipendono da nulla, come contatori, pubblicità e altro.

```
<!-- tutte le dipendenze vengono recuperate (analytics.js), e lo script viene eseguito -->
<!-- non aspetta che il documento o altri <script> tag siano pronti -->
<script async type="module">
 import {counter} from './analytics.js';

 counter.count();
</script>
```

## Script esterni

Gli script esterni che vengono segnalati come moduli, `type="module"`, sono diversi sotto due aspetti:

1. Più script esterni con lo stesso `src` vengono eseguiti solo una volta:

```
<!-- lo script my.js viene recuperato ed eseguito solo una volta -->
<script type="module" src="my.js"></script>
<script type="module" src="my.js"></script>
```

2. Gli script esterni che vengono recuperati da origini diverse (ad esempio un sito diverso) hanno bisogno delle intestazioni [CORS ↗](#), come descritto nel capitolo [Fetch: Cross-Origin Requests](#). In altre parole, se un modulo viene recuperato da un'altra fonte il server remoto deve fornire un header (intestazione) `Access-Control-Allow-Origin` dandoci il “permesso” di recuperare lo script.

```
<!-- another-site.com deve fornire Access-Control-Allow-Origin -->
<!-- altrimenti lo script non verrà eseguito -->
<script type="module" src="http://another-site.com/their.js"></script>
```

Questo meccanismo permette di avere una maggiore sicurezza.

### Non è possibile usare moduli “bare”

All'interno del browser, `import` accetta percorsi URL relativi o assoluti. Moduli senza nessun percorso specificato vengono chiamati moduli “bare”. Questi moduli non vengono accettati da `import` all'interno del browser.

Ad esempio, questo `import` non è valido:

```
import {sayHi} from 'sayHi'; // Errore, modulo "bare"
// Il modulo deve avere un percorso, es. './sayHi.js' od ovunque si trovi il modulo
```

Alcuni ambienti, come Node.js o tools per creare bundle accettano moduli bare, senza nessun percorso (path), dato che hanno metodologie per trovare e collegare i moduli. Al contrario i browser ancora non supportano i moduli bare.

### Compatibilità, “nomodule”

I vecchi browser non comprendono l'attributo `type="module"`. Gli script di una tipologia non conosciuta vengono semplicemente ignorati. Proprio per questo è possibile prevedere uno script di riserva usando l'attributo `nomodule`:

```

<script type="module">
 alert("Viene eseguito nei browser moderni");
</script>

<script nomodule>
 alert("I browser moderni conoscono sia type=module sia nomodule, quindi ignorano questo script");
 alert("I browser più vecchi ignorano i tipi di script che non conoscono come type=module, ma e");
</script>

```

## Strumenti per il building

Nella realtà, i moduli vengono raramente usati all'interno del browser in modo diretto. Normalmente, vengono uniti insieme con tool specifici come [Webpack ↗](#) e portati nel server di produzione.

Uno dei benefici di usare i “bundlers” – ci permettono più controllo su come i moduli vengono gestiti, ad esempio permettendoci di usare moduli “bare” e moduli CSS/HTML.

I tool per il building si comportano nel modo seguente:

1. Prendono un modulo “principale”, quello che era inteso per essere inserito in `<script type="module">`.
2. Analizza tutte le sue dipendenze: che moduli importa, cosa viene importato dai metodi importati etc...
3. Costruisce un singolo file con tutti i moduli (o più file, può essere impostato), sostituendo le chiamate `import` con funzioni del bundler. In questo modo può supportare anche moduli “speciali” come quelli CSS/HTML.
4. Durante il processo altre trasformazioni e ottimizzazioni possono essere eseguite:
  - Parti di codice che non possono essere raggiunte vengono eliminate.
  - `export` non utilizzati vengono rimossi (“tree-shaking”).
  - Parti di codice tipicamente utilizzati durante lo sviluppo come `console` e `debugger` rimosse.
  - Le sintassi più moderne di JavaScript vengono sostituite con funzionalità equivalenti più vecchie e compatibili usando [Babel ↗](#).
  - Il file risultante viene ridotto al minimo (minified), gli spazi superflui rimossi, i nomi delle variabili sostituiti con nomi corti etc...

Quindi se usiamo questa tipologia di strumenti, allora gli script vengono raggruppati in un singolo script (o pochi file), `import/export` sostituiti con speciali funzioni in modo che lo script finale non contenga più nessun `import/export`, non richiede l'uso di `type="module"` e può essere utilizzato come un normale script:

```

<!-- Assumendo che abbiamo ottenuto bundle.js da un tool come Webpack -->
<script src="bundle.js"></script>

```

Appurato questo, moduli in modo nativo possono comunque essere usati. Non useremo tools come Webpack qui: se necessario potrai configurarlo successivamente.

## Riepilogo

Per ricapitolare, i concetti principali sono:

1. Un modulo è un file. Per far funzionare `import/export`, il browser ha bisogno di `<script type="module">`. I moduli hanno alcune differenze:
  - Vengono eseguiti in modo differito automaticamente
  - Async funziona sui moduli in linea
  - Per caricare moduli esterni provenienti da un'origine diversa (un altro dominio/protocollo/porta), sono necessarie le intestazioni CORS.
  - I moduli esterni duplicati vengono ignorati (un modulo esterno viene eseguito solo la prima volta che viene importato)
2. I moduli hanno il loro livello di visibilità delle variabili (scope) e si scambiano funzionalità attraverso `import/export`.
3. I moduli utilizzano sempre `use strict` automaticamente.
4. Il codice di un modulo viene eseguito solamente una volta. Le esportazioni (`export`) vengono create un'unica volta e condivise con tutti i moduli che le importano.

Quando utilizziamo i moduli, ogni modulo implementa una certa funzionalità e la esporta. Successivamente utilizziamo `import` per importare quella funzionalità e utilizzarla dove è necessario. I browser caricano e eseguono lo script automaticamente.

In produzione, di solito si tende a usare tool detti “bundlers” come [Webpack ↗](#) per unire insieme tutti i moduli per maggiori prestazioni, compatibilità e altro.

Nel prossimo capitolo vedremo più esempi di moduli, e come le cose possono essere importate ed esportate.

## Export e Import

Le direttive `export` e `import` esistono sotto forma di diverse varianti.

Nell'articolo precedente, ne abbiamo visto un utilizzo molto semplice, ora esploriamo più esempi.

### Export prima della dichiarazione

Possiamo etichettare una qualsiasi dichiarazione come esportata, con il prefisso `export`, sia che questa sia una variabile, una funzione o una classe.

Ad esempio, i seguenti exports sono tutti validi:

```
// esportiamo un array
export let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// esportiamo una costante
export const MODULES_BECAME_STANDARD_YEAR = 2015;

// esportiamo una classe
export class User {
 constructor(name) {
 this.name = name;
 }
}
```

### **i Non c'è bisogno del punto e virgola dopo l'export di una classe/funzione**

Da notare che il termine `export` prima di una funzione, non la rende un'[espressione di funzione](#). Rimane sempre una dichiarazione di funzione, viene semplicemente esportata.

Molte *style guides* di JavaScript sconsigliano l'utilizzo del punto e virgola dopo la dichiarazione di una classe o funzione.

Questo è il motivo per cui non c'è alcun bisogno del punto e virgola dopo `export class` e `export function`:

```
export function sayHi(user) {
 alert(`Hello, ${user}!`);
} // no ; at the end
```

## Export oltre alle dichiarazioni

Inoltre, possiamo inserire `export` separatamente.

In questo caso, prima dichiariamo e successivamente esportiamo:

```
// say.js
function sayHi(user) {
 alert(`Hello, ${user}!`);
}

function sayBye(user) {
 alert(`Bye, ${user}!`);
}

export {sayHi, sayBye}; // una lista di variabili esportate
```

...Oppure, tecnicamente, potremmo inserire `export` anche prima delle dichiarazioni delle funzioni.

## Import \*

Soltanente, inseriamo una lista di ciò che vogliamo importare tra le parentesi graffe `import { . . . }`, in questo modo:

```
// main.js
import {sayHi, sayBye} from './say.js';

sayHi('John'); // Hello, John!
sayBye('John'); // Bye, John!
```

Ma nel caso in cui ci fossero molti import, potremmo importare tutto come un oggetto, utilizzando `import * as <obj>`, ad esempio:

```
// main.js
import * as say from './say.js';

say.sayHi('John');
say.sayBye('John');
```

A prima vista, “importa tutto” sembra una cosa comoda, veloce da scrivere, quindi perché dovremmo esplicitare una lista di ciò che vogliamo importare?

Ecco alcune ragioni valide.

1. Gli strumenti moderni di build ([webpack](#) ed altri) impacchettano i moduli in modo da ottimizzarli, velocizzarne il caricamento e rimuovere le cose inutili.

Ipotizziamo di aggiungere una libreria di terze parti `say.js` al nostro progetto, che contiene molte funzioni:

```
// say.js
export function sayHi() { ... }
export function sayBye() { ... }
export function becomeSilent() { ... }
```

Ora utilizziamo solamente una delle funzioni di `say.js` nel nostro progetto:

```
// main.js
import {sayHi} from './say.js';
```

...Ora l'ottimizzazione se ne accorgerà e rimuoverà le altre funzioni dal codice compilato, rendendo il pacchetto più piccolo. Questo viene chiamato “tree-shaking”.

2. Elencare esplicitamente ciò che vogliamo importare ci fornisce nomi più brevi: `sayHi()` piuttosto di `say.sayHi()`.
3. Elencare esplicitamente ciò che vogliamo importare ci fornisce una migliore visione della struttura del codice: cosa viene utilizzato e dove. Rende la manutenibilità e il refactoring del codice più semplice.

## Import “as”

Possiamo anche utilizzare la keyword `as` per importare con nomi differenti.

Ad esempio, importiamo `sayHi` in una variabile locale `hi` per brevità, e `sayBye` come `bye`:

```
// main.js
import {sayHi as hi, sayBye as bye} from './say.js';

hi('John'); // Hello, John!
bye('John'); // Bye, John!
```

## Export “as”

Una sintassi molto simile è disponibile per `export`.

Esportiamo le funzioni come `hi` e `bye`:

```
// say.js
...
export {sayHi as hi, sayBye as bye};
```

Ora `hi` e `bye` sono i nomi ufficiali per chi le vede esternamente, quelli da utilizzare per gli import:

```
// main.js
import * as say from './say.js';

say.hi('John'); // Hello, John!
say.bye('John'); // Bye, John!
```

## Export default

In pratica, esistono principalmente due tipi di moduli.

1. Moduli che contengono una libreria: pacchetti di funzioni, come `say.js` visto sopra.
2. Moduli che dichiarano una singola entità: esempio un modulo `user.js` che esporta solamente `class User`.

Nella maggior parte dei casi, si preferisce il secondo approccio, in modo tale che ogni “cosa” stia nel suo modulo.

Naturalmente, questa pratica richiede l'utilizzo di molti files, poiché ogni cosa richiede il suo modulo, ma questo non è un problema. In realtà, la navigazione del codice diventa più semplice se tutti i file hanno dei nomi descrittivi e sono ben strutturati all'interno di cartelle.

I moduli forniscono una speciale sintassi, `export default` (“export di default”), per rendere la pratica “una cosa per modulo” più elegante.

E' sufficiente inserire `export default` prima dell'entità da esportare:

```
// user.js
export default class User { // aggiungiamo "default"
 constructor(name) {
 this.name = name;
 }
}
```

Può esserci solamente un `export default` per file.

...E possiamo importarlo senza le parentesi graffe:

```
// main.js
import User from './user.js'; // non {User}, semplicemente User
```

```
new User('John');
```

Gli import senza parentesi graffe sono più eleganti. Un errore comune quando si inizia ad utilizzare i moduli è quello di dimenticarsi le parentesi graffe. Quindi, ricorda, `import` richiede le parentesi graffe per i named export, ma non le richiedere per i default export.

Named export	Default export
<code>export class User {...}</code>	<code>export default class User {...}</code>
<code>import {User} from ...</code>	<code>import User from ...</code>

Tecnicamente, potremmo avere sia il default che il named export nello stesso modulo, ma in pratica gli sviluppatori non lo fanno. Un modulo può essere named export o default export.

Poiché possiamo utilizzare al massimo un default export per file, l'entità esportata non richiede alcun nome.

Ad esempio, questi sono tutti export default validi:

```
export default class { // la classe non ha un nome
 constructor() { ... }
}
```

```
export default function(user) { // la funzione non ha un nome
 alert(`Hello, ${user}!`);
}
```

```
// esportiamo un valore, senza dichiarare la variabile
export default ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
```

Non dare un nome va bene, poiché abbiamo un solo `export default` per file, quindi `import` senza parentesi graffe, sa cosa importare.

Senza `default`, questo export genererebbe un errore:

```
export class { // Errore! (non-default export richiedono un nome)
 constructor() {}
}
```

## Il nome di “default”

In alcune situazioni la keyword `default` per fare riferimento al default export.

Ad esempio, per esportare una funzione separatamente dalla sua definizione:

```
function sayHi(user) {
 alert(`Hello, ${user}!`);
}
```

```
// equivalente a "export default"
export {sayHi as default};
```

Oppure, un'altra situazione, immaginiamo di avere un modulo `user.js` che esporta una “cosa” di default, ed un paio di altre con nome (accade raramente, ma è possibile):

```
// user.js
export default class User {
 constructor(name) {
 this.name = name;
 }
}

export function sayHi(user) {
 alert(`Hello, ${user}!`);
}
```

Vediamo come importare il default export insieme a quello named:

```
// main.js
import {default as User, sayHi} from './user.js';

new User('John');
```

E, infine, se importiamo tutto `*` come un oggetto, allora la proprietà di `default` corrisponde al default export:

```
// main.js
import * as user from './user.js';

let User = user.default; // il default export
new User('John');
```

### Una parola contro il default exports

I named export sono esplicativi, elencano esattamente il nome di ciò che vogliono importare. Avere in chiaro questa informazione (il nome), è sempre una buona cosa.

I named export ci forzano ad utilizzare il nome esatto di ciò che vogliamo importare:

```
import {User} from './user.js';
// import {MyUser} non funzionerebbe, il nome deve essere {User}
```

...Mentre per un default export, possiamo decidere il nome in fase di importazione:

```
import User from './user.js'; // funziona
import MyUser from './user.js'; // funziona
// potrebbe essere anche import Anything... e funzionerebbe comunque
```

Quindi i membri del team possono utilizzare nomi differenti per importare le stesse cose, e questa non è una buona cosa.

Soltamente, per evitare questo problema, e mantenere il codice consistente, ci si pone come regola che le variabili importate debbano corrispondere ai nomi dei file, esempio:

```
import User from './user.js';
import LoginForm from './loginForm.js';
import func from '/path/to/func.js';
...
```

Ancora, alcuni team ritengono questo uno svantaggio piuttosto serio dei default export. Quindi preferiscono utilizzare i named exports. Anche se viene esportata solamente una cosa, questa viene esportata con un nome, senza `default`.

Questo rende il re-export (vedi sotto) più semplice.

## Re-export

La sintassi di “re-export”, `export ... from ...` ci consente di importare cose ed esportarle immediatamente (eventualmente con un altro nome), come nell'esempio:

```
export {sayHi} from './say.js'; // re-export sayHi

export {default as User} from './user.js'; // re-export default
```

Perché questo dovrebbe essere necessario? Vediamolo con un esempio pratico.

Immaginiamo di scrivere un “package” (pacchetto): una cartella contenente molti moduli, con alcune funzionalità esportate esternamente (strumenti come NPM ci consentono di pubblicare e distribuire questi pacchetti, ma non li utilizzeremo), e con molti moduli che sono semplicemente “helpers”, per uso interno in altri moduli del package.

La struttura dei file potrebbe essere qualcosa del genere:

```
auth/
 index.js
 user.js
 helpers.js
 tests/
 login.js
 providers/
 github.js
 facebook.js
 ...
...
```

Vorremmo poi esporre le funzionalità del package con un singolo entry point (punto di ingresso).

In altre parole, la persona che volesse utilizzare il nostro package, dovrebbe importare solamente il “main file” `auth/index.js`.

Come vediamo qui:

```
import {login, logout} from 'auth/index.js'
```

Il "main file", `auth/index.js` esporta tutte le funzionalità che vogliamo fornire con il nostro package.

L'idea è che gli esterni, gli altri programmatori che utilizzano il nostro package, non debbano preoccuparsi della struttura interna e di cercare files tra le cartelle in esso contenute.

Esportiamo solamente ciò che è necessario in `auth/index.js` e teniamo il resto nascosto da occhi indiscreti.

Poiché le funzionalità di export sono sparpagliate nel package, possiamo importarle in `auth/index.js` ed esportarle da lì:

```
// auth/index.js

// importiamo login/logout e li esportiamo immediatamente
import {login, logout} from './helpers.js';
export {login, logout};

// importiamo default come User e lo esportiamo
import User from './user.js';
export {User};

...
```

Ora, gli utenti del nostro package possono `import {login} from "auth/index.js"`.

La sintassi `export ... from ...` è semplicemente una notazione più breve per questi import-export:

```
// auth/index.js
// re-export login/logout
export {login, logout} from './helpers.js';

// re-export il default export as User
export {default as User} from './user.js';
...
```

Un'importante differenza tra `export ... from ...` e `import/export`, è che i moduli ri-esportati non sono disponibili nel file corrente. Quindi, guardando l'esempio sopra `auth/index.js`, non possiamo utilizzare le funzioni ri-esportate `login/logout`.

## Re-exporting il default export

I default export richiedono una gestione separata quando li ri-esportiamo.

Ipotizziamo di avere `user.js` con `export default class User`, e di volerlo ri-esportare:

```
// user.js
export default class User {
 // ...
}
```

Potremmo incontrare due problemi:

1. `export User from './user.js'` non funziona. Genererebbe un errore sintattico.

Per ri-esportare il default export, dobbiamo scrivere `export {default as User}`, come nell'esempio sopra.

2. `export * from './user.js'` ri-esporta solamente i named exports, ma ignora quelli di default.

Nel caso in cui volessimo ri-esportare sia i named che i default export, allora avremmo bisogno di due istruzioni:

```
export * from './user.js'; // per ri-esportare i named exports
export {default} from './user.js'; // per ri-esportare i default export
```

Queste stranezze nella ri-esportazione di un default export sono tra le ragioni del per cui alcuni sviluppatori non amano i default export, ma preferiscono quelli named.

## Riepilogo

Vediamo tutti i tipi di `export` che abbiamo studiato in questo articolo e nei precedenti.

Puoi controllarli tu stesso, leggendoli e provando a ricordarne il significato:

- Prima della dichiarazione di una classe/funzione/...:
  - `export [default] class/function/variable ...`
- Export indipendente:
  - `export {x [as y], ...}.`
- Re-export:
  - `export {x [as y], ...} from "module"`
  - `export * from "module"` (non ri-esporta i default export).
  - `export {default [as y]} from "module"` (ri-esporta i default export).

Import:

- Importare Named exports:
  - `import {x [as y], ...} from "module"`
- Importare il default export:
  - `import x from "module"`
  - `import {default as x} from "module"`
- Importare tutto:
  - `import * as obj from "module"`
- Importare il modulo (il suo codice viene eseguito), ma senza assegnare i suoi exports ad alcuna variabile:
  - `import "module"`

Possiamo inserire le istruzioni di `import/export` in cima o in coda allo script, non ha importanza.

Quindi, tecnicamente, questo codice funziona:

```
sayHi();
// ...
import {sayHi} from './say.js'; // import alla fine del file
```

Nella pratica gli import sono posizionati all'inizio del file, ma solo per comodità.

**Da notare che le istruzioni di import/export non funzionano all'interno di `{ . . . }`.**

Un import condizionale, come questo, non funziona:

```
if (something) {
 import {sayHi} from "./say.js"; // Errore: import deve essere nello scope principale
}
```

...E nel caso in cui volessimo veramente importare qualcosa secondo una condizione? Oppure al momento giusto? Ad esempio, caricando un modulo in seguito ad una richiesta, nel momento in cui è veramente necessario?

Vederemo i dynamic imports nel prossimo articolo.

## Dynamic imports

Le istruzioni di export ed import che abbiamo visto nei capitoli precedenti sono detti “statici”. La sintassi è molto semplice e rigorosa.

Come prima cosa, non possiamo generare dinamicamente parametri di `import`.

Il percorso al modulo deve essere una stringa, non può essere una chiamata a funzione. Questo non funzionerebbe:

```
import ... from getModuleName(); // Errore, sono ammesse solamente string
```

Secondo, non possiamo importare a run-time in base a determinate condizioni:

```
if(...) {
 import ...; // Errore, non è possibile farlo!
}

{
 import ...; // Errore, non possiamo scrivere gli import all'interno di nessun blocco
}
```

Questo accade perché `import/export` mirano a fornire uno scheletro per la struttura del codice. Questa è una buona cosa, poiché la struttura del codice può essere analizzata, i moduli possono essere raccolti and impacchettati in un singolo file (grazie ad alcuni strumenti) e gli export inutilizzati possono essere rimossi (“tree-shaken”). Questo è possibile solamente perché la struttura degli imports/exports è semplice e preimpostata.

Ma come possiamo importare un modulo dinamicamente, a seconda delle necessità?

## L'espressione import()

L'espressione `import(module)` carica il modulo e ritorna una promise, che si risolve in un oggetto che contiene tutti gli export del modulo. Può essere quindi invocata in un qualsiasi punto del codice.

Possiamo utilizzarla dinamicamente ovunque, ad esempio:

```
let modulePath = prompt("Which module to load?");

import(modulePath)
 .then(obj => <module object>)
 .catch(err => <loading error, e.g. if no such module>)
```

oppure, potremmo utilizzare `let module = await import(modulePath)` se ci troviamo all'interno di una funzione asincrona.

Ad esempio, se abbiamo il seguente modulo `say.js`:

```
// say.js
export function hi() {
 alert(`Hello`);
}

export function bye() {
 alert(`Bye`);
}
```

...Allora il dynamic import può essere scritto così:

```
let {hi, bye} = await import('./say.js');

hi();
bye();
```

Oppure, se `say.js` ha un default export:

```
// say.js
export default function() {
 alert("Module loaded (export default)!");
}
```

...Quindi, per potervi accedere, possiamo utilizzare la proprietà `default` dell'oggetto:

```
let obj = await import('./say.js');
let say = obj.default;
// o, in una riga: let {default: say} = await import('./say.js');

say();
```

Qui vediamo l'esempio completo:

[https://plnkr.co/edit/nOcKVpSFXFbyQPH2?p=preview ↗](https://plnkr.co/edit/nOcKVpSFXFbyQPH2?p=preview)

**i Da notare:**

I dynamic import funzionano negli script regolari, non richiedono `script type="module"`.

**i Da notare:**

Anche se `import()` sembra una chiamata a funzione, in realtà è una speciale sintassi che utilizza le parentesi (in modo simile a `super()`).

Quindi non possiamo copiare `import` in una variabile o utilizzare `call/apply`. Non è una funzione.

## Miscellaneous

### Proxy e Reflect

Un oggetto `Proxy` racchiude un altro oggetto e ne intercetta le operazioni, come quelle di lettura/scrittura e molte altre; può eventualmente gestirle a modo suo oppure, in maniera del tutto trasparente, lasciare che sia l'oggetto ad occuparsene.

I proxy vengono utilizzati da molte librerie ed alcuni framework per browsers. Ne vedremo molte applicazioni pratiche in questo articolo.

### Proxy

La sintassi:

```
let proxy = new Proxy(target, handler)
```

- `target` – è l'oggetto da racchiudere; può essere qualsiasi cosa, anche una funzione.
- `handler` – configurazione del proxy: un oggetto con “trappole”, metodi che intercettano operazioni. Ad esempio una “trappola” `get` per la lettura di una proprietà di `target`, `set` per la scrittura di una proprietà di `target`, e così via.

Per le operazioni sul `proxy`, se c'è un “trappola” corrispondente in `handler`, allora questa verrà eseguita, e il proxy potrà gestirla, altrimenti l'operazione verrà eseguita su `target`. Come primo esempio, creiamo un proxy senza “trappole”:

```
let target = {};
let proxy = new Proxy(target, {}); // handler vuoto

proxy.test = 5; // scrittura su proxy (1)
alert(target.test); // 5, la proprietà si trova su target!

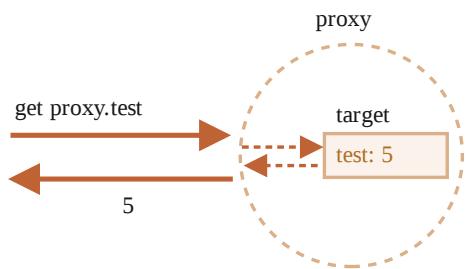
alert(proxy.test); // 5, possiamo leggerla anche dal proxy (2)
```

```
for(let key in proxy) alert(key); // test, l'iterazione funziona (3)
```

Poiché non ci sono “trappole”, tutte le operazioni su `proxy` vengono inoltrate a `target`.

1. Un’operazione di scrittura `proxy.test=` imposta il valore su `target`.
2. Un’operazione di lettura `proxy.test` ritorna il valore da `target`.
3. L’iterazione su `proxy` ritorna valori da `target`.

Come possiamo vedere, senza “trappole”, `proxy` è solamente un contenitore per `target`.



`Proxy` è uno speciale “oggetto esotico”. Non possiede proprietà proprie. Con un `handler` vuoto, le operazioni verranno automaticamente inoltrate a `target`.

Per attivare più funzionalità, aggiungiamo qualche “trappola”.

Cosa possiamo intercettare?

Per molte operazioni sugli oggetti, esiste un così detto “metodo interno” nella specifiche JavaScript che ne descrive il funzionamento a basso livello. Ad esempio `[[Get]]`, il metodo interno per la lettura delle proprietà, e `[[Set]]`, il metodo interno per la scrittura delle proprietà, e così via. Questi metodi vengono utilizzati solamente nelle specifiche, non possiamo invocarli direttamente utilizzandone il nome.

Le trappole “proxy” intercettano le invocazioni di questi metodi. Queste vengono elencate nelle specifiche [Proxy ↗](#) e nella tabella sottostante.

Per ogni metodo interno, esiste una “trappola” in questa tabella: il nome del metodo che possiamo aggiungere al parametro `handler` del `new Proxy` per intercettare l’operazione:

Metodo Interno	Handler	Innescato quando...
<code>[[Get]]</code>	<code>get</code>	lettura di un proprietà
<code>[[Set]]</code>	<code>set</code>	scrittura di un proprietà
<code>[[HasProperty]]</code>	<code>has</code>	operatore <code>in</code>
<code>[[Delete]]</code>	<code>deleteProperty</code>	operatore <code>delete</code>
<code>[[Call]]</code>	<code>apply</code>	invocazione di funzione
<code>[[Construct]]</code>	<code>construct</code>	operatore <code>new</code>
<code>[[GetPrototypeOf]]</code>	<code>getPrototypeOf</code>	<code>Object.getPrototypeOf ↗</code>
<code>[[SetPrototypeOf]]</code>	<code>setPrototypeOf</code>	<code>Object.setPrototypeOf ↗</code>
<code>[[IsExtensible]]</code>	<code>isExtensible</code>	<code>Object.isExtensible ↗</code>
<code>[[PreventExtensions]]</code>	<code>preventExtensions</code>	<code>Object.preventExtensions ↗</code>

Metodo Interno	Handler	Innescato quando...
<code>[[DefineOwnProperty]]</code>	<code>defineProperty</code>	<code>Object.defineProperty</code> ↗ , <code>Object.defineProperties</code> ↗
<code>[[GetOwnProperty]]</code>	<code>getOwnPropertyDescriptor</code>	<code>Object.getOwnPropertyDescriptor</code> ↗ , <code>for..in</code> , <code>Object.keys/values/entries</code>
<code>[[OwnPropertyKeys]]</code>	<code>ownKeys</code>	<code>Object.getOwnPropertyNames</code> ↗ , <code>Object.getOwnPropertySymbols</code> ↗ , <code>for..in</code> , <code>Object.keys/values/entries</code>

### ⚠ Invarianti

JavaScript applica alcune invarianti, ovvero condizioni che devono essere soddisfatte da metodi interni e “trappole”.

Molte di queste sono per i valori di ritorno:

- `[[Set]]` deve ritornare `true` se il valore è stato scritto con successo, altrimenti ritorna `false`.
- `[[Delete]]` deve ritornare `true` se il valore è stato rimosso con successo, altrimenti ritorna `false`.
- ...E così via, vedremo più esempi sotto.

Esistono anche altre invarianti, come:

- `[[GetPrototypeOf]]` , applicata all’oggetto proxy, il quale deve ritornare lo stesso valore di `[[GetPrototypeOf]]` che sarebbe ritornato dall’oggetto target. In altre parole, la lettura del prototype del proxy deve sempre ritornare il prototype dell’oggetto target.

Le “trappole” possono intercettare queste operazioni, ma devono seguire le regole viste.

Le invarianti assicurano che le funzionalità del linguaggio si comportino in maniera corretta e consistente. La lista completa delle invarianti è disponibile [nelle specifiche](#) ↗ . Probabilmente non le violerai, a meno che tu non stia facendo qualcosa di strano.

Vediamo come funzionano con esempi pratici.

## Valore di default con la trappola “get”

La maggior parte delle “trappole” sono dedicate alla lettura/scrittura di proprietà.

Per intercettare la lettura, l’`handler` dovrebbe possedere un metodo `get(target, property, receiver)`.

Verrà innescato quando una proprietà verrà letta, con i seguenti argomenti:

- `target` – è l’oggetto target, quello fornito come primo argomento a `new Proxy`,
- `property` – nome della proprietà,
- `receiver` – se la proprietà target è un getter, allora `receiver` sarà l’oggetto che verrà utilizzato come `this` in questa chiamata. Solitamente è l’oggetto `proxy` stesso (oppure un oggetto che eredita da esso, se stiamo ereditando dal proxy). Per ora non abbiamo bisogno di questo argomento, quindi lo analizzeremo nel dettaglio più avanti.

Utilizziamo `get` per implementare i valori di default di un oggetto.

Costruiremo un array numerico che ritornerà `0` per valori inesistenti.

Solitamente, quando si prova ad accedere ad un elemento non esistente di un array, si ottiene `undefined`, ma noi costruiremo un proxy di un array che ritorna `0` nel caso in cui la proprietà non esistesse:

```
let numbers = [0, 1, 2];

numbers = new Proxy(numbers, {
 get(target, prop) {
 if (prop in target) {
 return target[prop];
 } else {
 return 0; // valore di default
 }
 }
});

alert(numbers[1]); // 1
alert(numbers[123]); // 0 (elemento non esistente)
```

Come possiamo vedere, è molto semplice da fare con una “trappola” `get`.

Possiamo utilizzare un `Proxy` per implementare una logica per i valori di “default”.

Immaginiamo di avere un dizionario, contenente i termini e le rispettive traduzioni:

```
let dictionary = {
 'Hello': 'Hola',
 'Bye': 'Adiós'
};

alert(dictionary['Hello']); // Hola
alert(dictionary['Welcome']); // undefined
```

In questo modo, se non esiste un termine, la lettura dal `dictionary` ritorna `undefined`, ma nella pratica, ritornare un termine non tradotto è generalmente meglio. Quindi facciamo in modo che ritorni il termine non tradotto piuttosto di `undefined`.

Per farlo, costruiremo un contenitore per `dictionary` con un proxy che intreccerà le operazioni di lettura:

```
let dictionary = {
 'Hello': 'Hola',
 'Bye': 'Adiós'
};

dictionary = new Proxy(dictionary, {
 get(target, phrase) { // intercetta la lettura di una proprietà dal dictionary
 if (phrase in target) { // se è contenuto nel dictionary
 return target[phrase]; // ritorna la traduzione
 } else {
 // altrimenti, ritorna il termine non tradotto
 }
 }
});
```

```

 return phrase;
 }
}
});

// Cerchiamo un termine nel dictionary!
// Nel peggio dei casi, questo non sarà tradotto.
alert(dictionary['Hello']); // Hola
alert(dictionary['Welcome to Proxy']); // Welcome to Proxy (nessuna traduzione)

```

**i Da notare:**

Da notare come il proxy sovrascrive la variabile:

```
dictionary = new Proxy(dictionary, ...);
```

Il proxy dovrebbe rimpiazzare completamente l'oggetto target, ovunque. Nessuno dovrebbe più fare riferimento all'oggetto target una volta che questo è stato racchiuso da un proxy. Altrimenti diventerebbe molto facile commettere errori.

## Validazione con la trappola “set”

Ipotizziamo di volere un array di soli numeri. Se viene aggiunto un valore di un altro tipo, dovrebbe venire generato un errore.

La “trappola” `set` si innesca quando si accede in scrittura ad una proprietà.

```
set(target, property, value, receiver):
```

- `target` – rappresenta l'oggetto target, quello fornito come primo argomento a `new Proxy`,
- `property` – il nome della proprietà,
- `value` – il valore della proprietà,
- `receiver` – similmente alla trappola `get`, ha importanza solamente per le proprietà di tipo setter.

La trappola `set` dovrebbe ritornare `true` se è stata imposta correttamente, `false` altrimenti (innescando `TypeError`).

Utilizziamola per validare un nuovo valore:

```

let numbers = [];

numbers = new Proxy(numbers, { // (*)
 set(target, prop, val) { // per intercettare la scrittura di proprietà
 if (typeof val == 'number') {
 target[prop] = val;
 return true;
 } else {
 return false;
 }
 }
});

```

```

numbers.push(1); // aggiunta con successo
numbers.push(2); // aggiunta con successo
alert("Length is: " + numbers.length); // 2

numbers.push("test"); // TypeError ('set' di proxy ha ritornato false)

alert("This line is never reached (error in the line above)");

```

Da notare: le funzionalità interna degli array integrati continuano a funzionare! I valori vengono aggiunti tramite `push`. La proprietà `length` viene auto-incrementata quando i valori vengono aggiunti. Il nostro proxy non rompe nulla.

Non dobbiamo sovrascrivere i metodi di aggiunta valori agli array come `push`, `unshift` e così via per aggiungere i controlli, poiché questi metodi internamente utilizzano operazioni di `[ [Set] ]` che verranno intercettate dal proxy.

In questo modo il codice rimane pulito e conciso.

### Non dimenticate di ritornare `true`

Come detto sopra, vanno tenute in considerazione le invarianti.

Nel caso di `set`, questo deve ritornare `true` per scritture avvenute con successo.

Se ci dimentichiamo di farlo o ritorniamo qualsiasi altro valore, l'operazione innescherà `TypeError`.

## Iterazione con “`ownKeys`” e “`getOwnPropertyDescriptor`”

I cicli `Object.keys`, `for..in` e molti altri metodi che iterano sulle proprietà degli oggetti utilizzano il metodo interno `[ [OwnPropertyKeys] ]` (intercettate dalla trappola `ownKeys`) per ottenere la lista delle proprietà.

Questi metodi si distinguono per alcuni dettagli:

- `Object.getOwnPropertyNames(obj)` ritorna le chiavi non-symbol.
- `Object.getOwnPropertySymbols(obj)` ritorna le chiavi symbol.
- `Object.keys/values()` ritorna coppie keys/values non-symbol, con il flag `enumerable` (i flag sono stati spiegati nell'articolo [Attributi e descrittori di proprietà](#)).
- `for..in` itera su chiavi non-symbol, con il flag `enumerable`, ed anche sulle chiavi del prototype.

...Ma tutti questi incominciamo dalla stessa lista.

Nell'esempio sotto, utilizziamo la trappola `ownKeys` per far sì che `for..in` esegua il ciclo su `user`, `Object.keys` e `Object.values`, saltando le proprietà il cui nome incomincia con un underscore `_`:

```

let user = {
 name: "John",
 age: 30,
 _password: "****"

```

```

};

user = new Proxy(user, {
 ownKeys(target) {
 return Object.keys(target).filter(key => !key.startsWith('_'));
 }
});

// "ownKeys" filtra _password, saltandolo
for(let key in user) alert(key); // name, then: age

// abbiamo lo stesso effetto in questi metodi:
alert(Object.keys(user)); // name,age
alert(Object.values(user)); // John,30

```

Finora, funziona.

Anche se, nel caso in cui ritornassimo una chiave che non esiste nell'oggetto, `Object.keys` non la elencherà:

```

let user = { };

user = new Proxy(user, {
 ownKeys(target) {
 return ['a', 'b', 'c'];
 }
});

alert(Object.keys(user)); // <empty>

```

Perché? La motivazione è semplice: `Object.keys` ritorna solamente le proprietà con il flag `enumerable`. Per verificarlo, invoca il metodo interno `[[GetOwnProperty]]` su ogni proprietà per ottenere i suoi descrittori. In questo caso, poiché non ci sono proprietà, i descrittori sono vuoti, non abbiamo alcun flag `enumerable`, quindi questa verrà saltata.

Per far sì che `Object.keys` ritorni una proprietà, è necessario che questa esista nell'oggetto con il flag `enumerable`, oppure possiamo intercettare l'invocazione di `[[GetOwnProperty]]` (tramite la trappola `getOwnPropertyDescriptor`), e ritornare un descrittore con `enumerable: true`.

Qui vediamo un esempio:

```

let user = { };

user = new Proxy(user, {
 ownKeys(target) { // invocata una volta per ottenere una lista delle proprietà
 return ['a', 'b', 'c'];
 },
 getOwnPropertyDescriptor(target, prop) { // invocata per ogni proprietà
 return {
 enumerable: true,
 configurable: true
 /* ...altri flag, tra cui "value:..." */
 };
 }
});

```

```

 }
});

alert(Object.keys(user)); // a, b, c

```

Ripetiamolo una volta ancora: è sufficiente intercettare `[[GetOwnProperty]]` se la proprietà non è presente nell'oggetto.

## Proprietà protette da “`deleteProperty`” e altre trappole

Esiste una convenzione piuttosto diffusa, in cui le proprietà e i metodi il cui nome ha come suffisso un underscore `_`, sono da considerarsi interne. Non bisognerebbe quindi accedervi dall'esterno dell'oggetto.

Anche se rimane tecnicamente possibile accedervi:

```

let user = {
 name: "John",
 _password: "secret"
};

alert(user._password); // secret

```

Possiamo utilizzare un proxy per rendere inaccessibili le proprietà che iniziano con `_`.

Avremo bisogno delle seguenti trappole:

- `get` per ritornare un errore nel tentativo di accedere a questa proprietà,
- `set` per ritornare un errore nel tentativo di scrittura,
- `deleteProperty` per ritornare un errore nel tentativo di rimozione,
- `ownKeys` per escludere le proprietà che iniziano con `_` da `for..in` ed altri metodi come `Object.keys`.

Vediamo il codice:

```

let user = {
 name: "John",
 _password: "****"
};

user = new Proxy(user, {
 get(target, prop) {
 if (prop.startsWith('_')) {
 throw new Error("Access denied");
 }
 let value = target[prop];
 return (typeof value === 'function') ? value.bind(target) : value; // (*)
 },
 set(target, prop, val) { // per intercettare la scrittura delle proprietà
 if (prop.startsWith('_')) {
 throw new Error("Access denied");
 } else {

```

```

 target[prop] = val;
 return true;
 }
},
deleteProperty(target, prop) { // per intercettare la rimozione delle proprietà
 if (prop.startsWith('_')) {
 throw new Error("Access denied");
 } else {
 delete target[prop];
 return true;
 }
},
ownKeys(target) { // per intercettare lo scorrimento delle proprietà
 return Object.keys(target).filter(key => !key.startsWith('_'));
}
});

// "get" non consente di leggere _password
try {
 alert(user._password); // Errore: Access denied
} catch(e) { alert(e.message); }

// "set" non consente di scrivere _password
try {
 user._password = "test"; // Errore: Access denied
} catch(e) { alert(e.message); }

// "deleteProperty" non consente di rimuovere _password
try {
 delete user._password; // Errore: Access denied
} catch(e) { alert(e.message); }

// "ownKeys" rimuove _password dal ciclo
for(let key in user) alert(key); // name

```

Da notare un dettaglio importante nella trappola `get`, nella riga (\*) :

```

get(target, prop) {
 // ...
 let value = target[prop];
 return (typeof value === 'function') ? value.bind(target) : value; // (*)
}

```

Perché abbiamo bisogno di una funzione per invocare `value.bind(target)`?

La motivazione è che i metodi dell'oggetto, come `user.checkPassword()`, devono essere in grado di accedere a `_password`:

```

user = {
 // ...
 checkPassword(value) {
 // i metodi dell'oggetto devono essere in grado di leggere _password
 return value === this._password;
 }
}

```

Un'invocazione di `user.checkPassword()` passerà al proxy `user` come `this` (l'oggetto prima del punto diventa `this`), quindi quando proverà ad accedere a `this._password`, la trappola `get` si attiverà (viene innescata alla lettura di qualsiasi proprietà) e genererà un errore.

Quindi leghiamo il contesto dei metodi dell'oggetto all'oggetto originale, `target`, alla riga (\*). Le future invocazioni utilizzeranno `target` come `this`, senza alcuna trappola.

Questa soluzione solitamente funziona, ma non è ideale, poiché un metodo potrebbe passare l'oggetto senza proxy ovunque, e a quel punto faremmo un errore: dov'è l'oggetto originale, e dov'è quello con il proxy?

Oltretutto, un oggetto potrebbe essere racchiuso in più proxy (più proxy potrebbero aggiungere diverse funzionalità all'oggetto), e nel caso in cui passassimo un oggetto senza proxy ad un metodo, potremmo incorrere in conseguenze inaspettate.

Quindi, un proxy del genere non dovrebbe essere utilizzato ovunque.

### Proprietà private di una classe

I motori JavaScript moderni offrono un supporto nativo per le proprietà private nelle classi, aggiungendo il prefisso `#`. Questi sono descritti nell'articolo [Proprietà e metodi privati e protetti](#). Non è richiesto alcun proxy.

Anche se questo genere di proprietà hanno i loro problemi. In particolare, questi non vengono ereditati.

## “In range” con la trappola “has”

Vediamo altri esempi.

Abbiamo un oggetto `range`:

```
let range = {
 start: 1,
 end: 10
};
```

Vorremmo usare l'operatore `in` per verificare che un numero appartenga al `range`.

La trappola `has` intercetta le invocazioni di `in`.

`has(target, property)`

- `target` – è l'oggetto `target`, passato come primo argomento in `new Proxy`,
- `property` – nome della proprietà

Qui vediamo una dimostrazione:

```
let range = {
 start: 1,
 end: 10
};

range = new Proxy(range, {
```

```

has(target, prop) {
 return prop >= target.start && prop <= target.end;
}
});

alert(5 in range); // true
alert(50 in range); // false

```

Semplice zucchero sintattico, vero? Molto semplice da implementare.

## Wrapping con funzioni: "apply"

Possiamo costruire un proxy anche per funzioni.

La trappola `apply(target, thisArg, args)` gestisce l'invocazione di un proxy come funzione:

- `target` è l'oggetto target (le funzioni sono oggetti in JavaScript),
- `thisArg` è il valore di `this`.
- `args` è la lista degli argomenti.

Ad esempio, il decoratore `delay(f, ms)`, che abbiamo sviluppato nell'articolo [\\*Decorators\\*](#) e [forwarding, call/apply](#).

In quell'articolo lo abbiamo fatto senza proxy. Un'invocazione di `delay(f, ms)` ritornava una funzione che inoltra le chiamate di `f` dopo `ms` millisecondi.

Qui vediamo la precedente implementazione, basata sulla funzione:

```

function delay(f, ms) {
 // ritorna un wrapper che invoca f dopo il timeout
 return function() { // (*)
 setTimeout(() => f.apply(this, arguments), ms);
 };
}

function sayHi(user) {
 alert(`Hello, ${user}!`);
}

// dopo il wrapping, le invocazioni di sayHi verranno ritardate di 3 secondi
sayHi = delay(sayHi, 3000);

sayHi("John"); // Hello, John! (dopo 3 secondi)

```

Come abbiamo già visto, questo approccio funziona. La funzione wrapper `(*)` esegue l'invocazione dopo il timeout.

Ma una funzione wrapper non esegue l'inoltro delle operazioni di lettura/scrittura o altro di simile. Dopo il wrapping, l'accesso alle proprietà della funzione originale è perso, come `name`, `length` e altri:

```

function delay(f, ms) {
 return function() {

```

```

 setTimeout(() => f.apply(this, arguments), ms);
 };
}

function sayHi(user) {
 alert(`Hello, ${user}!`);
}

alert(sayHi.length); // 1 (la lunghezza della funzione è il numero degli argomenti nella sua dichiarazione)

sayHi = delay(sayHi, 3000);

alert(sayHi.length); // 0 (nella dichiarazione del wrapper, ci sono zero argomenti)

```

Il `proxy` è molto più potente, poiché inoltra tutto all'oggetto target.

Utilizziamo il `Proxy` piuttosto della funzione di wrapping:

```

function delay(f, ms) {
 return new Proxy(f, {
 apply(target, thisArg, args) {
 setTimeout(() => target.apply(thisArg, args), ms);
 }
 });
}

function sayHi(user) {
 alert(`Hello, ${user}!`);
}

sayHi = delay(sayHi, 3000);

alert(sayHi.length); // 1 (*) il proxy inoltra l'operazione "get length" all'oggetto target

sayHi("John"); // Hello, John! (after 3 seconds)

```

Il risultato è lo stesso, ma ora non viene inoltrata solamente l'invocazione, anche tutte le altre operazioni sul proxy vengono inoltrate alla funzione originale. Quindi `sayHi.length` viene ritornato correttamente dopo il wrapping alla riga (\*).

Abbiamo ottenuto un wrapper più “ricco”.

Esistono altre trappole: la lista completa la puoi trovare all'inizio di questo articolo. Il loro utilizzo è molto simile a quanto spiegato sopra.

## Reflect

`Reflect` è un oggetto integrato che semplifica la creazione di `Proxy`.

Come detto in precedenza, i metodi interni, come `[[Get]]`, `[[Set]]` e altri, esistono solamente nelle specifiche, non possono essere invocati direttamente.

L'oggetto `Reflect` lo rende in qualche modo possibile. I suoi metodi sono wrapper dei metodi interni.

Qui vediamo degli esempi di operazioni e invocazioni di `Reflect` che fanno questo:

Operazione	invocazione Reflect	Metodo interno
obj[prop]	Reflect.get(obj, prop)	[[Get]]
obj[prop] = value	Reflect.set(obj, prop, value)	[[Set]]
delete obj[prop]	Reflect.deleteProperty(obj, prop)	[[Delete]]
new F(value)	Reflect.construct(F, value)	[[Construct]]
...	...	...

Ad esempio:

```
let user = {};
Reflect.set(user, 'name', 'John');
alert(user.name); // John
```

In particolare, `Reflect` ci consente di invocare operatori (`new`, `delete`...) come funzioni (`Reflect.construct`, `Reflect.deleteProperty`, ...). Questa è una caratteristica interessante, ma qui vediamo un'altra cosa molto importante.

**Per ogni metodo interno a cui possiamo aggiungere una trappola con il `Proxy`, abbiamo un metodo corrispondente in `Reflect`, con lo stesso nome e gli stessi argomenti della trappola `Proxy`.**

Quindi possiamo utilizzare `Reflect` per inoltrare un'operazione all'oggetto originale.

In questo esempio, entrambe le trappole `get` e `set` inoltrano in maniera trasparente (come se non esistessero) le operazioni di lettura/scrittura all'oggetto, mostrando un messaggio:

```
let user = {
 name: "John",
};

user = new Proxy(user, {
 get(target, prop, receiver) {
 alert(`GET ${prop}`);
 return Reflect.get(target, prop, receiver); // (1)
 },
 set(target, prop, val, receiver) {
 alert(`SET ${prop}=${val}`);
 return Reflect.set(target, prop, val, receiver); // (2)
 }
});

let name = user.name; // mostra "GET name"
user.name = "Pete"; // mostra "SET name=Pete"
```

Qui:

- `Reflect.get` legge una proprietà di un oggetto.
- `Reflect.set` scrive una proprietà di un oggetto e ritorna `true` se questa ha successo, `false` altrimenti.

Questo è tutto, piuttosto semplice: se una trappola vuole inoltrare l'invocazione all'oggetto, è sufficiente invocare `Reflect.<method>` con gli stessi argomenti.

In molti casi possiamo ottenere lo stesso risultato senza `Reflect`, ad esempio la lettura di una proprietà `Reflect.get(target, prop, receiver)` può essere sostituita da `target[prop]`. Ci sono però delle sfumature importanti.

## Creare un proxy per un getter

Vediamo un esempio che dimostra perché `Reflect.get` è migliore. E vedremo anche perché `get/set` possiede il terzo argomento `receiver`, che non abbiamo utilizzato finora.

Abbiamo un oggetto `user` con la proprietà `_name` ed il relativo getter.

Costruiamo un proxy:

```
let user = {
 _name: "Guest",
 get name() {
 return this._name;
 }
};

let userProxy = new Proxy(user, {
 get(target, prop, receiver) {
 return target[prop];
 }
});

alert(userProxy.name); // Guest
```

La trappola `get` è “trasparente” in questo caso, ritorna la proprietà originale e non fa nient’altro. Questo è sufficiente per il nostro esempio.

Tutto sembra funzionare correttamente. Ma rendiamo l’esempio leggermente più complesso.

Dopo aver ereditato con un oggetto `admin` da `user`, possiamo osservare un comportamento non corretto:

```
let user = {
 _name: "Guest",
 get name() {
 return this._name;
 }
};

let userProxy = new Proxy(user, {
 get(target, prop, receiver) {
 return target[prop]; // (*) target = user
 }
});
```

```

let admin = {
 __proto__: userProxy,
 _name: "Admin"
};

// Risultato atteso: Admin
alert(admin.name); // outputs: Guest (?!?)

```

La lettura di `admin.name` dovrebbe ritornare "Admin", non "Guest"!

Qual'è il problema? Magari abbiamo sbagliato qualcosa con l'ereditarietà?

Ma se rimuoviamo il proxy, tutto funziona correttamente.

Il problema sta quindi nel proxy, alla riga (\*) .

1. Quando leggiamo `admin.name`, poiché l'oggetto `admin` non possiede questa proprietà, la ricerca prosegue nel suo prototype.
2. Il prototype è `userProxy`.
3. Durante la lettura della proprietà `name` dal proxy, la trappola `get` viene innescata e ritorna la proprietà dell'oggetto originale `target[prop]` alla riga (\*) .

Un'invocazione di `target[prop]`, nel caso in cui `prop` sia un getter, ne esegue il codice con contesto `this=target`. Quindi il risultato sarà `this._name` dell'oggetto `target`, quindi: da `user`.

Per evitare questo, abbiamo bisogno di `receiver`, il terzo argomento della trappola `get`. Questo fa riferimento al `this` corretto, quello che deve essere passato al getter. Nel nostro caso `admin`.

Come possiamo passare il contesto per un getter? Per una funzione regolare potremmo usare `call/apply`, ma questo è un getter, non viene "invocato", ma vi si accede semplicemente.

`Reflect.get` fa al caso nostro. Tutto funzionerà correttamente se ne facciamo uso.

Vediamo la variante corretta:

```

let user = {
 _name: "Guest",
 get name() {
 return this._name;
 }
};

let userProxy = new Proxy(user, {
 get(target, prop, receiver) { // receiver = admin
 return Reflect.get(target, prop, receiver); // (*)
 }
});

let admin = {
 __proto__: userProxy,
 _name: "Admin"
};

```

```
alert(admin.name); // Admin
```

Ora `receiver` fa riferimento al `this` corretto (cioè `admin`), e verrà passato al getter utilizzando `Reflect.get`, come in riga (\*).

Possiamo riscrivere la trappola in maniera ancora più breve:

```
get(target, prop, receiver) {
 return Reflect.get(...arguments);
}
```

Le funzioni `reflect` hanno lo stesso nome delle trappole ed accettano gli stessi argomenti. Sono stati progettati in questo modo.

Quindi, `return Reflect...` è un modo sicuro e semplice per inoltrare le operazioni ed essere sicuri di non dimenticarci nulla.

## Limitazioni del proxy

I proxy forniscono un modo unico per alterare o aggirare il comportamento a basso livello degli oggetti esistenti. Non è comunque perfetto. Ha delle limitazioni.

### Oggetti integrati: slot interni

Molti oggetti integrati, ad esempio `Map`, `Set`, `Date`, `Promise` e altri, fanno uso dei così detti “internal slots”.

Questi sono come le proprietà, ma sono riservati ad usi interni, fanno parte solamente delle specifiche. Ad esempio, `Map` memorizza gli elementi nello slot interno `[[MapData]]`. I metodi integrati accedono direttamente a questi, non utilizzano i metodi `[[Get]]/[[Set]]`. Quindi `Proxy` non potrà intercettarli.

Perché questo ha importanza? Sono comunque entità interne!

Non proprio, vediamo qual'è il problema. Dopo aver creato un proxy per un oggetto integrato, il proxy non avrà questi slot interni, quindi i metodi integrati falliranno.

Ad esempio:

```
let map = new Map();

let proxy = new Proxy(map, {});

proxy.set('test', 1); // Errore
```

Internamente, una `Map` memorizza i suoi dati nello slot `[[MapData]]`. Il proxy non possiede questo slot. Il metodo integrato `Map.prototype.set` prova ad accedere alla proprietà interna `this.[[MapData]]`, ma poiché `this=proxy`, non la trova nel `proxy` e fallisce.

Fortunatamente, esiste un modo per evitare questo:

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

```

let proxy = new Proxy(map, {
 get(target, prop, receiver) {
 let value = Reflect.get(...arguments);
 return typeof value == 'function' ? value.bind(target) : value;
 }
});

proxy.set('test', 1);
alert(proxy.get('test')) // 1 (funziona!)

```

Ora funziona senza problemi, poiché la trappola `get` si lega alle proprietà della funzione, come `map.set`, per ottenere l'oggetto target (`map`) stesso.

A differenza dell'esempio precedente, il valore di `this` all'interno di `proxy.set(...)` non sarà `proxy`, ma piuttosto l'oggetto originale `map`. Quindi quando l'implementazione interna di `set` proverà ad accedere allo slot interno `this.[[MapData]]`, l'operazione avverrà con successo.

### i Array non possiede slot interni

Un'eccezione degna di nota: l'oggetto integrato `Array` non utilizza slot interni. Questo per ragioni storiche, poiché esistono da molto tempo.

Quindi non avremo nessun problema nel creare proxy per un array.

## Campi privati

Un comportamento simile avviene con i campi privati di una classe.

Ad esempio, il metodo `getName()` accede alla proprietà privata `#name` e comporta il fallimento del proxy:

```

class User {
 #name = "Guest";

 getName() {
 return this.#name;
 }
}

let user = new User();

user = new Proxy(user, {});

alert(user.getName()); // Errore

```

La motivazione è che i campi privati sono implementati utilizzando gli slot interni. JavaScript non utilizza `[Get]`/`[Set]` per accedervi.

Nell'invocazione `getName()` il valore di `this` è il proxy di `user`, e questo non possiede lo slot interno con i campi privati.

Nuovamente, la soluzione di legare il metodo è corretta anche in questo caso:

```

class User {
 #name = "Guest";

 getName() {
 return this.#name;
 }
}

let user = new User();

user = new Proxy(user, {
 get(target, prop, receiver) {
 let value = Reflect.get(...arguments);
 return typeof value == 'function' ? value.bind(target) : value;
 }
});

alert(user.getName()); // Guest

```

Detto questo, la soluzione avrà degli svantaggi, come spiegato in precedenza: espone l'oggetto originale al metodo, consentendo, potenzialmente, che questo venga passato ulteriormente rompendo la funzionalità avvolta nel proxy.

## Proxy != target

Il proxy e l'oggetto originale sono due oggetti differenti. Normale, giusto?

Quindi se utilizziamo l'oggetto originale come chiave, e successivamente ne creiamo un proxy, allora il proxy non sarà accessibile:

```

let allUsers = new Set();

class User {
 constructor(name) {
 this.name = name;
 allUsers.add(this);
 }
}

let user = new User("John");

alert(allUsers.has(user)); // true

user = new Proxy(user, {});

alert(allUsers.has(user)); // false

```

Come possiamo vedere, dopo aver aggiunto il proxy, non riusciamo ad accedere a `user` con il setter `allUsers`, poiché il proxy è un oggetto differente.

### I proxy non possono intercettare un test di uguaglianza stretta ===

I proxy possono intercettare molti operatori, come `new` (con `construct`), `in` (con `has`), `delete` (con `deleteProperty`) e così via.

Ma non esiste alcun modo per poter intercettare un test di uguaglianza stretta tra oggetti. Un oggetto è strettamente uguale solamente a se stesso, e a nient'altro.

Quindi tutte le operazioni e le classi integrate che verificano l'uguaglianza tra oggetti differenzieranno l'oggetto dal suo proxy. Non c'è alcun sistema di sostituzione "trasparente" in questo caso.

## Proxy revocabili

Un proxy *revocabile* è un proxy che può essere disabilitato.

Ipotizziamo di avere una risorsa, di cui vorremmo poter bloccare gli accessi in qualsiasi momento.

Quello che possiamo fare è creare un proxy *revocabile*, senza alcuna trappola. Un proxy di questo tipo inoltrerà tutte le operazioni all'oggetto originale, e possiamo disabilitarlo in ogni momento.

La sintassi da utilizzare è la seguente:

```
let {proxy, revoke} = Proxy.revocable(target, handler)
```

L'invocazione ritorna un oggetto con le funzioni `proxy` e `revoke` per disabilitarlo.

Vediamo un esempio:

```
let object = {
 data: "Valuable data"
};

let {proxy, revoke} = Proxy.revocable(object, {});

// passiamo il proxy da qualche parte, piuttosto dell'oggetto...
alert(proxy.data); // Dati preziosi

// più tardi nel nostro codice
revoke();

// il proxy non funzionerà più (revocato)
alert(proxy.data); // Errore
```

L'invocazione di `revoke()` rimuove dal proxy tutti i referimenti interni all'oggetto, quindi questi non risulteranno essere più connessi.

Inizialmente, `revoke` è separato da `proxy`, in questo modo possiamo passare il `proxy` in giro, mantenendo il `revoke` nello scope attuale.

Possiamo anche legare il metodo `revoke` al proxy, impostando `proxy.revoke = revoke`.

Un'altra opzione è quella di creare una `WeakMap` che possiede il `proxy` come chiave e il corrispondente `revoke` come valore; questo consente di trovare facilmente il `revoke` per un `proxy`:

```
let revokes = new WeakMap();

let object = {
 data: "Valuable data"
};

let {proxy, revoke} = Proxy.revocable(object, {});

revokes.set(proxy, revoke);

// ..da qualche altra parte nel nostro codice..
revoke = revokes.get(proxy);
revoke();

alert(proxy.data); // Errore (revocato)
```

In questo caso utilizziamo una `WeakMap` piuttosto di `Map` di modo che non blocchi il processo di garbage collection. Se un proxy diventa “irraggiungibile” (e.g. nessuna variabile fa riferimento ad esso), `WeakMap` consente di rimuoverlo dalla memoria insieme al relativo `revoke` che non sarà più necessario.

## Riferimenti

- Specifiche: [Proxy ↗](#).
- MDN: [Proxy ↗](#).

## Riepilogo

Il `Proxy` è un contenitore per un oggetto, che inoltra tutte le operazioni su di esso all’oggetto originale, e consente di definire delle “trappole” per determinate operazioni.

E’ possibile creare un proxy per qualsiasi tipo di oggetto, incluse le classi e le funzioni.

La sintassi da utilizzare è la seguente:

```
let proxy = new Proxy(target, {
 /* trappole */
});
```

...Successivamente, dovremmo utilizzare il `proxy` ovunque, ed evitare l’utilizzo di `target`. Un proxy non possiede proprietà o metodi propri. Si occupa di intercettare le operazioni (se sono definite le relative trappole), altrimenti le inoltra all’oggetto `target`.

Possiamo intercettare:

- Lettura (`get`), scrittura (`set`), rimozione (`deleteProperty`) di una proprietà (anche di quelle non esistenti).
- Invocazione di funzione (trappola `apply`).

- Operatore `new` (trappola `construct`).
- Molte altre operazioni (puoi trovare la lista completa a inizio articolo e nella [documentazione ↗](#)).

Questo ci consente di creare proprietà e metodi “virtuali”, implementare valori di default, oggetti observables, decorators e molto altro.

Possiamo anche costruire proxy multipli di un oggetto, decorandolo con diverse funzionalità.

L'API [Reflect ↗](#) è stata progettata per completare l'utilizzo dei [Proxy ↗](#). Per ogni trappola `Proxy`, esiste un'invocazione di `Reflect` con gli stessi argomenti. Possiamo utilizzarlo per inoltrare le invocazioni agli oggetti target.

I proxy hanno però delle limitazioni:

- Gli oggetti integrati possiedono degli “slot interni”, ma l'accesso a questi non può essere intercettato dai proxy. Guardate il workaround descritto sopra.
- Lo stesso vale per i campi privati delle classi; questi vengono implementati internamente utilizzando gli slot. Quindi le invocazioni dei metodi tramite proxy devono possedere il target object assegnato a `this` per potervi accedere.
- I test di uguaglianza `==` sugli oggetti, non possono essere intercettati.
- Performance: i benchmark dipendono molto dal motore JavaScript, ma generalmente l'accesso alle proprietà utilizzando un proxy, richiede più tempo. Anche se, nella pratica, questo ha importanza solo per oggetti che creano “colli di bottiglia”.

## ✓ Esercizi

### Errore in lettura di una proprietà non esistente

Solitamente, un tentativo di accesso ad una proprietà non esistente ritorna `undefined`.

Create un proxy che generi un errore ad ogni tentativo di accesso ad una proprietà non esistente.

Questo può aiutare a trovare errori di programmazione in anticipo.

Scrivete una funzione `wrap(target)` che prende un oggetto `target` e ne ritorna un proxy con la funzionalità appena descritta.

Ecco come dovrebbe funzionare:

```
let user = {
 name: "John"
};

function wrap(target) {
 return new Proxy(target, {
 /* il vostro codice */
 });
}

user = wrap(user);
```

```
alert(user.name); // John
alert(user.age); // ReferenceError: Property doesn't exist: "age"
```

Alla soluzione

---

## Accesso ad un array[-1]

In alcuni linguaggi di programmazione, possiamo accedere agli elementi dell'array utilizzando indici negativi, che iniziano il conteggio dalla coda dell'array.

Come nell'esempio:

```
let array = [1, 2, 3];

array[-1]; // 3, l'ultimo elemento
array[-2]; // 2, il penultimo elemento
array[-3]; // 1, il terzultimo elemento
```

In altre parole, `array[-N]` equivale a `array[array.length - N]`.

Create un proxy che implementa questa funzionalità.

Ecco come dovrebbe funzionare:

```
let array = [1, 2, 3];

array = new Proxy(array, {
 /* il vostro codice */
});

alert(array[-1]); // 3
alert(array[-2]); // 2

// Le altre funzionalità dell'array devono rimanere inalterate
```

Alla soluzione

---

## Observable

Create una funzione `makeObservable(target)` che “rende l'oggetto osservabile” ritornandone un proxy.

Ecco come dovrebbe funzionare:

```
function makeObservable(target) {
 /* il vostro codice */
}

let user = {};
user = makeObservable(user);

user.observe((key, value) => {
```

```
 alert(`SET ${key}=${value}`);
 });

user.name = "John"; // alerts: SET name=John
```

In altre parole, un oggetto ritornato da `makeObservable` equivale a quello originale, ma possiede il metodo `observe(handler)` che imposta la funzione `handler` per essere invocata quando una qualsiasi proprietà cambia.

Quando una proprietà verrà modificata, `handler(key, value)` verrà invocato con il nome ed il valore della proprietà.

P.S. In questo task, gestite solamente la scrittura della proprietà. Le altre operazioni possono essere implementate in maniera simile.

[Alla soluzione](#)

## Eval: eseguire una stringa di codice

La funzione integrata `eval` consente di eseguire stringhe di codice.

La sintassi da utilizzare è:

```
let result = eval(code);
```

Ad esempio:

```
let code = 'alert("Hello")';
eval(code); // Hello
```

Una stringa di codice può essere molto lunga, contenere interruzioni di riga, dichiarazioni di funzione, variabili e così via.

Il risultato ritornato da `eval` corrisponde a quello dell'ultima istruzione.

Ad esempio:

```
let value = eval('1+1');
alert(value); // 2
```

```
let value = eval('let i = 0; ++i');
alert(value); // 1
```

Il codice valutato viene eseguito nel *lexical environment* corrente, quindi può accedere alle variabili esterne:

```
let a = 1;
```

```
function f() {
 let a = 2;

 eval('alert(a)'); // 2
}

f();
```

Allo stesso modo, può modificare le variabili esterne:

```
let x = 5;
eval("x = 10");
alert(x); // 10, valore modificato
```

In strict mode, `eval` viene eseguito in un *lexical environment* separato. Quindi le funzioni e le variabili dichiarate internamente, non sono visibili all'esterno:

```
// attenzione: 'use strict' è abilitato di default negli esempi che stiamo eseguendo

eval("let x = 5; function f() {}");

alert(typeof x); // undefined (variabile inesistente)
// anche la funzione f non è visibile
```

Senza `use strict`, `eval` non viene eseguito in un *lexical environment* separato, quindi saremo in grado di vedere `x` e `f` dall'esterno.

## Utilizzare “eval”

Nella programmazione moderna `eval` viene utilizzato di rado. Spesso si dice “eval is evil” (“eval è il diavolo”).

La motivazione è piuttosto semplice: molto tempo fa, JavaScript era un linguaggio molto povero, molte cose potevano essere fatte solamente tramite `eval`. Ma quei tempi, ormai sono passati da un decennio.

Al momento, non esiste alcuna ragione per utilizzare `eval`. Se qualcuno lo sta utilizzando, c'è una buona possibilità che questo sia rimpiazzabile con un costrutto del linguaggio oppure con un [modulo JavaScript](#).

Fate attenzione: la sua capacità di accedere alle variabili esterne può generare side-effects.

I *code minifiers* (strumenti utilizzati per comprimere gli script JS prima di metterli in produzione) rinominano le variabili locali con nomi più brevi (come `a`, `b` etc) in modo da rendere il codice più breve. Questa operazione, solitamente, è sicura. Non lo è, però, se stiamo utilizzando `eval`, poiché al suo interno potremmo provare ad accedere alle variabili locali. Quindi i minifiers non rinominano tutte le variabili che sono potenzialmente accessibili da `eval`. Questo comporta un peggioramento del fattore di compressione del codice.

L'utilizzo delle variabili locali all'interno di `eval` è considerata una bad practice nella programmazione, poiché rende il codice molto più complesso da mantenere.

Esistono due modi per mettersi al sicuro da questi problemi.

**Se il codice all'interno di eval non richiede variabili esterne, allora invocate eval come window.eval(...):**

In questo modo il codice viene eseguito nello scope globale:

```
let x = 1;
{
 let x = 5;
 window.eval('alert(x)'); // 1 (variabile globale)
}
```

**Se il codice all'interno di eval ha bisogno di variabili locali, sostituite eval con new Function e passategliele come argomenti:**

```
let f = new Function('a', 'alert(a)');
f(5); // 5
```

Il costrutto `new Function`, che viene spiegato più nel dettaglio nel capitolo [La sintassi "new Function"](#), crea una nuova funzione a partire da una stringa. Questa viene creata nel contesto globale, quindi non può accedere alle variabili locali, ma è comunque possibile passargliele esplicitamente come argomenti, come nell'esempio visto sopra.

## Riepilogo

L'invocazione di `eval(code)` esegue una stringa di codice e ne ritorna il risultato dell'ultima istruzione.

- Viene raramente utilizzato in JavaScript moderno, quindi in genere non ne avremo bisogno.
- Può accedere alle variabili locali. Questa è considerata una bad practice.
- Piuttosto di utilizzare `eval` nello scope globale, utilizzate `window.eval(code)`.
- Oppure, se il codice dovesse avere bisogno di dati dallo scope esterno, utilizzate il costrutto `new Function` e passateglieli come argomenti.

## ✓ Esercizi

### Eval-calculator

importanza: 4

Create una calcolatrice che richieda all'utente (tramite `prompt`) un'espressione aritmetica e ne ritorni il risultato.

Non c'è alcun bisogno di testare la correttezza dell'espressione. Limitatevi ad eseguirla e ritornarne il risultato.

[Esegui la demo](#)

[Alla soluzione](#)

## Currying

Il [currying](#) è una tecnica avanzata che si applica alle funzioni. Non viene utilizzata solamente in JavaScript, ma anche in altri linguaggi di programmazione.

Il currying è una trasformazione che traduce una funzione invocabile come `f(a, b, c)` in una invocabile come `f(a)(b)(c)`.

Il currying non invoca la funzione. Si occupa solamente della sua trasformazione.

Come prima cosa vediamo un esempio, in modo da capire di cosa stiamo parlando, e le applicazioni nella pratica.

Creeremo una funzione di supporto `curry(f)` che esegue il currying per una funzione a due argomenti `f`. In altre parole, `curry(f)`, trasformerà `f(a, b)` in una funzione invocabile come `f(a)(b)`:

```
function curry(f) { // curry(f) esegue il currying
 return function(a) {
 return function(b) {
 return f(a, b);
 };
 };
}

// utilizzo
function sum(a, b) {
 return a + b;
}

let curriedSum = curry(sum);

alert(curriedSum(1)(2)); // 3
```

Come potete vedere, l'implementazione è piuttosto semplice: sono due semplici wrappers.

- Il risultato di `curry(func)` è un wrapper `function(a)`.
- Quando viene invocato come `curriedSum(1)`, l'argomento viene memorizzato nel Lexical Environment, e viene ritornato un nuovo wrapper `function(b)`.
- Successivamente questo wrapper viene invocato con `2` come argomento, che passerà l'invocazione a `sum`.

Implementazioni più avanzate del currying, come [\\_.curry](#) fornito dalla libreria lodash, ritorna un wrapper che consente di invocare una funzione sia nella forma standard che in quella parziale:

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

```
let curriedSum = _.curry(sum); // utilizzando _.curry della libreria lodash

alert(curriedSum(1, 2)); // 3, riamane invocabile normalmente
alert(curriedSum(1)(2)); // 3, invocata parzialmente
```

## Currying? Per quale motivo?

Per poterne comprendere i benefici abbiamo bisogno di un esempio di applicazione reale.

Ad esempio, abbiamo una funzione di logging `log(date, importance, message)` che formatta e ritorna le informazioni. In un progetto reale, una funzione del genere ha diverse funzionalità utili, come l'invio di log in rete; qui useremo semplicemente un `alert`:

```
function log(date, importance, message) {
 alert(`[${date.getHours()}]:${date.getMinutes()}] [${importance}] ${message}`);
}
```

Eseguiamo il currying!

```
log = _.curry(log);
```

Successivamente al `log`, funzionerà normalmente:

```
log(new Date(), "DEBUG", "some debug"); // log(a, b, c)
```

...Ma funzionerà anche nella forma parziale:

```
log(new Date())("DEBUG")("some debug"); // log(a)(b)(c)
```

Ora possiamo creare una funzione utile per registrare i logs:

```
// logNow sarà la versione parziale di log con il primo argomento fisso
let logNow = log(new Date());

// utilizziamola
logNow("INFO", "message"); // [HH:mm] INFO message
```

Ora `logNow` equivale a `log` con il primo argomento fissato, in altre parole, una “funzione applicata parzialmente” o “parziale” (più breve).

Possiamo anche andare oltre, e creare una funzione utile per registrare i logs di debug:

```
let debugNow = logNow("DEBUG");

debugNow("message"); // [HH:mm] DEBUG message
```

Quindi:

1. Non perdiamo nulla dopo il currying: `log` rimane invocabile normalmente.
2. Possiamo generare molto semplicemente funzioni parziali per i logs quotidiani.

## Implementazione avanzata del currying

Nel caso in cui vogliate entrare più nel dettaglio, di seguito vediamo un'implementazione "avanzata" del currying per funzioni con più argomenti, che avremmo anche potuto usare sopra.

E' piuttosto breve:

```
function curry(func) {

 return function curried(...args) {
 if (args.length >= func.length) {
 return func.apply(this, args);
 } else {
 return function(...args2) {
 return curried.apply(this, args.concat(args2));
 }
 }
 };
}
```

Esempi di utilizzo:

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

let curriedSum = curry(sum);

alert(curriedSum(1, 2, 3)); // 6, ancora invocabile normalmente
alert(curriedSum(1)(2,3)); // 6, currying del primo argomento
alert(curriedSum(1)(2)(3)); // 6, currying completo
```

La funzione `curry` può sembrare complicata, ma in realtà è piuttosto semplice da capire.

Il risultato dell'invocazione `curry(func)` è il wrapper `curried` (che ha subito il processo di currying), ed appare in questo modo:

```
// func è la funzione trasformata
function curried(...args) {
 if (args.length >= func.length) { // (1)
 return func.apply(this, args);
 } else {
 return function(...args2) { // (2)
 return curried.apply(this, args.concat(args2));
 }
 }
};
```

Quando la eseguiamo, ci sono due percorsi di esecuzione `if`:

1. Se il numero di `args` forniti è uguale o maggiore rispetto a quelli che la funzione originale ha nella sua definizione (`func.length`), allora gli giriamo semplicemente l'invocazione utilizzando `func.apply`.
2. Altrimenti, otterremo un parziale: non invochiamo ancora `func`. Invece, viene ritornato un altro wrapper, che riapplicherà il `curried` passando gli argomenti precedenti insieme a quelli nuovi.

Quindi, se la invochiamo, di nuovo, avremo o una nuova funzione parziale (se non vengono forniti abbastanza argomenti) oppure otterremo il risultato.

#### Solo funzioni di lunghezza fissa

Il currying richiede che la funzione abbia un numero fissato di argomenti.

Una funzione che utilizza i parametri rest, come `f(...args)`, non può passare attraverso il processo di currying in questo modo.

#### Un po' più del currying

Per definizione, il currying dovrebbe convertire `sum(a, b, c)` in `sum(a)(b)(c)`.

Ma la maggior parte delle implementazioni in JavaScript sono più avanzate di così, come descritto: queste mantengono la funzione invocabile nella variante a più argomenti.

## Riepilogo

Il *currying* è una trasformazione che rende `f(a, b, c)` invocabile come `f(a)(b)(c)`. Le implementazioni in JavaScript, solitamente, mantengono entrambe le varianti, sia quella normale che quella parziale, se il numero di argomenti non è sufficiente.

Il currying permette di ottenere delle funzioni parziali molto semplicemente. Come abbiamo visto nell'esempio del logging, dopo il currying la funzione universale a tre argomenti `log(date, importance, message)` ci fornisce una funzione parziale quando invocata con un solo argomento (come `log(date)`) o due argomenti (come `log(date, importance)`).

## Il tipo Reference

#### Caratteristica avanzata di linguaggio

Questo articolo tratta un argomento avanzato, utile per capire meglio alcuni casi limite.

Non è di fondamentale importanza. Molti sviluppatori esperti vivono bene senza esserne a conoscenza. Continua la lettura se sei interessato a sapere come funzionano le cose internamente.

Un'invocazione di un metodo valutata dinamicamente può perdere il `this`.

Ad esempio:

```

let user = {
 name: "John",
 hi() { alert(this.name); },
 bye() { alert("Bye"); }
};

user.hi(); // funziona

// ora invochiamo user.hi o user.bye in base al nome
(user.name == "John" ? user.hi : user.bye)(); // Errore!

```

Nell'ultima riga abbiamo un operatore condizionale che sceglie tra `user.hi` o `user.bye`. In questo caso il risultato è `user.hi`.

Successivamente il metodo viene invocato immediatamente con le parentesi `( )`. Ma non funziona correttamente!

Come potete vedere, l'invocazione genera un errore, perché il valore di `"this"` all'interno della chiamata diventa `undefined`.

Questo invece funziona (object punto metodo):

```
user.hi();
```

Questo no (valutazione del metodo):

```
(user.name == "John" ? user.hi : user.bye)(); // Errore!
```

Perché? Se vogliamo capire il motivo, dobbiamo addentrarci nei dettagli del funzionamento della chiamata `obj.method()`.

## Il tipo Reference spiegato

Guardando da più vicino, possiamo notare due operazioni nell'istruzione `obj.method()`:

1. Primo, il punto `'.'` recupera la proprietà `obj.method`.
2. Successivamente le parentesi `( )` la eseguono.

Quindi, come vengono passate le informazioni riguardo al `this` dalla prima alla seconda parte?

Se spostiamo queste istruzioni in righe separate, allora `this` verrà sicuramente perso:

```

let user = {
 name: "John",
 hi() { alert(this.name); }
}

// dividiamo l'accesso e l'invocazione in due righe
let hi = user.hi;
hi(); // Errore, perché this è undefined

```

Qui `hi = user.hi` assegna la funzione alla variabile, e nell'ultima riga è completamente autonoma, quindi non si ha alcun `this`.

Per rendere l'invocazione `user.hi()` funzionante, JavaScript applica un trucco – il punto `'.'` non ritorna una funzione, ma piuttosto un valore del tipo speciale [Reference ↗](#).

Il tipo Reference è un “tipo descritto dalla specifica”. Non possiamo utilizzarlo esplicitamente, ma viene utilizzato internamente dal linguaggio.

Il valore del tipo Reference è una combinazione di tre valori `(base, name, strict)`, dove:

- `base` è l'oggetto.
- `name` è il nome della proprietà.
- `strict` vale true se `use strict` è attivo.

Il risultato dell'accesso alla proprietà `user.hi` non è una funzione, ma un valore di tipo Reference. Per `user.hi` in strict mode vale:

```
// valore di tipo Reference
(user, "hi", true)
```

Quando le parentesi `()` vengono invocate in un tipo Reference, queste ricevono tutte le informazioni riguardo l'oggetto ed il metodo, e possono quindi impostare correttamente il valore di `this` (`=user` in questo caso).

Il tipo Reference è uno speciale tipo “intermedio” utilizzato internamente, con lo scopo di passare le informazioni dal punto `.` all'invocazione con le parentesi `()`.

Qualsiasi altra operazione come un assegnazione `hi = user.hi` scarta completamente il tipo Reference, accede al valore `user.hi` (una funzione) e lo ritorna. Quindi qualsiasi ulteriore operazione “perderà” `this`.

Quindi, come risultato, il valore di `this` viene passato correttamente solo se la funzione viene invocata direttamente utilizzando il punto `obj.method()` o la sintassi con le parentesi quadre `obj['method']()` (in questo caso si equivalgono). Esistono diversi modi per evitare questo problema, come [func.bind\(\)](#).

## Riepilogo

Il tipo Reference è un tipo interno del linguaggio.

La lettura di una proprietà, con il punto `.` in `obj.method()` non ritorna esattamente il valore della proprietà, ma uno speciale “tipo reference” che memorizza sia il valore della proprietà che l'oggetto a cui accedere.

Questo accade per consentire che la successiva invocazione con `()` imposti correttamente il `this`.

Per tutte le altre operazioni, il tipo reference diventa automaticamente il valore della proprietà (una funzione nel nostro caso).

Il meccanismo descritto è nascosto ai nostri occhi. Ha importanza solo in alcuni casi, ad esempio quando un metodo viene ottenuto dinamicamente dall'oggetto, utilizzando un'espressione.

## ✓ Esercizi

### Controllo di sintassi

importanza: 2

Qual'è il risultato di questo codice?

```
let user = {
 name: "John",
 go: function() { alert(this.name) }
}

(user.go)()
```

P.S. C'è una trappola :)

[Alla soluzione](#)

### Spiegate il valore di "this"

importanza: 3

Nel codice sotto vogliamo chiamare il metodo `user.go()` volte di fila.

Ma le chiamate (1) e (2) funzionano diversamente da (3) e (4). Perché?

```
let obj, method;

obj = {
 go: function() { alert(this); }
};

obj.go(); // (1) [object Object]

(obj.go)(); // (2) [object Object]

(method = obj.go)(); // (3) undefined

(obj.go || obj.stop)(); // (4) undefined
```

[Alla soluzione](#)

## BigInt

### ⚠ Aggiunta di recente

Questa funzionalità è stata aggiunta di recente al linguaggio. Puoi trovare l'attuale stato di supporto [https://caniuse.com/#feat=bigint ↗](https://caniuse.com/#feat=bigint).

`BigInt` è uno speciale tipo numerico che supporta numeri interi di lunghezza arbitraria.

Un bigint viene creato aggiungendo il suffisso `n` alla fine di un numero intero, oppure invocando la funzione `BigInt`, la quale crea bigints a partire da stringhe, numeri etc.

```
const bigint = 1234567890123456789012345678901234567890n;
const sameBigint = BigInt("1234567890123456789012345678901234567890");
const bigintFromNumber = BigInt(10); // equivale a 10n
```

## Operatori matematici

`BigInt` può essere utilizzato come un normale numero, ad esempio:

```
alert(1n + 2n); // 3
alert(5n / 2n); // 2
```

Da notare: la divisione `5/2` ritorna il risultato arrotondato verso lo zero, senza la parte decimale. Tutte le operazioni con bigints ritornano bigints.

Non possiamo mischiare i bigints e i numeri regolari:

```
alert(1n + 2); // Errore: Non possiamo mischiare BigInt e altri tipi
```

Dobbiamo esplicitamente convertirli se necessario: utilizzando `BigInt()` o `Number()`, in questo modo:

```
let bigint = 1n;
let number = 2;

// da number a bigint
alert(bigint + BigInt(number)); // 3

// da bigint a number
alert(Number(bigint) + number); // 3
```

Le operazioni di conversione sono sempre silenziose, non generano mai errori, ma se il bigint dovesse essere troppo grande per essere contenuto in un numero, i bit in eccesso verranno rimossi, quindi dobbiamo stare attenti quando facciamo queste conversioni.

## **i L'operatore di somma unaria non è supportato dai bigints**

L'operatore di somma unaria `+value` è una pratica molto conosciuta per convertire `value` ad un numero.

Per evitare confusione, non viene supportato dai bigints:

```
let bigint = 1n;

alert(+bigint); // error
```

Quindi dovremo utilizzare `Number()` per convertire un bigint in number.

## Confronti

Confronti, come `<`, `>` funzionano correttamente con i bigints e i number:

```
alert(2n > 1n); // true

alert(2n > 1); // true
```

Da notare che, poichè number e bigint appartengono a tipi differenti, possono essere uguali `==`, ma non strettamente equivalenti `===`:

```
alert(1 == 1n); // true

alert(1 === 1n); // false
```

## Operazioni booleane

Quando utilizzati all'interno di un `if` o qualsiasi altra operazione booleana, i bigints si comportano come numbers.

Ad esempio, in `if`, bigint `0n` vale `false`, gli altri valori valgono `true`:

```
if (0n) {
 // non verrà mai eseguito
}
```

Operatori booleani, come `||`, `&&` e tutti gli altri, funzionano con i bigint in maniera simile ai number:

```
alert(1n || 2); // 1 (1n viene considerato true)

alert(0n || 2); // 2 (0n viene considerato false)
```

## Polyfills

Costruire un polyfill per bigints è difficile. Il motivo è che molti operatori JavaScript, come `+`, `-` e `*` così via si comportano in maniera differente con i bigint rispetto ai numeri regolari.

Ad esempio, la divisione di bigint ritorna sempre un bigint (arrotondato se necessario).

Per poter emulare questo comportamento, un polyfill deve analizzare il codice e rimpiazzare tutti questi operatori con funzioni proprie. Fare questo può essere complesso e costerebbe molto in termini di performance.

Quindi, non esiste alcun polyfill ottimale.

Comunque, un'alternativa è stata proposta dagli sviluppatori della libreria [JSBI](#).

Questa libreria implementa i bigint utilizzando metodi propri. Possiamo utilizzare questi invece dei bigint integrati dal linguaggio:

Operazione	BigInt integrati	JSBI
Creazione da Number	<code>a = BigInt(789)</code>	<code>a = JSBI.BigInt(789)</code>
Addizione	<code>c = a + b</code>	<code>c = JSBI.add(a, b)</code>
Sottrazione	<code>c = a - b</code>	<code>c = JSBI.subtract(a, b)</code>
...	...	...

...quindi utilizzare il polyfill (Babel plugin) per convertire le invocazioni a JSBI in bigint nativi, per i browser che li supportano.

In altre parole, questo approccio suggerisce di scrivere il codice utilizzando JSBI piuttosto dei bigint integrati. JSBI funziona con i numbers proprio come i bigint integrati, emulandoli secondo quanto descritto nelle specifiche, quindi il codice sarà “bigint-ready”.

Possiamo utilizzare questo codice JSBI “così com’è” sia per i motori che non supportano i bigint che per quelli che li supportano – il polyfill convertirà le invocazioni in bigint integrati.

## Riferimenti

- [Documentazione MDN sui BigInt](#).
- [Specifiche](#).

## Soluzioni

### Hello, world!

---

#### Mostra un alert

```
<!DOCTYPE html>
<html>

<body>

<script>
```

```
 alert("I'm JavaScript!");
 </script>

</body>

</html>
```

[Apri la soluzione in una sandbox.](#) ↗

[Alla formulazione](#)

---

## Mostra un alert con uno script esterno

Il codice HTML:

```
<!DOCTYPE html>
<html>

<body>

 <script src="alert.js"></script>

</body>

</html>
```

Per il file `alert.js` nella stessa cartella:

```
alert("I'm JavaScript!");
```

[Alla formulazione](#)

## Variabili

### Lavorare con le variabili

Nel codice sotto, ogni linea corrisponde ad un elemento della lista.

```
let admin, name; // can declare two variables at once

name = "John";

admin = name;

alert(admin); // "John"
```

[Alla formulazione](#)

## Scegliere il giusto nome

Variabile del nostro pianeta

Questo è semplice:

```
let ourPlanetName = "Earth";
```

Nota che avremmo anche potuto usare un nome più corto, come `planet`, ma potrebbe non essere ovvio a cosa si riferisce. E' una buona pratica essere un po' più descrittivi. Almeno finché le variabili nonDiventanoTroppoLunghe.

Il nome del visitatore attuale

```
let currentUserName = "John";
```

Di nuovo, potremmo accorciare il nome con `userName`, se fossimo sicuri che quell'utente sia quello corrente.

I moderni editor e gli autocompletamenti rendono i nomi delle variabili molto lunghe, molto più facili da scrivere. Quindi non risparmiatevi con i nomi. Un nome con 3 parole va bene.

E se il vostro editor non ha un autocompletamento efficiente, cambiate editor [a new one](#).

[Alla formulazione](#)

## Costanti maiuscole?

Generalmente si usano lettere maiuscole per le costanti che vengono pre-inizializzate. O, in altre parole, quando il valore è conosciuto prima dell'esecuzione e vengono scritte direttamente nel codice.

In questo caso `birthday` è già conosciuta prima di eseguire il codice. Quindi in questo caso utilizziamo le lettere maiuscole.

Per quanto riguarda `age` la situazione è differente, perché viene calcolata a run-time(in esecuzione). Oggi abbiamo un'età, l'anno dopo ne avremmo un'altra. E' comunque costante perché non cambia con l'esecuzione del codice. Ma è un po' "meno costante" di `birthday`, poichè viene calcolata, quindi dovremmo utilizzare le lettere minuscole.

[Alla formulazione](#)

## Tipi di dato

### Gli apici nelle stringhe

Il backtick viene utilizzato per integrare espressioni all'interno delle stringhe  `${...}` .

```
let name = "Ilya";

// l'espressione è il numero 1
alert(`hello ${1}`); // hello 1

// l'espressione è la stringa "name"
alert(`hello ${"name"}`); // hello name

// l'espressione è una variabile che viene integrata
alert(`hello ${name}`); // hello Ilya
```

[Alla formulazione](#)

## Interazioni: alert, prompt, confirm

### Una semplice pagina

Codice JavaScript:

```
let name = prompt("What is your name?", "");
alert(name);
```

La pagina completa:

```
<!DOCTYPE html>
<html>
<body>

<script>
'use strict';

let name = prompt("What is your name?", "");
alert(name);
</script>

</body>
</html>
```

[Alla formulazione](#)

## Operatori di base

### La forma post-fissa e pre-fissa

La soluzione:

- `a = 2`
- `b = 2`
- `c = 2`
- `d = 1`

```
let a = 1, b = 1;

alert(++a); // 2, la forma pre-fissa restituisce il nuovo valore
alert(b++); // 1, la forma pre-fissa restituisce il vecchio valore

alert(a); // 2, incrementato una volta
alert(b); // 2, incrementato una volta
```

[Alla formulazione](#)

## Risultato dell'assegnazione

La soluzione:

- `a = 4` (moltiplicato per 2)
- `x = 5` (calcolato come  $1 + 4$ )

[Alla formulazione](#)

## Conversioni di tipo

```
"" + 1 + 0 = "10" // (1)
"" - 1 + 0 = -1 // (2)
true + false = 1
6 / "3" = 2
"2" * "3" = 6
4 + 5 + "px" = "9px"
"$" + 4 + 5 = "$45"
"4" - 2 = 2
"4px" - 2 = NaN
"-9" + 5 = "-9 5" // (3)
"-9" - 5 = -14 // (4)
null + 1 = 1 // (5)
undefined + 1 = NaN // (6)
"\t\n" - 2 = -2 // (7)
```

1. L'addizione con una stringa `"" + 1` converte `1` a stringa: `"" + 1 = "1"`, applichiamo la stessa regola a `"1" + 0`.
2. La sottrazione `-` (come molte altre operazioni matematiche) funziona solamente con i numeri, quindi una stringa vuota come: `""` viene convertita in `0`.
3. La somma con una stringa appende in coda alla stringa il numero `5`.
4. La sottrazione converte sempre in numero, quindi `" -9 "` diventa `-9` (gli spazi vuoti vengono ignorati).

5. `null` diventa `0` dopo la conversione numerica.
6. `undefined` diventa `Nan` dopo la conversione numerica.
7. Gli spazi all'inizio e alla fine di una stringa vengono rimossi quando questa viene convertita ad numero. In questo caso l'intera stringa è composta da spazi, come `\t`, `\n` ed uno "spazio" normale tra di essi. Quindi allo stesso modo di una stringa vuota, diventa `0`.

[Alla formulazione](#)

---

## Sistema l'addizione

La ragione è che il `prompt` ritorna delle stringhe .

Quindi le variabili hanno valori `"1"` e `"2"` rispettivamente.

```
let a = "1"; // prompt("First number?", 1);
let b = "2"; // prompt("Second number?", 2);

alert(a + b); // 12
```

Quello che dovremmo fare è convertire le stringhe in numeri prima che vengano sommate. Questo può essere fatto facendole precedere da `+` o usando `Number()`

Per esempio, appena prima di `prompt`:

```
let a = +prompt("First number?", 1);
let b = +prompt("Second number?", 2);

alert(a + b); // 3
```

Oppure in `alert`:

```
let a = prompt("First number?", 1);
let b = prompt("Second number?", 2);

alert(+a + +b); // 3
```

Usando `+` sia in modalità unaria che binaria. Sembra divertente vero?

[Alla formulazione](#)

---

## Confronti

### Confronti

```
5 > 4 → true
```

```
"apple" > "pineapple" → false
"2" > "12" → true
undefined == null → true
undefined === null → false
null == "\n0\n" → false
null === +"\n0\n" → false
```

I motivi:

1. Ovviamente è true.
2. Confronto lessicografico, quindi false.
3. Nuovamente, confronto lessicografico, il primo carattere di "2" è maggiore del primo carattere "1".
4. I valori null e undefined sono uguali solo tra di loro.
5. L'uguaglianza stretta è stretta. I tipi differenti dei due operandi portano ad un risultato false.
6. Vedi (4), null equivale solamente a undefined ...
7. Uguaglianza stretta di tipi differenti.

[Alla formulazione](#)

## Operatori condizionali: if, '?'

### if (una stringa con zero)

Certo.

Qualunque stringa ad eccezione di quella vuota ("0" non lo è)- diventa true in contesto booleano.

Possiamo eseguirlo e controllare:

```
if ("0") {
 alert('Hello');
}
```

[Alla formulazione](#)

## Il nome di JavaScript

```
<!DOCTYPE html>
<html>

<body>
 <script>
 'use strict';
```

```
let value = prompt('What is the "official" name of JavaScript?', '');

if (value == 'ECMAScript') {
 alert('Right!');
} else {
 alert("You don't know? ECMAScript!");
}

</script>

</body>

</html>
```

Alla formulazione

---

## Mostra il segno

```
let value = prompt('Type a number', 0);

if (value > 0) {
 alert(1);
} else if (value < 0) {
 alert(-1);
} else {
 alert(0);
}
```

Alla formulazione

---

## Riscrivi 'if' con '?:'

```
let result = (a + b < 4) ? 'Below' : 'Over';
```

Alla formulazione

---

## Riscrivi 'if..else' con '?:'

```
let message = (login == 'Employee') ? 'Hello' :
(login == 'Director') ? 'Greetings' :
(login == '') ? 'No login' :
'';
```

Alla formulazione

---

## Operatori logici

## Qual è il risultato dell'OR?

La risposta è 2, questo è il primo valore vero.

```
alert(null || 2 || undefined);
```

[Alla formulazione](#)

## Qual è il risultato dell'alert con l'OR?

La risposta: prima 1, poi 2.

```
alert(alert(1) || 2 || alert(3));
```

La chiamata ad `alert` non ritorna alcun valore; ossia `undefined`.

1. Il primo OR `||` valuta l'operando sinistro `alert(1)`. Questo mostra il primo messaggio, 1.
2. La funzione `alert` ritorna `undefined`, quindi OR prosegue con il secondo operando, alla ricerca di un valore vero.
3. Il secondo operando 2 è vero; quindi l'esecuzione si ferma, viene ritornato 2 e mostrato dall'alert esterno.

Non ci sarà il 3, perchè la valutazione non arriva a `alert(3)`.

[Alla formulazione](#)

## Qual è il risultato dell'AND?

La risposta è: `null`, perchè è il primo valore falso nella lista.

```
alert(1 && null && 2);
```

[Alla formulazione](#)

## Qual è il risultato degli alert?

La risposta è: 1, e poi `undefined`.

```
alert(alert(1) && alert(2));
```

La chiamata `alert` ritorna `undefined` (mostra solo un messaggio, quindi non ha nessuna valore significativo di ritorno).

Per questo `&&`, valutato l'operando di sinistra (che mostra `1`), si ferma: `undefined` è un valore falso e `&&` lo ritorna immediatamente.

[Alla formulazione](#)

---

## Il risultato di OR AND OR

La risposta è: `3`.

```
alert(null || 2 && 3 || 4);
```

La precedenza di AND `&&` è maggiore di `||`, quindi verrà eseguito per primo.

Il risultato di `2 && 3 = 3`, quindi l'espressione diventa:

```
null || 3 || 4
```

Adesso il risultato è il primo valore vero: `3`.

[Alla formulazione](#)

---

## Controlla l'intervallo

```
if (age >= 14 && age <= 90)
```

[Alla formulazione](#)

---

## Controlla l'intervallo fuori

La prima variante:

```
if (!(age >= 14 && age <= 90))
```

La seconda variante:

```
if (age < 14 || age > 90)
```

[Alla formulazione](#)

---

## Un indovinello con "if"

La risposta: il primo e il terzo verranno eseguiti.

I dettagli:

---

```
// Viene eseguito
// Il risultato di -1 || 0 = -1 è vero
if (-1 || 0) alert('first');

// Non viene eseguito
// -1 && 0 = 0, falso
if (-1 && 0) alert('second');

// Eseguito
// L'operatore && ha la precedenza su ||,
// quindi -1 && 1 vengono eseguiti per primi; la catena dentro `if` diventa:
// null || -1 && 1 -> null || 1 -> 1
if (null || -1 && 1) alert('third');
```

Alla formulazione

## Controlla il login

```
let userName = prompt("Who's there?", '');
if (userName === 'Admin') {

 let pass = prompt('Password?', '');

 if (pass === 'TheMaster') {
 alert('Welcome!');
 } else if (pass === '' || pass === null) {
 alert('Canceled');
 } else {
 alert('Wrong password');
 }

} else if (userName === '' || userName === null) {
 alert('Canceled');
} else {
 alert("I don't know you");
}
```

Nota l'indentazione verticale all'interno del blocco `if`. Tecnicamente non è richiesto, ma rende il codice molto più leggibile.

Alla formulazione

## Cicli: while e for

### Valore all'ultimo ciclo

La risposta è: 1.

```
let i = 3;
```

```
while (i) {
 alert(i--);
}
```

Ogni iterazione del ciclo decrementa `i` di 1. Il controllo `while(i)` interrompe il ciclo quando `i = 0`.

Quindi, gli step del ciclo sono (“loop unrolled”):

```
let i = 3;

alert(i--); // mostra 3, decrementa i a 2

alert(i--) // mostra 2, decrementa i a 1

alert(i--) // mostra 1, decrementa i a 0

// finito, `i` è ora 0, che convertito a booleano è falso
```

[Alla formulazione](#)

## Quali valori mostrerà il ciclo while?

L'esercizio dimostrava come le forme prefissa/postfissa possano portare a risultati differenti.

1.

**Da 1 a 4**

```
let i = 0;
while (++i < 5) alert(i);
```

Il primo valore è `i = 1`, perchè `++i` incrementa prima `i` e poi ritorna il nuovo valore. Quindi il primo confronto è `1 < 5` e `alert` mostra `1`.

Poi seguono `2, 3, 4...` – i valori vengono mostrati uno dopo l'altro. Il confronto utilizza sempre il valore incrementato, perchè `++` è posto prima della variabile.

Ultima iterazione, `i = 4` viene incrementato a `5`, il confronto `while(5 < 5)` risulta `false`, e il ciclo termina. `5` non viene mostrato.

2.

**Da 1 a 5**

```
let i = 0;
while (i++ < 5) alert(i);
```

Il primo valore è ancora `i = 1`. La forma postfissa `i++` incrementa `i` e ritorna il vecchio valore, quindi il confronto `i++ < 5` utilizza `i = 0` (a differenza di `++i < 5`).

La chiamata ad `alert` è separata. E' un istruzione che viene eseguita dopo l'incremento di `i` e il controllo della condizione. Quindi `i = 1`.

Seguono `2, 3, 4...`

Fermiamoci a `i = 4`. La forma prefissa `++i` incrementerebbe ed utilizzerebbe `5` nel controllo della condizione `i < 5`. Qui però stiamo usando la forma postfissa `i++`. Quindi incrementa `i` a `5`, ma ritorna il vecchio valore. Il confronto è dunque `while(4 < 5)` – vero; il controllo passa alla chiamata `alert`.

Il valore `i = 5` è l'ultimo, perchè nell'iterazione successiva avremmo `while(5 < 5)` e la condizione sarebbe falsa.

[Alla formulazione](#)

## Quali valori verranno mostrati dal ciclo "for"?

La risposta: da `0` a `4` in entrambi i casi.

```
for (let i = 0; i < 5; ++i) alert(i);

for (let i = 0; i < 5; i++) alert(i);
```

Questo può essere facilmente dedotto dall'algoritmo `for`:

1. Esegue come prima cosa `i = 0` (una sola volta)
2. Verifica la condizione `i < 5`
3. Se è `true` – esegue il corpo del ciclo, dove si trova `alert(i)`, poi incrementa `i` di 1

L'incremento `i++` è separato dal controllo della condizione(2). E' un'istruzione differente.

Il valore ritornato non viene utilizzato, quindi non c'è differenza tra `i++` e `++i`.

[Alla formulazione](#)

## Mostra i numeri pari con un ciclo 'for'

```
for (let i = 2; i <= 10; i++) {
 if (i % 2 == 0) {
 alert(i);
 }
}
```

Utilizziamo l'operatore modulo `%` per controllare che un numero sia pari.

[Alla formulazione](#)

## Sostituisci "for" con "while"

```
let i = 0;
while (i < 3) {
 alert(`number ${i}!`);
 i++;
}
```

[Alla formulazione](#)

## Ripeti fino a quando l'input è corretto

```
let num;

do {
 num = prompt("Enter a number greater than 100?", 0);
} while (num <= 100 && num);
```

Il ciclo `do..while` si ripete fintanto che entrambe le condizioni non risultano vere:

1. Il controllo `num <= 100` – controlla se il valore non risulti ancora maggiore di `100`.
2. Il controllo `&& num` diventa falso quando `num` è `null` o una stringa. Quindi il ciclo `while` termina.

P.S. Se `num` è `null` allora la condizione `num <= 100` è `true`, quindi senza la seconda condizione il ciclo non terminerebbe nel caso in cui l'utente prema CANCEL. Entrambe le condizioni sono necessarie.

[Alla formulazione](#)

## Mostra i numeri primi

Ci sono diversi possibili algoritmi per il nostro scopo.

Usiamo un ciclo annidato:

```
For each i in the interval {
 check if i has a divisor from 1..i
 if yes => the value is not a prime
 if no => the value is a prime, show it
}
```

Il codice usa un'etichetta:

```

let n = 10;

nextPrime:
for (let i = 2; i <= n; i++) { // per ogni i...

 for (let j = 2; j < i; j++) { // controlla i suoi divisori..
 if (i % j == 0) continue nextPrime; // se è divisibile per uno di essi, non è un numero primo
 }

 alert(i); // un numero primo
}

```

Ci sono molti modi per ottimizzare questo algoritmo. Ad esempio, potremmo controllare i divisori di `2` fino alla radice di `i`. In ogni caso, se vogliamo essere veramente efficienti su grandi intervalli, abbiamo bisogno di cambiare approccio ed affidarci ad algoritmi matematici più avanzati e complessi, come [Quadratic sieve ↗](#), [General number field sieve ↗](#) etc.

[Alla formulazione](#)

## L'istruzione "switch"

### Riscrivi il costrutto "switch" come un "if"

Per ottenere una corrispondenza precisa con il costrutto `switch`, `if` deve utilizzare l'uguaglianza stretta `'==='`.

Per le stringhe fornite, tuttavia, un semplice `'=='` può bastare.

```

if(browser == 'Edge') {
 alert("You've got the Edge!");
} else if (browser == 'Chrome'
 || browser == 'Firefox'
 || browser == 'Safari'
 || browser == 'Opera') {
 alert('Okay we support these browsers too');
} else {
 alert('We hope that this page looks ok!');
}

```

Da notare: il costrutto `browser == 'Chrome' || browser == 'Firefox' ...` viene diviso in più linee per una maggiore leggibilità.

Il costrutto `switch` risulta però più pulito e descrittivo.

[Alla formulazione](#)

### Riscrivi "if" utilizzando "switch"

I primi due controlli vengono trasformati in due `case` separati. Il terzo controllo viene diviso in due `case` raggruppati:

```
let a = +prompt('a?', '');

switch (a) {
 case 0:
 alert(0);
 break;

 case 1:
 alert(1);
 break;

 case 2:
 case 3:
 alert('2,3');
 break;
}
```

Da notare: il `break` alla fine non è richiesto. Lo abbiamo messo per rendere il codice pronto ad aggiornamenti futuri.

In futuro, potremmo voler aggiungere un ulteriore `case`, ad esempio `case 4`. E se ci dimentichiamo di aggiungere il `break` prima di scrivere il nuovo `case`, al termine del `case 3` ci sarà un errore. Quindi aggiungere il `break` è una sorta di prevenzione.

[Alla formulazione](#)

## Funzioni

### E' richiesto "else"?

No, non ci sono differenze.

[Alla formulazione](#)

### Riscrivi la funzione utilizzando '?' o '||'

Utilizzando l'operatore `?`:

```
function checkAge(age) {
 return (age > 18) ? true : confirm('Did parents allow you?');
}
```

Utilizzando OR `||` (la variante più breve):

```
function checkAge(age) {
```

```
 return (age > 18) || confirm('Did parents allow you?');
}
```

Nota che le parentesi che includono `age > 18` non sono obbligatorie. Vengono utilizzate per migliorare la leggibilità.

Alla formulazione

---

## Funzione min(a, b)

Una soluzione utilizza `if`:

```
function min(a, b) {
 if (a < b) {
 return a;
 } else {
 return b;
 }
}
```

Un'altra soluzione, con l'operatore `'?' :`

```
function min(a, b) {
 return a < b ? a : b;
}
```

P.S. Nel caso di uguaglianza `a == b` non ha importanza quale si ritorna.

Alla formulazione

---

## Funzione pow(x,n)

```
function pow(x, n) {
 let result = x;

 for (let i = 1; i < n; i++) {
 result *= x;
 }

 return result;
}

let x = prompt("x?", '');
let n = prompt("n?", '');

if (n < 1) {
 alert(`Power ${n} is not supported, use a positive integer`);
} else {
 alert(pow(x, n));
}
```

Alla formulazione

## Arrow functions, le basi

### Riscrivi usando le arrow functions

```
function ask(question, yes, no) {
 if (confirm(question)) yes();
 else no();
}

ask(
 "Do you agree?",
 () => alert("You agreed."),
 () => alert("You canceled the execution.")
);
```

Sembra più corto e pulito, vero?

Alla formulazione

## Stile di programmazione

### Pessimo stile

Avrete notato:

```
function pow(x,n) // <- no space between arguments
{ // <- figure bracket on a separate line
 let result=1; // <- no spaces before or after =
 for(let i=0;i<n;i++) {result*=x;} // <- no spaces
 // the contents of { ... } should be on a new line
 return result;
}

let x=prompt("x?",''), n=prompt("n?",'') // <-- technically possible,
// but better make it 2 lines, also there's no spaces and missing ;
if (n<=0) // <- no spaces inside (n <= 0), and should be extra line above it
{ // <- figure bracket on a separate line
 // below - long lines can be split into multiple lines for improved readability
 alert(`Power ${n} is not supported, please enter an integer number greater than zero`);
}
else // <- could write it on a single line like "} else {""
{
 alert(pow(x,n)) // no spaces and missing ;
```

La variante sistemata:

```

function pow(x, n) {
 let result = 1;

 for (let i = 0; i < n; i++) {
 result *= x;
 }

 return result;
}

let x = prompt("x?", "");
let n = prompt("n?", "");

if (n <= 0) {
 alert(`Power ${n} is not supported,
 please enter an integer number greater than zero`);
} else {
 alert(pow(x, n));
}

```

[Alla formulazione](#)

## Test automatici con Mocha

### Cosa c'è di sbagliato in questo test?

Il test dimostra una delle tentazioni che uno sviluppatore potrebbe incontrare mentre scrive dei test.

Quello che abbiamo qui sono 3 test, ma sono stati scritti come una singola funzione con 3 assunzioni.

Qualche volta può risultare più semplice scrivere in questo modo, ma in caso di errori, risulta molto meno ovvio cosa è andato storto.

Se si genera un errore all'interno di un flusso d'esecuzione complesso, dovremmo controllare ogni dato. Saremmo costretti a *debuggare il test*.

Una scelta migliore potrebbe essere di rompere i test in più `it` scrivendo chiaramente gli inpute gli output.

Come qui:

```

describe("Raises x to power n", function() {
 it("5 in the power of 1 equals 5", function() {
 assert.equal(pow(5, 1), 5);
 });

 it("5 in the power of 2 equals 25", function() {
 assert.equal(pow(5, 2), 25);
 });
}

```

```
it("5 in the power of 3 equals 125", function() {
 assert.equal(pow(5, 3), 125);
});
});
```

Rimpiazziamo quindi il singolo `it` con `describe` e creiamo un gruppo di blocchi `it`. Ora se qualche test fallisce saremmo in grado di vedere chiaramente quale.

Possiamo anche isolare un singolo test ed eseguirlo in solitaria scrivendo `it.only` piuttosto di `it`:

```
describe("Raises x to power n", function() {
 it("5 in the power of 1 equals 5", function() {
 assert.equal(pow(5, 1), 5);
 });

 // Mocha will run only this block
 it.only("5 in the power of 2 equals 25", function() {
 assert.equal(pow(5, 2), 25);
 });

 it("5 in the power of 3 equals 125", function() {
 assert.equal(pow(5, 3), 125);
 });
});
```

[Alla formulazione](#)

## Oggetti

### Hello, object

```
let user = {};
user.name = "John";
user.surname = "Smith";
user.name = "Pete";
delete user.name;
```

[Alla formulazione](#)

### Controlla se è vuoto

E' sufficiente eseguire un ciclo e ritornare `false` se l'oggetto contiene almeno una proprietà.

```
function isEmpty(obj) {
 for (let key in obj) {
 return false;
 }
}
```

```
 return true;
}
```

```
function isEmpty(obj) {
 for (let key in obj) {
 // if the loop has started, there is a property
 return false;
 }
 return true;
}
```

Apri la soluzione con i test in una sandbox. [↗](#)

[Alla formulazione](#)

---

## Oggetti costanti?

Certo, funziona senza problemi.

La keyword `const` protegge la variabile solo da riassegnazioni.

In altre parole, `user` memorizza un riferimento all'oggetto. Questo non può cambiare. Ma l'oggetto contenuto non ha nessun vincolo.

```
const user = {
 name: "John"
};

// funziona (abbiamo cambiato una proprietà dell'oggetto)
user.name = "Pete";

// errore (abbiamo cercato di cambiare tutto valore in un colpo solo)
user = 123;
```

[Alla formulazione](#)

---

## Somma le proprietà dell'oggetto

```
let salaries = {
 John: 100,
 Ann: 160,
 Pete: 130
};

let sum = 0;
for (let key in salaries) {
 sum += salaries[key];
}

alert(sum); // 390
```

[Alla formulazione](#)

## Moltiplica le proprietà numeriche per 2

```
function multiplyNumeric(obj) { for (let key in obj) { if (typeof obj[key] == 'number') { obj[key] *= 2; } } }
```

```
function multiplyNumeric(obj) {
 for (let key in obj) {
 if (typeof obj[key] == 'number') {
 obj[key] *= 2;
 }
 }
}
```

Apri la soluzione con i test in una sandbox. [↗](#)

[Alla formulazione](#)

## Metodi degli oggetti,"this"

### Utilizzare "this" in un oggetto letterale

**Risposta: un errore.**

Provate:

```
function makeUser() {
 return {
 name: "John",
 ref: this
 };
}

let user = makeUser();

alert(user.ref.name); // Error: Cannot read property 'name' of undefined
```

Questo avviene perché le regole che impostano `this` non guardano agli oggetti letterali.

Qui il valore di `this` dentro `makeUser()` è `undefined`, perché viene chiamato dentro una funzione, non un metodo.

Gli oggetti letterali non hanno alcun effetto su `this`. Il valore di `this` è unico per tutta la funzione, quindi i blocchi di codice e gli oggetti letterali che vi si trovano dentro non hanno alcuna importanza.

Quindi `ref: this` prende il `this` della funzione.

Possiamo riscrivere la funzione e ritornare lo stesso `this` con valore `undefined`:

```
function makeUser(){
 return this; // questa volta non c'e' un oggetto letterale
}

alert(makeUser().name); // Error: Cannot read property 'name' of undefined
```

Come possiamo vedere, il risultato di `alert( makeUser().name )` è lo stesso di `alert( user.ref.name )` nell'esempio precedente.

Qui abbiamo il caso opposto:

```
function makeUser() {
 return {
 name: "John",
 ref() {
 return this;
 }
 };
}

let user = makeUser();

alert(user.ref().name); // John
```

Ora funziona, perché `user.ref()` è un metodo. E il valore di `this` si riferisce all'oggetto prima del punto `.`.

Alla formulazione

## Create una calcolatrice

```
let calculator = {
 sum() {
 return this.a + this.b;
 },

 mul() {
 return this.a * this.b;
 },

 read() {
 this.a = +prompt('a?', 0);
 this.b = +prompt('b?', 0);
 }
};

calculator.read();
alert(calculator.sum());
alert(calculator.mul());
```

Apri la soluzione con i test in una sandbox. ↗

[Alla formulazione](#)

## Concatenazione

La soluzione sta nel ritornare l'oggetto stesso ad ogni chiamata.

```
let ladder = {
 step: 0,
 up() {
 this.step++;
 return this;
 },
 down() {
 this.step--;
 return this;
 },
 showStep() {
 alert(this.step);
 return this;
 }
};

ladder.up().up().down().showStep().down().showStep(); // shows 1 then 0
```

Possiamo anche scrivere una singola chiamata per riga. Per catene molto lunghe diventa più leggibile:

```
ladder
 .up()
 .up()
 .down()
 .showStep() // 1
 .down()
 .showStep(); // 0
```

[Apri la soluzione con i test in una sandbox.](#) ↗

[Alla formulazione](#)

## Costruttore, operatore "new"

### Due funzioni – un oggetto

Si, è possibile.

Se una funzione ritorna un oggetto, questo verrà ritornato da `new` invece di `this`.

Quindi, le due funzioni potrebbero ritornare un oggetto definito esternamente `obj`:

```
let obj = {};
```

```
function A() { return obj; }
function B() { return obj; }

alert(new A() == new B()); // true
```

Alla formulazione

---

## Create una nuova Calculator

```
function Calculator() {

 this.read = function() {
 this.a = +prompt('a?', 0);
 this.b = +prompt('b?', 0);
 };

 this.sum = function() {
 return this.a + this.b;
 };

 this.mul = function() {
 return this.a * this.b;
 };
}

let calculator = new Calculator();
calculator.read();

alert("Sum=" + calculator.sum());
alert("Mul=" + calculator.mul());
```

Apri la soluzione con i test in una sandbox. ↗

Alla formulazione

---

## Create un nuovo Accumulator

```
function Accumulator(startingValue) {
 this.value = startingValue;

 this.read = function() {
 this.value += +prompt('How much to add?', 0);
 };
}

let accumulator = new Accumulator(1);
accumulator.read();
accumulator.read();
alert(accumulator.value);
```

Apri la soluzione con i test in una sandbox. ↗

[Alla formulazione](#)

## Metodi dei tipi primitivi

### Posso aggiungere una proprietà ad una stringa?

Provate ad eseguirlo:

```
let str = "Hello";

str.test = 5; // (*)

alert(str.test);
```

Potremmo ottenere due diversi risultati:

1. `undefined`
2. Un errore (strict mode).

Perché? Proviamo ad esaminare cosa succede alla riga `(*)`:

1. Quando si prova ad accedere ad una proprietà di `str`, viene creato un “oggetto contenitore”.
2. L’operazione di accesso viene eseguito su questo. Quindi l’oggetto ottiene la proprietà `test`.
3. L’operazione termina e “l’oggetto contenitore” viene distrutto.

Quindi, nell’ultima riga di codice, `str` non possiede alcuna traccia di quella proprietà. Viene creato un nuovo oggetto per ogni operazione su un tipo stringa.

**Questo esempio mostra chiaramente che le variabili primitive non sono oggetti.**

Le variabili di tipo primitivo infatti non possono memorizzare dati.

[Alla formulazione](#)

## Numeri

### Sommate i numeri forniti dall’utente

```
let a = +prompt("The first number?", "");
let b = +prompt("The second number?", "");

alert(a + b);
```

Notate la somma unaria `+` prima di `prompt`. Converte immediatamente la stringa in numero.

Altrimenti `a` e `b` vengono interpretate come stringhe e la loro somma sarebbe la loro concatenazione, cioè : `"1" + "2" = "12"`.

[Alla formulazione](#)

## Perchè `6.35.toFixed(1) == 6.3?`

Internamente il decimale `6.35` possiede una rappresentazione binaria infinita. Come sempre in questi casi, viene memorizzato con una perdita di precisione.

Vediamo infatti:

```
alert(6.35.toFixed(20)); // 6.3499999999999964473
```

La perdita di precisione può causare sia incremento che decremento del numero. In questo caso diventa leggermente più piccolo, questa è la spiegazione per cui viene arrotondato per difetto.

Mentre `1.35` ?

```
alert(1.35.toFixed(20)); // 1.35000000000000008882
```

In questo caso la perdita di precisione rende il numero più grande, quindi viene arrotondato per eccesso.

**Come possiamo risolvere il problema di `6.35` se vogliamo che venga arrotondato correttamente?**

Dovremmo prima avvicinarlo il più possibile ad un numero intero:

```
alert((6.35 * 10).toFixed(20)); // 63.50000000000000000000000000
```

Notate che `63.5` non provoca perdita di precisione. Infatti la parte decimale `0.5` vale `1/2`. I decimali che sono potenze di `2` vengono rappresentati perfettamente nel sistema binario, ora possiamo quindi arrotondare il numero:

```
alert(Math.round(6.35 * 10) / 10); // 6.35 -> 63.5 -> 64(rounded) -> 6.4
```

[Alla formulazione](#)

## Ripeti finché non viene inserito un numero

```
function readNumber() {
 let num;
```

```

do {
 num = prompt("Enter a number please?", 0);
} while (!isFinite(num));

if (num === null || num === '') return null;

return +num;
}

alert(`Read: ${readNumber()}`);

```

La soluzione risulta essere leggermente più complessa poiché è necessario trattare i casi `null`/riga vuota.

In questo modo possiamo quindi accettare input finché non viene inserito un “numero valido”. Sia `null` (cancel) sia una riga vuota soddisfano questa condizione, poiché la loro forma numerica vale `0`.

Dopo esserci fermati, abbiamo bisogno di gestire i casi `null` e riga vuota diversamente (ritornando `null`), poiché convertirli alla forma numerica li “trasformerebbe” in `0`.

[Apri la soluzione con i test in una sandbox.](#) ↗

[Alla formulazione](#)

---

## Un ciclo infinito occasionale

Questo accade perché `i` non sarà mai uguale `10`.

Eseguitelo per vedere il *vero* risultato di `i`:

```

let i = 0;
while (i < 11) {
 i += 0.2;
 if (i > 9.8 && i < 10.2) alert(i);
}

```

Nessuno di questi sarà esattamente `10`.

Questo tipo di errori accadono a causa della perdita di precisione quando sommiamo decimali come `0.2`.

Conclusione: evitate controlli di uguaglianza quando utilizzate numeri decimali.

[Alla formulazione](#)

---

## Un numero random fra min e max

Abbiamo bisogno di far “scorrere” l’intervallo da `0...1` a `min ... max`.

Questo può essere ottenuto con due passi:

1. Se moltiplichiamo un numero casuale compreso tra 0...1 per `max-min`, l'intervallo dei possibili valori cresce da `0..1` a `0..max-min`.
2. Ora se aggiungiamo `min`, il possibile intervallo si estende da `min` a `max`.

La funzione:

```
function random(min, max) {
 return min + Math.random() * (max - min);
}

alert(random(1, 5));
alert(random(1, 5));
alert(random(1, 5));
```

Alla formulazione

## Un intero casuale tra min e max

La soluzione più semplice non funziona

La più semplice, ma errata soluzione genera un valore compreso tra `min` e `max` e lo arrotonda:

```
function randomInteger(min, max) {
 let rand = min + Math.random() * (max - min);
 return Math.round(rand);
}

alert(randomInteger(1, 3));
```

La funzione esegue correttamente, ma il risultato è logicamente errato. La probabilità di ottenere i limiti `min` e `max` è due volte minore di qualsiasi altro numero.

Se provate ad eseguire il codice sopra molte volte, potrete notare che `2` apparirà molto spesso.

Questo accade perché `Math.round()` genera un numero casuale nell'intervallo `1..3` e lo arrotonda:

```
values from 1 ... to 1.4999999999 become 1
values from 1.5 ... to 2.4999999999 become 2
values from 2.5 ... to 2.9999999999 become 3
```

Ora possiamo vedere chiaramente che `1` ha due volte meno probabilità del `2`. Lo stesso vale per il `3`.

La soluzione corretta

Ci sono diverse soluzioni funzionanti. Una di queste consiste nell’“aggiustare” i bordi dell’intervallo. Per assicurare i casi limite, possiamo generare valori casuali compresi tra `0.5` to `3.5`, questa tecnica fornirebbe all’intero intervallo la stessa probabilità:

```
function randomInteger(min, max) {
 // now rand is from (min-0.5) to (max+0.5)
 let rand = min - 0.5 + Math.random() * (max - min + 1);
 return Math.round(rand);
}

alert(randomInteger(1, 3));
```

Un modo alternativo potrebbe essere l’utilizzo di `Math.floor` su un numero casuale nell’intervallo `min – max+1`:

```
function randomInteger(min, max) {
 // here rand is from min to (max+1)
 let rand = min + Math.random() * (max + 1 - min);
 return Math.floor(rand);
}

alert(randomInteger(1, 3));
```

Ora l’intero intervallo viene considerato allo stesso modo:

```
values from 1 ... to 1.9999999999 become 1
values from 2 ... to 2.9999999999 become 2
values from 3 ... to 3.9999999999 become 3
```

L’intero intervall ha la stessa lunghezza, rendendo la distribuzione uniforme.

[Alla formulazione](#)

## Stringhe

### Prima lettera maiuscola

Non possiamo “rimpiazzare” il primo carattere, perché in JavaScript le stringhe sono immutabili.

Possiamo invece creare una nuova stringa basata su quella già esistente, con la prima lettera maiuscola:

```
let newStr = str[0].toUpperCase() + str.slice(1);
```

C’è comunque un piccolo problema. Se `str` è vuota, allora `str[0]` è `undefined`, quindi otterremo un errore.

Ci sono due possibili varianti qui:

1. Utilizzare `str.charAt(0)`, che ritorna sempre una stringa (eventualmente vuota).
2. Aggiungere una verifica di stringa vuota.

Qui abbiamo scelto la seconda variante:

```
function ucFirst(str) {
 if (!str) return str;

 return str[0].toUpperCase() + str.slice(1);
}

alert(ucFirst("john")); // John
```

Apri la soluzione con i test in una sandbox. [↗](#)

[Alla formulazione](#)

---

## Controllo spam

Per far sì che la ricerca sia case-insensitive (non distingue lettere minuscole da quelle maiuscole), portiamo l'intera stringa a lettere minuscole e poi eseguiamo la ricerca:

```
function checkSpam(str) {
 let lowerStr = str.toLowerCase();

 return lowerStr.includes('viagra') || lowerStr.includes('xxx');
}

alert(checkSpam('buy ViAgRA now'));
alert(checkSpam('free xxxx'));
alert(checkSpam("innocent rabbit"));
```

Apri la soluzione con i test in una sandbox. [↗](#)

[Alla formulazione](#)

---

## Troncate il testo

La lunghezza massima deve essere `maxlength`, quindi abbiamo bisogno di troncare la stringa, per fare spazio al carattere “...”

Da notare che esiste un carattere Unicode che identifica “...”. Questo non è lo stesso che usare tre punti.

```
function truncate(str, maxlength) {
 return (str.length > maxlength) ?
 str.slice(0, maxlength - 1) + '...' : str;
}
```

[Apri la soluzione con i test in una sandbox.](#)

[Alla formulazione](#)

---

## Estraete il denaro

```
function extractCurrencyValue(str) {
 return +str.slice(1);
}
```

[Apri la soluzione con i test in una sandbox.](#)

[Alla formulazione](#)

---

## Array

### L'array è stato copiato?

La risposta è **4**:

```
let fruits = ["Apples", "Pear", "Orange"];

let shoppingCart = fruits;

shoppingCart.push("Banana");

alert(fruits.length); // 4
```

Questo perché gli array sono oggetti. Quindi `shoppingCart` e `fruits` sono riferimenti allo stesso array.

[Alla formulazione](#)

---

### Operazioni sugli array.

```
let styles = ["Jazz", "Blues"];
styles.push("Rock-n-Roll");
styles[Math.floor(styles.length - 1) / 2] = "Classics";
alert(styles.shift());
styles.unshift("Rap", "Reggae");
```

[Alla formulazione](#)

---

## Chiamata di funzione

La chiamata `arr[2]()` è sintatticamente equivalente a `obj[method]()`; al posto di `obj` abbiamo `arr`, e al posto di `method` abbiamo `2`.

Quindi abbiamo una chiamata al metodo `arr[2]`. Naturalmente, riceve il riferimento a `this` e ritorna l'array:

```
let arr = ["a", "b"];
arr.push(function() {
 alert(this);
})
arr[2](); // a,b,function(){...}
```

L'array ha 3 valori: inizialmente ne ha due, successivamente viene aggiunta la funzione.

[Alla formulazione](#)

## Somma dei numeri inseriti

Da notare un dettaglio sottile ma importante. Non convertiamo immediatamente `value` ad un numero subito dopo averlo prelevato con `prompt`, perché successivamente tramite `value = +value` non saremmo in grado di distinguere una stringa vuota da uno zero. Quindi è necessario eseguire la conversione in un secondo momento.

```
function sumInput() {
 let numbers = [];
 while (true) {
 let value = prompt("A number please?", 0);
 // should we cancel?
 if (value === "" || value === null || !isFinite(value)) break;
 numbers.push(+value);
 }
 let sum = 0;
 for (let number of numbers) {
 sum += number;
 }
 return sum;
}
alert(sumInput());
```

[Alla formulazione](#)

## Il sub-array massimo

La soluzione più lenta

Possiamo calcolare tutte le somme possibili.

La via più semplice è di prendere ogni elemento e calcolare la somma di tutti i sotto-array possibili.

Ad esempio, per `[-1, 2, 3, -9, 11]`:

```
// Starting from -1:
-1
-1 + 2
-1 + 2 + 3
-1 + 2 + 3 + (-9)
-1 + 2 + 3 + (-9) + 11

// Starting from 2:
2
2 + 3
2 + 3 + (-9)
2 + 3 + (-9) + 11

// Starting from 3:
3
3 + (-9)
3 + (-9) + 11

// Starting from -9
-9
-9 + 11

// Starting from 11
11
```

Il codice è un ciclo annidato: il ciclo esterno processa tutti gli elementi dell'array, quello interno esegue le somme a partire dall'elemento corrente.

```
function getMaxSubSum(arr) {
 let maxSum = 0; // if we take no elements, zero will be returned

 for (let i = 0; i < arr.length; i++) {
 let sumFixedStart = 0;
 for (let j = i; j < arr.length; j++) {
 sumFixedStart += arr[j];
 maxSum = Math.max(maxSum, sumFixedStart);
 }
 }

 return maxSum;
}

alert(getMaxSubSum([-1, 2, 3, -9])); // 5
alert(getMaxSubSum([-1, 2, 3, -9, 11])); // 11
alert(getMaxSubSum([-2, -1, 1, 2])); // 3
alert(getMaxSubSum([1, 2, 3])); // 6
alert(getMaxSubSum([100, -9, 2, -3, 5])); // 100
```

La soluzione ha una complessità di  $O(n^2)$  ↗. In altre parole, se l'array fosse 2 volte più grande, l'algoritmo lavorerebbe 4 volte più lentamente.

Per grandi array (1000, 10000 o più elementi) questi algoritmi possono portare ad enormi attese.

### Soluzione performante

Iniziamo ad esaminare l'array mantenendo la somma parziale degli elementi nella variabile `s`. Se `s` diventa negativa, allora assegniamo `s=0`. La somma di tutte queste `s` sarà la risposta.

Se la risposta vi sembra troppo vaga, date un'occhiata al codice:

```
function getMaxSubSum(arr) {
 let maxSum = 0;
 let partialSum = 0;

 for (let item of arr) { // for each item of arr
 partialSum += item; // add it to partialSum
 maxSum = Math.max(maxSum, partialSum); // remember the maximum
 if (partialSum < 0) partialSum = 0; // zero if negative
 }

 return maxSum;
}

alert(getMaxSubSum([-1, 2, 3, -9])); // 5
alert(getMaxSubSum([-1, 2, 3, -9, 11])); // 11
alert(getMaxSubSum([-2, -1, 1, 2])); // 3
alert(getMaxSubSum([100, -9, 2, -3, 5])); // 100
alert(getMaxSubSum([1, 2, 3])); // 6
alert(getMaxSubSum([-1, -2, -3])); // 0
```

L'algoritmo richiede esattamente uno solo scorrimento dell'array, quindi la complessità è  $O(n)$ .

Potete trovare maggiori dettagli riguardo l'algoritmo qui: [Maximum subarray problem ↗](#). Se ancora non vi risulta ovvio il funzionamento, esamineate più in dettaglio il codice fornito sopra.

[Apri la soluzione con i test in una sandbox.](#) ↗

[Alla formulazione](#)

## Metodi per gli array

### Traducete border-left-width in borderLeftWidth

```
function camelize(str) {
```

```

return str
 .split('-') // divide 'my-long-word' in un array ['my', 'long', 'word']
 .map(
 // rende maiuscole le prime lettere di tutti gli elementi dell'array eccetto il pri
 // trasforma ['my', 'long', 'word'] in ['My', 'Long', 'Word']
 (word, index) => index == 0 ? word : word[0].toUpperCase() + word.slice(1)
)
 .join(''); // unisce ['my', 'Long', 'Word'] in 'myLongWord'
}

```

Apri la soluzione con i test in una sandbox. ↗

Alla formulazione

---

## Filtri

```

function filterRange(arr, a, b) {
 //aggiunte parentesi attorno all'espressione per una migliore leggibilità
 return arr.filter(item => (a <= item && item <= b));
}

let arr = [5, 3, 8, 1];

let filtered = filterRange(arr, 1, 4);

alert(filtered); // 3,1 (i valori filtrati)

alert(arr); // 5,3,8,1 (non modificato)

```

Apri la soluzione con i test in una sandbox. ↗

Alla formulazione

---

## Filtrare un range "sul post"

```

function filterRangeInPlace(arr, a, b) {

 for (let i = 0; i < arr.length; i++) {
 let val = arr[i];

 // rimuove se fuori dal range
 if (val < a || val > b) {
 arr.splice(i, 1);
 i--;
 }
 }

 let arr = [5, 3, 8, 1];

 filterRangeInPlace(arr, 1, 4); // rimossi tutti i numeri tranne quelli da 1 a 4

 alert(arr); // [3, 1]
}

```

[Apri la soluzione con i test in una sandbox.](#)

[Alla formulazione](#)

## Riordinare in ordine decrescente

```
let arr = [5, 2, 1, -10, 8];

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

alert(arr);
```

[Alla formulazione](#)

## Copiare e ordinare un array

Possiamo utilizzare `slice()` per fare una copia e solo dopo riordinarla:

```
function copySorted(arr) {
 return arr.slice().sort();
}

let arr = ["HTML", "JavaScript", "CSS"];

let sorted = copySorted(arr);

alert(sorted);
alert(arr);
```

[Alla formulazione](#)

## Create una calcolatrice estensibile

- Da notare come vengono memorizzati i metodi. Vengono semplicemente aggiunti all'interno dell'oggetto.
- Tutti i test e le conversioni numeriche vengono effettuati nel metodo `calculate`. In futuro potrebbe essere esteso per supportare espressioni molto più complesse.

```
function Calculator() {

 this.methods = {
 "-": (a, b) => a - b,
 "+": (a, b) => a + b
 };

 this.calculate = function(str) {

 let split = str.split(' '),
 a = +split[0],
```

```

op = split[1],
b = +split[2];

if (!this.methods[op] || isNaN(a) || isNaN(b)) {
 return NaN;
}

return this.methods[op](a, b);
};

this.addMethod = function(name, func) {
 this.methods[name] = func;
};
}

```

Apri la soluzione con i test in una sandbox. ↗

Alla formulazione

---

## Map di nomi

```

let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let users = [john, pete, mary];

let names = users.map(item => item.name);

alert(names); // John, Pete, Mary

```

Alla formulazione

---

## Map di oggetti

```

let john = { name: "John", surname: "Smith", id: 1 };
let pete = { name: "Pete", surname: "Hunt", id: 2 };
let mary = { name: "Mary", surname: "Key", id: 3 };

let users = [john, pete, mary];

let usersMapped = users.map(user => ({
 fullName: `${user.name} ${user.surname}`,
 id: user.id
}));

/*
usersMapped = [
 { fullName: "John Smith", id: 1 },
 { fullName: "Pete Hunt", id: 2 },
 { fullName: "Mary Key", id: 3 }
]
*/

```

```
alert(usersMapped[0].id); // 1
alert(usersMapped[0].fullName); // John Smith
```

Da notare che nell'arrow function abbiamo bisogno di utilizzare un'ulteriore parentesi.

Non possiamo scrivere semplicemente:

```
let usersMapped = users.map(user => {
 fullName: `${user.name} ${user.surname}`,
 id: user.id
});
```

Se ricordate, ci sono due tipi di arrow function: senza corpo `value => expr` e con il corpo `value => {...}`.

Qui JavaScript tratterà `{` come l'inizio del corpo della funzione, non l'inizio dell'oggetto. Questo trucco viene utilizzato per racchiuderle nelle normali parentesi:

```
let usersMapped = users.map(user => ({
 fullName: `${user.name} ${user.surname}`,
 id: user.id
}));
```

Ora funziona.

[Alla formulazione](#)

## Riordinare oggetti per età

```
function sortByAge(arr) {
 arr.sort((a, b) => a.age - b.age);
}

let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let arr = [pete, john, mary];

sortByAge(arr);

// ordinato: [john, mary, pete]
alert(arr[0].name); // John
alert(arr[1].name); // Mary
alert(arr[2].name); // Pete
```

[Alla formulazione](#)

## Rimescolare un array

La soluzione più semplice potrebbe essere:

```

function shuffle(array) {
 array.sort(() => Math.random() - 0.5);
}

let arr = [1, 2, 3];
shuffle(arr);
alert(arr);

```

Questa in qualche modo funziona, perché `Math.random()` è un numero casuale che può essere sia positivo che negativo, quindi la funzione riordina gli elementi casualmente.

Con questa funzione di ordinamento, non tutte le permutazioni hanno la stessa probabilità.

Ad esempio, considerando il codice sotto. Esegue `shuffle` 1000000 di volte e conta il numero di occorrenze di tutti i risultati possibili:

```

function shuffle(array) {
 array.sort(() => Math.random() - 0.5);
}

// counts of appearances for all possible permutations
let count = {
 '123': 0,
 '132': 0,
 '213': 0,
 '231': 0,
 '321': 0,
 '312': 0
};

for (let i = 0; i < 1000000; i++) {
 let array = [1, 2, 3];
 shuffle(array);
 count[array.join('')]++;
}

// show counts of all possible permutations
for (let key in count) {
 alert(` ${key}: ${count[key]}`);
}

```

Un esempio di risultato possibile (dipende dal motore JS):

```

123: 250706
132: 124425
213: 249618
231: 124880
312: 125148
321: 125223

```

Possiamo chiaramente vedere che le combinazioni `123` e `213` appaiono molto più spesso delle altre.

Il risultato del codice potrebbe variare in base al motore JavaScript, ma già possiamo notare che questo tipo di approccio è inaccettabile.

Perché non funziona? Generalmente parlando, `sort` è una “scatola nera”: gli passiamo un array ed una funzione di confronto e ci aspettiamo di ottenere l’array ordinato. Ma a causa della difficoltà nell’implementazione della casualità la scatola nera potrebbe funzionare male; quanto male, dipende dal motore JavaScript.

Esistono altri modi per questo compito. Ad esempio, c’è un ottimo algoritmo chiamato [Fisher-Yates shuffle ↗](#). L’idea è di attraversare l’array in ordine inverso e di scambiare l’elemento con un altro casuale, che venga prima di lui:

```
function shuffle(array) {
 for (let i = array.length - 1; i > 0; i--) {
 let j = Math.floor(Math.random() * (i + 1)); // indice casuale da 0 a i

 //scambia gli elementi array[i] e array[j]
 // usiamo la sintassi "destructuring assignment"
 //troverai maggiori dettagli su questa sintassi nei capitoli seguenti
 //potrebbe essere scritto come
 // let t = array[i]; array[i] = array[j]; array[j] = t
 [array[i], array[j]] = [array[j], array[i]];
 }
}
```

Proviamo ad eseguire lo stesso test:

```
function shuffle(array) {
 for (let i = array.length - 1; i > 0; i--) {
 let j = Math.floor(Math.random() * (i + 1));
 [array[i], array[j]] = [array[j], array[i]];
 }
}

// counts of appearances for all possible permutations
let count = {
 '123': 0,
 '132': 0,
 '213': 0,
 '231': 0,
 '321': 0,
 '312': 0
};

for (let i = 0; i < 1000000; i++) {
 let array = [1, 2, 3];
 shuffle(array);
 count[array.join('')]++;
}

// show counts of all possible permutations
for (let key in count) {
 alert(` ${key}: ${count[key]}`);
}
```

Un possibile risultato:

```
123: 166693
132: 166647
213: 166628
231: 167517
312: 166199
321: 166316
```

Ora sembra funzionare: tutte le occorrenze appaiono con la stessa probabilità.

Inoltre, anche le performance dell'algoritmo Fisher-Yates sono migliori, poiché non è richiesto alcun riordinamento.

[Alla formulazione](#)

---

## Ottenerne l'età media+

```
function getAverageAge(users) {
 return users.reduce((prev, user) => prev + user.age, 0) / users.length;
}

let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 29 };

let arr = [john, pete, mary];

alert(getAverageAge(arr)); // 28
```

[Alla formulazione](#)

---

## Filtrare un array per ottenere elementi unici

Attraversiamo gli elementi dell'array:

- Per ogni elemento controlliamo se l'array risultante già lo contiene.
- Se lo troviamo, passiamo al prossimo, altrimenti lo aggiungiamo.

```
function unique(arr) {
 let result = [];

 for (let str of arr) {
 if (!result.includes(str)) {
 result.push(str);
 }
 }

 return result;
}

let strings = ["Hare", "Krishna", "Hare", "Krishna",
```

```
"Krishna", "Krishna", "Hare", "Hare", ":-0"
];
alert(unique(strings)); // Hare, Krishna, :-0
```

Il codice funziona, ma c'è un potenziale problema di performance.

Il metodo `result.includes(str)` internamente attraversa l'array `result` e confronta ogni elemento con `str` per trovare una corrispondenza.

Quindi se ci sono 100 elementi in `result` e nessuna corrispondenza con `str`, attraverseremo l'intero array `result` eseguendo essattamente 100 confronti. Se l'array `result` è grande, ad esempio 10000, ci sarebbero 10000 di confronti.

Non è propriamente un problema, perché il motore JavaScript è molto rapido, quindi un array grande 10000 è questione di pochi microsecondi.

Ma dovremo eseguire questo test per ogni elemento di `arr` nel ciclo `for`.

Quindi se `arr.length` è 10000 avremmo qualcosa come  $10000 * 10000 = 100$  milioni di confronti. Sono molti.

Quindi la soluzione funziona bene solo con array di piccola taglia.

Più avanti nel capitolo [Map e Set](#) vedremo come ottimizzare questo metodo.

[Apri la soluzione con i test in una sandbox.](#) ↗

[Alla formulazione](#)

## Create un oggetto da un array

```
function groupById(array) {
 return array.reduce((obj, value) => {
 obj[value.id] = value;
 return obj;
 }, {})
}
```

[Apri la soluzione con i test in una sandbox.](#) ↗

[Alla formulazione](#)

## Map e Set

### Filtrare gli elementi dell'array unici

```
function unique(arr) {
```

```
 return Array.from(new Set(arr));
}
```

Apri la soluzione con i test in una sandbox. ↗

Alla formulazione

## Filtrare anagrammi

Per trovare tutti gli anagrammi, dividiamo ogni parola in lettere ed ordiniamole. Con le lettere ordinate, tutti gli anagrammi sono uguali.

Ad esempio:

```
nap, pan -> anp
ear, era, are -> aer
cheaters, hectares, teachers -> aceehrst
...
```

Utilizzeremo la variante con le lettere ordinate come chiave di una map per memorizzare un solo valore:

```
function aclean(arr) {
 let map = new Map();

 for (let word of arr) {
 // dividi la parola in lettere, ordinale e ricongiungile
 let sorted = word.toLowerCase().split(' ').sort().join(''); // (*)
 map.set(sorted, word);
 }

 return Array.from(map.values());
}

let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"];
alert(aclean(arr));
```

L'ordinamento delle lettere è fatto dalla concatenazione di chiamate alla riga (\*) .

Per convenzione le dividiamo in più linee:

```
let sorted = word // PAN
 .toLowerCase() // pan
 .split('') // ['p', 'a', 'n']
 .sort() // ['a', 'n', 'p']
 .join(''); // anp
```

Due parole diverse 'PAN' e 'nap' possiedono la stessa forma in lettere ordinate 'anp' .

La prossima lettera inserirà la parola nella map:

```
map.set(sorted, word);
```

Se abbiamo già incontrato una parola con la stessa forma, la sovrascriviamo con quella nuova, in modo tale da avere sempre una sola occorrenza all'interno della map.

Alla fine `Array.from(map.values())` prende un iteratore sui valori di `map` (non abbiamo bisogno delle chiavi nel risultato) e ne ritorna un array.

Qui potremmo anche utilizzare un normale oggetto piuttosto di `Map`, poiché le chiavi sono stringhe.

Questo è un esempio di possibile soluzione:

```
function aclean(arr) {
 let obj = {};

 for (let i = 0; i < arr.length; i++) {
 let sorted = arr[i].toLowerCase().split("").sort().join("");
 obj[sorted] = arr[i];
 }

 return Object.values(obj);
}

let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"];
alert(aclean(arr));
```

Apri la soluzione con i test in una sandbox. ↗

[Alla formulazione](#)

## Chiavi iterabili

Questo accade perché `map.keys()` ritorna un oggetto iterabile, non un array.

Possiamo convertirlo in un array utilizzando `Array.from`:

```
let map = new Map();

map.set("name", "John");

let keys = Array.from(map.keys()); // This line is highlighted in yellow

keys.push("more");

alert(keys); // name, more
```

[Alla formulazione](#)

# WeakMap e WeakSet

## Memorizzare le bandiere non visualizzate

Memorizziamo i messaggi letti in `WeakSet`:

```
let messages = [
 {text: "Hello", from: "John"},
 {text: "How goes?", from: "John"},
 {text: "See you soon", from: "Alice"}
];

let readMessages = new WeakSet();

//due messaggi sono stati letti
readMessages.add(messages[0]);
readMessages.add(messages[1]);
// readMessages ha due elementi

//...leggiamo nuovamente il primo messaggio!
readMessages.add(messages[0]);
// readMessages ha 2 elementi unici

//risposta: message[0] è stato letto?
alert("Read message 0: " + readMessages.has(messages[0])); // true

messages.shift();
// ora readMessages ha un elemento (teoricamente la memoria potrebbe essere ripulita dopo)
```

La struttura `WeakSet` consente di memorizzare un insieme di messaggi e di verificare facilmente la presenza di un dato messaggio.

Viene ripulita automaticamente. Il lato negativo è che non possiamo eseguire iterazioni. Non possiamo ottenere direttamente “tutti i messaggi letti”. Ma possiamo farlo iterando su tutti i messaggi e filtrando tutti quelli che sono presenti nel set.

Another, different solution could be to add a property like `message.isRead=true` to a message after it's read. As messages objects are managed by another code, that's generally discouraged, but we can use a symbolic property to avoid conflicts.

Un'altra soluzione potrebbe essere aggiungere una proprietà come `message.isRead=true`, ma farlo potrebbe essere pericoloso, se questo oggetto viene gestito dal codice di un'altra persona; per evitare conflitti possiamo utilizzare un *symbol*.

Come qui:

```
//la proprietà simbolica è visibile solo al nostro codice
let isRead = Symbol("isRead");
messages[0][isRead] = true;
```

Ora anche se qualcun altro utilizza `for..in` per avere accesso a tutte le proprietà di `messages`, la nostra proprietà sarà segreta.

Although symbols allow to lower the probability of problems, using `WeakSet` is better from the architectural point of view. Sebbene i simboli permettano una minore probabilità di problemi, utilizzare `weakSet` è meglio da un punto di vista architetturale.

[Alla formulazione](#)

## Memorizzare le date di lettura

Per memorizzare una data possiamo utilizzare `WeakMap`:

```
let messages = [
 {text: "Hello", from: "John"},
 {text: "How goes?", from: "John"},
 {text: "See you soon", from: "Alice"}
];

let readMap = new WeakMap();

readMap.set(messages[0], new Date(2017, 1, 1));
// Oggetto di tipo Date che studieremo più avanti
```

[Alla formulazione](#)

## Object.keys, values, entries

### Sommare le proprietà

```
function sumSalaries(salaries) {

 let sum = 0;
 for (let salary of Object.values(salaries)) {
 sum += salary;
 }

 return sum; // 650
}

let salaries = {
 "John": 100,
 "Pete": 300,
 "Mary": 250
};

alert(sumSalaries(salaries)); // 650
```

O, in alternativa, possiamo ottenere la somma utilizzando `Object.values` e `reduce`:

```
// reduce itera su un array con i salari,
// li sommiamo
// e ritorniamo il risultato
```

```
function sumSalaries(salaries) {
 return Object.values(salaries).reduce((a, b) => a + b, 0) // 650
}
```

Apri la soluzione con i test in una sandbox. ↗

Alla formulazione

---

## Conteggio proprietà

```
function count(obj) {
 return Object.keys(obj).length;
}
```

Apri la soluzione con i test in una sandbox. ↗

Alla formulazione

---

## Assegnamento di destrutturazione

### Assegnamento di destrutturazione

```
let user = {
 name: "John",
 years: 30
};

let {name, years: age, isAdmin = false} = user;

alert(name); // John
alert(age); // 30
alert(isAdmin); // false
```

Alla formulazione

---

## Il salario massimo

```
function topSalary(salaries) {

 let maxSalary = 0;
 let maxName = null;

 for(const [name, salary] of Object.entries(salaries)) {
 if (maxSalary < salary) {
 maxSalary = salary;
 maxName = name;
 }
 }
}
```

```
 return maxName;
}
```

Apri la soluzione con i test in una sandbox. [↗](#)

[Alla formulazione](#)

## Date e time

### Creare un oggetto date

Il costruttore `new Date` utilizza l'ora locale di default. Quindi l'unica cosa da ricordare è che il conteggio dei mesi comincia da zero.

Quindi Febbraio è il numero 1.

Qui c'è un esempio con i numeri come componenti della data:

```
//new Date(anno, mese, data, ora, minuti, secondi, millisecondi)
let d1 = new Date(2012, 1, 20, 3, 12);
alert(d1);
```

Potremmo anche creare una data da una stringa, così:

```
//new Date(datastring)
let d2 = new Date("February 20, 2012 03:12:00");
alert(d2);
```

[Alla formulazione](#)

### Mostrare il giorno della settimana

Il metodo `date.getDay()` ritorna il numero del giorno della settimana, cominciando da Domenica.

Creiamo quindi un array con i giorni della settimana, che utilizzeremo per assegnare il numero della settimana al giorno corretto:

```
function getWeekDay(date) {
 let days = ['DOM', 'LUN', 'MAR', 'MER', 'GIO', 'VEN', 'SAB'];
 return days[date.getDay()];
}

let date = new Date(2014, 0, 3); // 3 Gen 2014
alert(getWeekDay(date)); // FR
```

[Apri la soluzione con i test in una sandbox.](#)

[Alla formulazione](#)

## Giorno della settimana Europeo

```
function getLocalDay(date) {

 let day = date.getDay();

 if (day == 0) { // weekday 0 (sunday) is 7 in european
 day = 7;
 }

 return day;
}
```

[Apri la soluzione con i test in una sandbox.](#)

[Alla formulazione](#)

## Quale giorno del mese era qualche giorno fa?

L'idea è semplice: sottrarre il numero di giorni da `date`:

```
function getDateAgo(date, days) {
 date.setDate(date.getDate() - days);
 return date.getDate();
}
```

...Ma la funzione non dovrebbe modificare l'oggetto `date`. Questo è un aspetto importante, poiché chi ci fornisce l'oggetto non si aspetta cambiamenti.

Per implementarlo correttamente dovremmo clonare l'oggetto, come nel codice seguente:

```
function getDateAgo(date, days) {
 let dateCopy = new Date(date);

 dateCopy.setDate(date.getDate() - days);
 return dateCopy.getDate();
}

let date = new Date(2015, 0, 2);

alert(getDateAgo(date, 1)); // 1, (1 Gen 2015)
alert(getDateAgo(date, 2)); // 31, (31 Dec 2014)
alert(getDateAgo(date, 365)); // 2, (2 Gen 2014)
```

[Apri la soluzione con i test in una sandbox.](#)

[Alla formulazione](#)

## Ultimo giorno del mese?

Creiamo un oggetto `date` con il mese successivo, e come giorno passiamo zero:

```
function getLastDayOfMonth(year, month) {
 let date = new Date(year, month + 1, 0);
 return date.getDate();
}

alert(getLastDayOfMonth(2012, 0)); // 31
alert(getLastDayOfMonth(2012, 1)); // 29
alert(getLastDayOfMonth(2013, 1)); // 28
```

Formalmente, le date cominciano da 1, ma tecnicamente possiamo passare qualsiasi numero, l'oggetto si aggiusterà automaticamente. Quindi quando gli passiamo 0, allora significherà “il giorno precedente al primo giorno del mese”; in altre parole: “l’ultimo giorno del mese precedente”.

```
function getLastDayOfMonth(year, month) {
 let date = new Date(year, month + 1, 0);
 return date.getDate();
}
```

[Apri la soluzione con i test in una sandbox.](#) ↗

[Alla formulazione](#)

## Quanti giorni sono passati oggi?

Per ottenere il numero di secondi, possiamo generare una data usando il giorno corrente e il tempo 00:00:00; la differenza rappresenta il tempo trascorso.

La differenza è il numero di millisecondi trascorsi dall'inizio del giorno, che dovremmo poi dividere per 1000 per ottenere i secondi:

```
function getSecondsToday() {
 let now = new Date();

 // crea un oggetto utilizzando il giorno/mese/anno corrente
 let today = new Date(now.getFullYear(), now.getMonth(), now.getDate());

 let diff = now - today; // differenza in ms
 return Math.round(diff / 1000); // converti in secondi
}

alert(getSecondsToday());
```

Una soluzione alternativa potrebbe essere quella di ottenere ore/minuti/secondi e convertirli tutti in secondi:

```

function getSecondsToday() {
 let d = new Date();
 return d.getHours() * 3600 + d.getMinutes() * 60 + d.getSeconds();
}

alert(getSecondsToday());

```

[Alla formulazione](#)

## Quanti secondi mancano a domani?

Per ottenere il numero di millisecondi mancanti al giorno successivo, possiamo sottrarre da “domani alle 00:00:00” la data attuale.

Prima generiamo l’oggetto “domani”:

```

function getSecondsToTomorrow() {
 let now = new Date();

 // data di domani
 let tomorrow = new Date(now.getFullYear(), now.getMonth(), now.getDate()+1);

 let diff = tomorrow - now; // differenza in ms
 return Math.round(diff / 1000); // converti in seconds
}

```

Soluzione alternativa:

```

function getSecondsToTomorrow() {
 let now = new Date();
 let hour = now.getHours();
 let minutes = now.getMinutes();
 let seconds = now.getSeconds();
 let totalSecondsToday = (hour * 60 + minutes) * 60 + seconds;
 let totalSecondsInADay = 86400;

 return totalSecondsInADay - totalSecondsToday;
}

```

Da notare che molti stati potrebbero sottostare a DST, quindi alcuni giorni potrebbero durare 23 ore mentre altri 25. Vorremmo trattare queste situazioni separatamente.

[Alla formulazione](#)

## Formattare la data

Per ottenere il tempo passato da `date` fino ad ora – sottraiamo le date.

```

function formatDate(date) {
 let diff = new Date() - date; // la differenza in millisecondi

 if (diff < 1000) { // less than 1 second

```

```

 return 'right now';
 }

let sec = Math.floor(diff / 1000); // converti diff in secondi

if (sec < 60) {
 return sec + ' sec. ago';
}

let min = Math.floor(diff / 60000); // converti diff in minuti
if (min < 60) {
 return min + ' min. ago';
}

// formatta la data
// aggiungi gli zero a day/month/hours/minutes
let d = date;
d = [
 '0' + d.getDate(),
 '0' + (d.getMonth() + 1),
 '' + d.getFullYear(),
 '0' + d.getHours(),
 '0' + d.getMinutes()
].map(component => component.slice(-2)); // prendi gli ultimi 2 numeri da ogni componente

// unisci i componenti in una data
return d.slice(0, 3).join('.') + ' ' + d.slice(3).join(':');
}

alert(formatDate(new Date(new Date - 1))); // "right now"

alert(formatDate(new Date(new Date - 30 * 1000))); // "30 sec. ago"

alert(formatDate(new Date(new Date - 5 * 60 * 1000))); // "5 min. ago"

// la data di ieri come: 31.12.2016 20:00
alert(formatDate(new Date(new Date - 86400 * 1000)));

```

Soluzione alternativa:

```

function formatDate(date) {
 let dayOfMonth = date.getDate();
 let month = date.getMonth() + 1;
 let year = date.getFullYear();
 let hour = date.getHours();
 let minutes = date.getMinutes();
 let diffMs = new Date() - date;
 let diffSec = Math.round(diffMs / 1000);
 let diffMin = diffSec / 60;
 let diffHour = diffMin / 60;

 // formatta
 year = year.toString().slice(-2);
 month = month < 10 ? '0' + month : month;
 dayOfMonth = dayOfMonth < 10 ? '0' + dayOfMonth : dayOfMonth;
 hour = hour < 10 ? '0' + hour : hour;
 minutes = minutes < 10 ? '0' + minutes : minutes;

 if (diffSec < 1) {

```

```
 return 'right now';
} else if (diffMin < 1) {
 return `${diffSec} sec. ago`;
} else if (diffHour < 1) {
 return `${diffMin} min. ago`;
} else {
 return `${dayOfMonth}.${month}.${year} ${hour}:${minutes}`;
}
}
```

Apri la soluzione con i test in una sandbox. [↗](#)

[Alla formulazione](#)

## Metodi JSON, toJSON

### Trasformare l'oggetto in JSON e vice versa

```
let user = {
 name: "John Smith",
 age: 35
};

let user2 = JSON.parse(JSON.stringify(user));
```

[Alla formulazione](#)

### Escludere contro-referenze

```
let room = {
 number: 23
};

let meetup = {
 title: "Conference",
 occupiedBy: [{name: "John"}, {name: "Alice"}],
 place: room
};

room.occupiedBy = meetup;
meetup.self = meetup;

alert(JSON.stringify(meetup, function replacer(key, value) {
 return (key != "" && value == meetup) ? undefined : value;
}));

/*
{
 "title": "Conference",
 "occupiedBy": [{"name": "John"}, {"name": "Alice"}],
 "place": {"number": 23}
```

```
}
```

```
*/
```

In questo caso abbiamo bisogno, oltre che del test su `key==" "`, anche di quello su `value`, per escludere il caso in cui quest'ultimo valga `meetup`.

Alla formulazione

## Ricorsione e pila

### Sommare tutti i numeri fino a quello dato

La soluzione che utilizza il ciclo:

```
function sumTo(n) {
 let sum = 0;
 for (let i = 1; i <= n; i++) {
 sum += i;
 }
 return sum;
}

alert(sumTo(100));
```

La soluzione ricorsiva:

```
function sumTo(n) {
 if (n == 1) return 1;
 return n + sumTo(n - 1);
}

alert(sumTo(100));
```

La soluzione che sfrutta la formula:  $\text{sumTo}(n) = n * (n+1) / 2$ :

```
function sumTo(n) {
 return n * (n + 1) / 2;
}

alert(sumTo(100));
```

P.S. Naturalmente, la formula risulta essere la soluzione più rapida. Arriva al risultato con solamente 3 operazioni, qualsiasi sia `n`. La matematica serve!

La soluzione che utilizza il ciclo è la seconda in termini di velocità. Sia nella soluzione ricorsiva che in quella iterativa sommiamo gli stessi numeri. La ricorsione però coinvolge un gran numero di chiamate annidate e richiede una gestione dei contesti d'esecuzione. Richiede molte più risorse, questo la rende più lenta.

P.P.S. Lo standard descrive un ottimizzazione: se la chiamata ricorsiva è l'ultima cosa che avviene nella funzione (come in `sumTo`), allora la funzione esterna non ha alcuna necessità di riprendere l'esecuzione e non c'è quindi bisogno di memorizzare il contesto d'esecuzione. In questo particolare caso `sumTo(100000)` viene risolta. Ma se il motore JavaScript non lo supporta, ci sarà un errore: "maximum stack size exceeded", che indica il raggiungimento del massimo numero di esecuzioni annidate.

[Alla formulazione](#)

## Calcolare il fattoriale

Per definizione, il fattoriale `n!` può essere riscritto come `n * (n-1)!`.

In altre parole, il risultato di `factorial(n)` può essere calcolato come `n` moltiplicato per il risultato di `factorial(n-1)`. E la chiamata per `n-1` decresce ricorsivamente, fino a `1`.

```
function factorial(n) {
 return (n != 1) ? n * factorial(n - 1) : 1;
}

alert(factorial(5)); // 120
```

La base della ricorsione è il valore `1`. Potremmo anche utilizzare `0` come caso base, non ha molta importanza, ma esegue uno step in più:

```
function factorial(n) {
 return n ? n * factorial(n - 1) : 1;
}

alert(factorial(5)); // 120
```

[Alla formulazione](#)

## Successione di Fibonacci

Proviamo come prima cosa una soluzione ricorsiva.

La successione di Fibonacci è ricorsiva per definizione:

```
function fib(n) {
 return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

alert(fib(3)); // 2
alert(fib(7)); // 13
// fib(77); // will be extremely slow!
```

...Ma per valori di  $n$  elevati risulta essere molto lenta. Ad esempio, `fib(77)` potrebbe esaurire completamente le risorse della CPU.

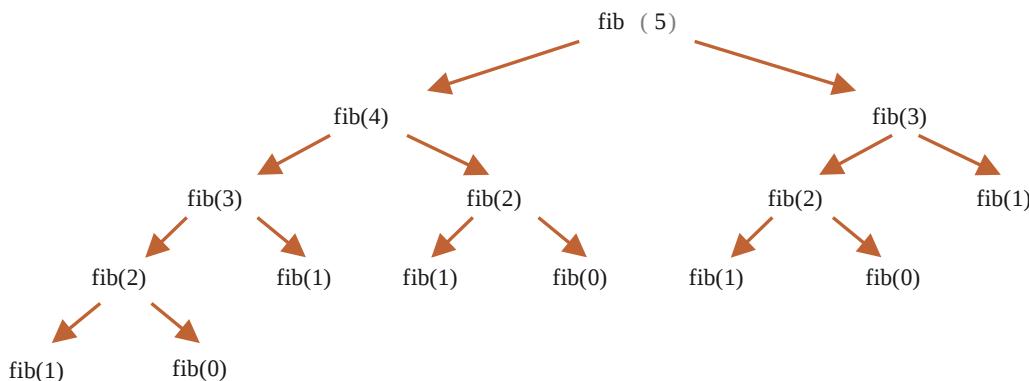
Questo accade perché la funzione esegue troppo sotto-chiamate. Gli stessi valori vengono rivalutati più volte.

Ad esempio, osserviamo una parte del calcolo di `fib(5)`:

```
...
fib(5) = fib(4) + fib(3)
fib(4) = fib(3) + fib(2)
...
```

Qui possiamo vedere che il valore di `fib(3)` è richiesto sia per `fib(5)` che per `fib(4)`. Quindi `fib(3)` verrà chiamato e valutato due volte.

L'albero di ricorsione completo:



Possiamo vedere chiaramente che `fib(3)` viene valutato due volte, mentre `fib(2)` viene valutato 3 volte. Il numero di computazioni eseguite cresce molto rapidamente, più di  $n$ , rendendo  $n=77$  un'operazione molto lenta.

Possiamo ottimizzare tenendo a mente i valori già valutati: se abbiamo già calcolato `fib(3)`, possiamo semplicemente riutilizzarlo per i calcoli successivi.

Un'altra possibilità è di utilizzare un approccio iterativo.

Piuttosto che partire da  $n$  e scendere ai valori inferiori, possiamo eseguire un ciclo partendo da 1 e 2, e proseguire per `fib(3)` come la loro somma, poi `fib(4)` come somma dei due risultati precedenti, successivamente `fib(5)` e così via, fino ad arrivare al valore desiderato. Ad ogni step sarà necessario ricordare solamente i due valori precedenti.

Vediamo quindi il nuovo algoritmo.

Inizio:

```
// a = fib(1), b = fib(2), these values are by definition 1
let a = 1, b = 1;

// get c = fib(3) as their sum
```

```

let c = a + b;

/* we now have fib(1), fib(2), fib(3)
a b c
1, 1, 2
*/

```

Vogliamo ottenere  $\text{fib}(4) = \text{fib}(2) + \text{fib}(3)$ .

Quindi scambiamo le variabili: `a, b` diventeranno `fib(2), fib(3)`, e `c` conterrà la loro somma:

```

a = b; // now a = fib(2)
b = c; // now b = fib(3)
c = a + b; // c = fib(4)

/* now we have the sequence:
 a b c
1, 1, 2, 3
*/

```

Il prossimo passo otterrà un altro numero della successione:

```

a = b; // now a = fib(3)
b = c; // now b = fib(4)
c = a + b; // c = fib(5)

/* now the sequence is (one more number):
 a b c
1, 1, 2, 3, 5
*/

```

...E così via fino al raggiungimento del numero desiderato. Questo approccio risulta essere più rapido di quello ricorsivo e si evitano duplicazioni di calcoli.

Il codice completo:

```

function fib(n) {
 let a = 1;
 let b = 1;
 for (let i = 3; i <= n; i++) {
 let c = a + b;
 a = b;
 b = c;
 }
 return b;
}

alert(fib(3)); // 2
alert(fib(7)); // 13
alert(fib(77)); // 5527939700884757

```

Il ciclo inizia con `i=3`, perché il primo e secondo valore della successione vengono memorizzati in precedenza nelle variabili `a=1, b=1`.

Questo approccio viene chiamato [programmazione dinamica bottom-up](#).

## Alla formulazione

### Stampare una single-linked list

Soluzione iterativa

La soluzione iterativa:

```
let list = {
 value: 1,
 next: {
 value: 2,
 next: {
 value: 3,
 next: {
 value: 4,
 next: null
 }
 }
 }
};

function printList(list) {
 let tmp = list;

 while (tmp) {
 alert(tmp.value);
 tmp = tmp.next;
 }
}

printList(list);
```

Da notare l'utilizzo di una variabile temporanea `tmp` per attraversare la lista.

Tecnicamente, potremmo utilizzare `list`:

```
function printList(list) {

 while(list) {
 alert(list.value);
 list = list.next;
 }
}
```

...Ma potrebbe portare ad errori. In futuro potremmo voler estendere una funzione, fare qualcosa altro con la lista. Se modifichiamo `list`, perderemmo questa capacità.

Parlando della scelta dei nomi delle variabili, `list` è la lista stessa. Il primo elemento. E dovrebbe rimanere tale.

D'altra parte, l'utilizzo di `tmp` ha esclusivamente lo scopo di attraversare la lista, come `i` nel caso di cicli `for`.

### Soluzione ricorsiva

La variante ricorsiva di `printList(list)` segue una semplice logica: per stampare una lista dovremmo stampare l'elemento corrente `list`, e fare lo stesso per `list.next`:

```
let list = {
 value: 1,
 next: {
 value: 2,
 next: {
 value: 3,
 next: {
 value: 4,
 next: null
 }
 }
 }
};

function printList(list) {
 alert(list.value); // output the current item

 if (list.next) {
 printList(list.next); // do the same for the rest of the list
 }
}

printList(list);
```

In questo caso qual'è la soluzione migliore?

Tecnicamente, la soluzione iterativa è più efficace. Queste due varianti portano allo stesso risultato, ma il ciclo non spende risorse aggiuntive per le chiamate annidate.

D'altra parte, la soluzione ricorsiva è più breve e talvolta più semplice da capire.

[Alla formulazione](#)

---

## Stampare una single-linked list in ordine inverso

### Soluzione ricorsiva

In questo caso la logica ricorsiva è un pò più complessa.

Dobbiamo prima stampare il resto della lista e *successivamente* stampare l'elemento corrente:

```

let list = {
 value: 1,
 next: {
 value: 2,
 next: {
 value: 3,
 next: {
 value: 4,
 next: null
 }
 }
 }
};

function printReverseList(list) {

 if (list.next) {
 printReverseList(list.next);
 }

 alert(list.value);
}

printReverseList(list);

```

## Soluzione iterativa

Anche la soluzione iterativa risulta essere un pò complicata.

Non abbiamo alcun modo per ottenere l'ultimo valore della nostra `list`. E comunque non potremmo "andare indietro".

Quindi quello che dobbiamo fare in questo caso è attraversare gli elementi e memorizzarli in un array, successivamente stamparli in ordine inverso:

```

let list = {
 value: 1,
 next: {
 value: 2,
 next: {
 value: 3,
 next: {
 value: 4,
 next: null
 }
 }
 }
};

function printReverseList(list) {
 let arr = [];
 let tmp = list;

 while (tmp) {
 arr.push(tmp.value);
 tmp = tmp.next;
 }
}

```

```

for (let i = arr.length - 1; i >= 0; i--) {
 alert(arr[i]);
}

printReverseList(list);

```

Da notare che la soluzione ricorsiva fa esattamente la stessa cosa: scorre la lista, memorizza gli elementi nel contesto d'esecuzione, e successivamente li stampa.

[Alla formulazione](#)

## Variable scope, closure

### Does a function pickup latest changes?

The answer is: **Pete**.

A function gets outer variables as they are now, it uses the most recent values.

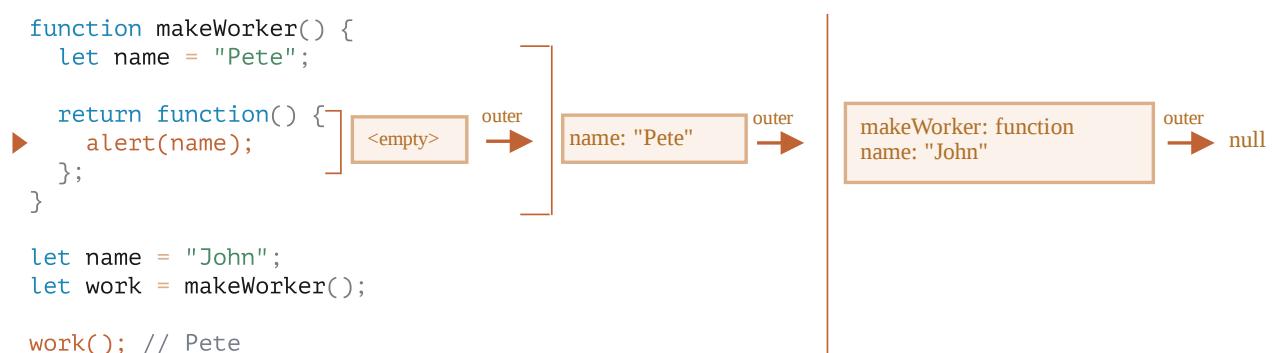
Old variable values are not saved anywhere. When a function wants a variable, it takes the current value from its own Lexical Environment or the outer one.

[Alla formulazione](#)

### Which variables are available?

The answer is: **Pete**.

The `work()` function in the code below gets `name` from the place of its origin through the outer lexical environment reference:



So, the result is "Pete" here.

But if there were no `let name` in `makeWorker()`, then the search would go outside and take the global variable as we can see from the chain above. In that case the result would be "John".

[Alla formulazione](#)

## Sono indipendenti i contatori?

La risposta corretta è: **0,1.**

Le funzioni `counter` e `counter2` vengono create da diverse invocazioni di `makeCounter`.

Quindi ognuna di esse possiede Lexical Environment indipendente, ognuno con la propria variabile `count`.

[Alla formulazione](#)

## Oggetto contatore

Ovviamente funziona.

Entrambe le funzioni annidate vengono create all'interno dello stesso Lexical Environment, quindi hanno accesso alla stessa variabile `count`:

```
function Counter() {
 let count = 0;

 this.up = function() {
 return ++count;
 };

 this.down = function() {
 return --count;
 };
}

let counter = new Counter();

alert(counter.up()); // 1
alert(counter.up()); // 2
alert(counter.down()); // 1
```

[Alla formulazione](#)

## Funzione interna ad if

Il risultato sarà **un errore**.

La funzione `sayHi` viene dichiarata internamente ad un blocco `if`, per questo è visibile solamente al suo interno. Non è accessibile esternamente.

[Alla formulazione](#)

## Somma con le closure

Perchè la seconda parentesi funzioni, la prima deve ritornare una funzione.

Come in questo esempio:

```
function sum(a) {

 return function(b) {
 return a + b; // takes "a" from the outer lexical environment
 };

}

alert(sum(1)(2)); // 3
alert(sum(5)(-1)); // 4
```

Alla formulazione

## Is variable visible?

The result is: **error**.

Try running it:

```
let x = 1;

function func() {
 console.log(x); // ReferenceError: Cannot access 'x' before initialization
 let x = 2;
}

func();
```

In this example we can observe the peculiar difference between a “non-existing” and “uninitialized” variable.

As you may have read in the article [Variable scope, closure](#), a variable starts in the “uninitialized” state from the moment when the execution enters a code block (or a function). And it stays uninitialized until the corresponding `let` statement.

In other words, a variable technically exists, but can’t be used before `let`.

The code above demonstrates it.

```
function func() {
 // the local variable x is known to the engine from the beginning of the function,
 // but "uninitialized" (unusable) until let ("dead zone")
 // hence the error

 console.log(x); // ReferenceError: Cannot access 'x' before initialization
```

```
let x = 2;
}
```

This zone of temporary unusability of a variable (from the beginning of the code block till `let`) is sometimes called the “dead zone”.

[Alla formulazione](#)

## Filter su funzioni

Filter `inBetween`

```
function inBetween(a, b) {
 return function(x) {
 return x >= a && x <= b;
 };
}

let arr = [1, 2, 3, 4, 5, 6, 7];
alert(arr.filter(inBetween(3, 6))); // 3,4,5,6
```

Filter `inArray`

```
function inArray(arr) {
 return function(x) {
 return arr.includes(x);
 };
}

let arr = [1, 2, 3, 4, 5, 6, 7];
alert(arr.filter(inArray([1, 2, 10]))); // 1,2
```

```
function inArray(arr) {
 return x => arr.includes(x);
}

function inBetween(a, b) {
 return x => (x >= a && x <= b);
}
```

[Apri la soluzione con i test in una sandbox.](#) ↗

[Alla formulazione](#)

## Ordinare per campo

```
function byField(fieldName){
 return (a, b) => a[fieldName] > b[fieldName] ? 1 : -1;
}
```

[Apri la soluzione con i test in una sandbox.](#)

[Alla formulazione](#)

## Funzione crea eserciti

Esaminiamo cosa accade dentro `makeArmy`, e la soluzione ci apparirà ovvia.

1.

Crea un array vuoto `shooters`:

```
let shooters = [];
```

2.

Lo riempie con un ciclo `shooters.push(function...)`.

Ogni elemento è una funzione, quindi l'array finale risulterà essere:

```
shooters = [
 function () { alert(i); },
 function () { alert(i); }
];
```

3.

L'array viene ritornato dalla funzione.

Successivamente, la chiamata `army[5]()` recupererà l'elemento `army[5]` dall'array (cioè una funzione) e la invocherà.

Ora perché tutte le funzioni mostrano lo stesso risultato, `10`?

Questo accade perché non c'è alcuna variabile locale `i` interna alla funzione `shooter`. Quando questa funzione viene invocata, prende `i` dal lexical environment esterno.

Quale sarà il valore di `i`?

Se guardiamo il codice:

```
function makeArmy() {
 ...
 let i = 0;
```

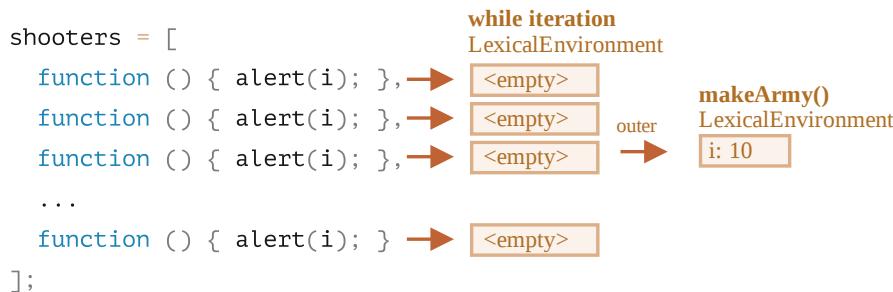
```

while (i < 10) {
 let shooter = function() { // shooter function
 alert(i); // dovrebbe mostrare il suo numero
 };
 shooters.push(shooter); // aggiunge function all'array
 i++;
}
...
}

```

Possiamo notare che la funzione `shooter` viene creata nel lexical environment di `makeArmy()`. Ma quando invochiamo `army[5]()`, `makeArmy` ha già terminato l'esecuzione, ed il valore finale di `i` è `10` (while si ferma a `i=10`).

Il risultato è che tutte le funzioni `shooter` prendono lo stesso valore dal lexical environment esterno, in cui l'ultimo valore è `i=10`.



Come puoi vedere qui sotto, a ogni iterazione del blocco `while {...}` viene creato un nuovo lexical environment. Quindi, per correggere, possiamo copiare il valore di `i` in una variabile all'interno del blocco `while {...}` stesso, così:

```

function makeArmy() {
 let shooters = [];

 let i = 0;
 while (i < 10) {
 let j = i;
 let shooter = function() { // shooter function
 alert(j); // dovrebbe mostrare il suo numero
 };
 shooters.push(shooter);
 i++;
 }

 return shooters;
}

let army = makeArmy();

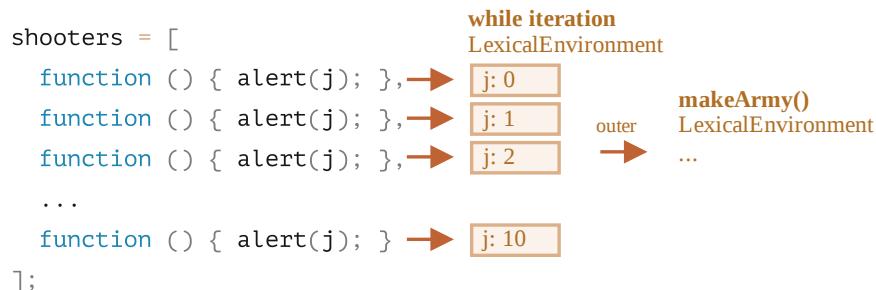
// Ora il codice funziona correttamente
army[0](); // 0
army[5](); // 5

```

Qui `let j = i` dichiara la variabile `j` "locale all'iterazione" e copia `i` al suo interno. I tipi primitivi vengono copiati "per valore", quindi otteniamo una copia indipendente di `i`,

appartenente all'iterazione corrente del ciclo.

Shooters funziona correttamente, perché il valore di `i` è un po' più vicino. Non è nel Lexical Environment di `makeArmy()`, ma nel Lexical Environment che corrisponde all'iterazione corrente del ciclo:



Questo tipo di problema può anche essere evitato usando `for` all'inizio, in questo modo:

```
function makeArmy() {
 let shooters = [];

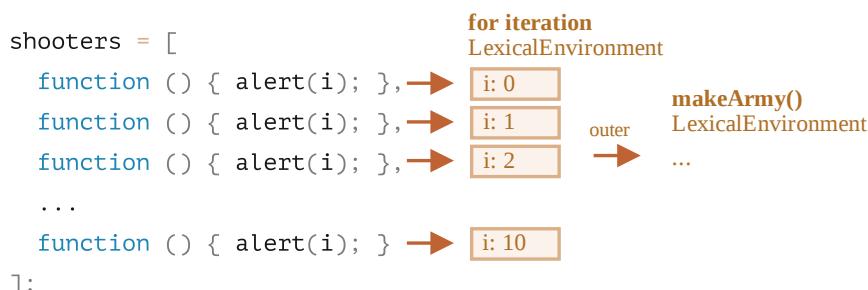
 for(let i = 0; i < 10; i++) {
 let shooter = function() { // shooter function
 alert(i); // dovrebbe mostrare il suo numero
 };
 shooters.push(shooter);
 }

 return shooters;
}

let army = makeArmy();

army[0](); // 0
army[5](); // 5
```

Essenzialmente è la stessa cosa, perché ad ogni iterazione `for` viene generato un nuovo lexical environment, con la propria variabile `i`. Quindi `shooter` generato in ogni iterazione fa riferimento alla propria `i`, in quella stessa iterazione.



Apri la soluzione con i test in una sandbox. ↗

Alla formulazione

## Oggetto funzione, NFE

### Impostare e decrementare il contatore

La soluzione utilizza `count` come variabile locale, ma i metodi aggiuntivi sono scritti dentro `counter`. Condividono lo stesso lexical environment esterno e possono accedere al valore di `count`.

```
function makeCounter() {
 let count = 0;

 function counter() {
 return count++;
 }

 counter.set = value => count = value;

 counter.decrease = () => count--;

 return counter;
}
```

Apri la soluzione con i test in una sandbox. ↗

Alla formulazione

### Sommare con un numero arbitrario di parentesi

1. Per far sì che le funzioni *comunque*, il risultato di `sum` deve essere una funzione.
2. Questa funzione deve tenere in memoria il valore corrente.
3. Come richiesto dall'esercizio, la funzione deve essere convertita in numero quando viene utilizzata con `==`. Le funzioni sono oggetti, quindi la conversione avviene come descritto nel capitolo [Conversione da oggetto a primitivi](#), e possiamo fornire un nostro metodo che si occupi di trasformare la funzione in tipo numerico.

Il codice:

```
function sum(a) {

 let currentSum = a;

 function f(b) {
 currentSum += b;
 return f;
 }

 f.toString = function() {
 return currentSum;
 };

 return f;
}
```

```
}

alert(sum(1)(2)); // 3
alert(sum(5)(-1)(2)); // 6
alert(sum(6)(-1)(-2)(-3)); // 0
alert(sum(0)(1)(2)(3)(4)(5)); // 15
```

Da notare che la funzione `sum` esegue una sola volta. Ritorna una funzione `f`.

Poi, in ogni chiamata successiva, `f` aggiunge il suo parametro alla somma presente in `currentSum`, e ritorna se stessa.

**Non c'è ricorsione nell'ultima linea di `f`.**

Una ricorsione appare in questo modo:

```
function f(b) {
 currentSum += b;
 return f(); // <-- chiamata ricorsiva
}
```

Nel nostro caso, semplicemente ritorniamo una funzione, senza effettuare alcuna chiamata:

```
function f(b) {
 currentSum += b;
 return f; // <-- non viene invocata, ritorna solamente se stessa
}
```

Questa `f` verrà utilizzata nella chiamata successiva, e ritornerà ancora se stessa, tutte le volte che sarà necessario. Quindi, quando la utilizzeremo come numero o stringa, `toString` ritorna la `currentSum`. Possiamo anche utilizzare `Symbol.toPrimitive` o `valueOf` per la conversione.

Apri la soluzione con i test in una sandbox. ↗

Alla formulazione

## Pianificazione: setTimeout e setInterval

### Output ogni secondo

Usando `setInterval`:

```
function stampaNumeri(da, a) {
 let attuale = da;

 let timerId = setInterval(function() {
 alert(attuale);
```

```

if (attuale == a) {
 clearInterval(timerId);
}
attuale++;
}, 1000);
}

// utilizzo:
stampaNumeri(5, 10);

```

Usando `setTimeout` ricorsivo:

```

function stampaNumeri(da, a) {
 let attuale = da;

 setTimeout(function vai() {
 alert(attuale);
 if (attuale < a) {
 setTimeout(vai, 1000);
 }
 attuale++;
 }, 1000);
}

// utilizzo:
stampaNumeri(5, 10);

```

Nota che in entrambe le soluzioni c'è un ritardo iniziale prima del primo output. La funzione viene eseguita la prima volta dopo `1000ms`.

Se vogliamo che la funzione venga eseguita subito, possiamo aggiungere una chiamata addizionale su di una linea separata, come questa:

```

function stampaNumeri(da, a) {
 let attuale = da;

 function vai() {
 alert(attuale);
 if (attuale == a) {
 clearInterval(timerId);
 }
 attuale++;
 }

 vai();
 let timerId = setInterval(vai, 1000);
}

stampaNumeri(5, 10);

```

[Alla formulazione](#)

---

**Cosa mostrerà `setTimeout`?**

Ogni `setTimeout` verrà eseguito solo dopo che il codice corrente è completo.

La `i` sarà l'ultimo: `1000000000`.

```
let i = 0;

setTimeout(() => alert(i), 100); // 1000000000

// ipotizza che il tempo necessario a eseguire questa funzione sia >100ms
for(let j = 0; j < 1000000000; j++) {
 i++;
}
```

[Alla formulazione](#)

## \*Decorators\* e forwarding, call/apply

### decorator spia

Il wrapper restituito da `spy (f)` dovrebbe memorizzare tutti gli argomenti e quindi usare `f.apply` per inoltrare la chiamata.

```
function spy(func) {

 function wrapper(...args) {
 // usiamo ...args invece di arguments per memorizzare un vero array in wrapper.calls
 wrapper.calls.push(args);
 return func.apply(this, args);
 }

 wrapper.calls = [];

 return wrapper;
}
```

[Apri la soluzione con i test in una sandbox.](#) ↗

[Alla formulazione](#)

### decorator ritardante

La soluzione:

```
function delay(f, ms) {

 return function() {
 setTimeout(() => f.apply(this, arguments), ms);
 };
}
```

```
let f1000 = delay(alert, 1000);

f1000("test"); // mostra "test" dopo 1000ms
```

Qui, nota come viene utilizzata un arrow function. Come sappiamo le arrow functions non hanno un proprio `this` né `arguments`, quindi `f.apply(this, arguments)` prende `this` e `arguments` dal wrapper.

Se passassimo una funzione regolare, `setTimeout` la chiamerebbe senza argomenti e `this = window` (supponendo essere in un browser).

Possiamo anche passare il `this` corretto usando una variabile intermedia, ma è un po' più complicato:

```
function delay(f, ms) {

 return function(...args) {
 let savedThis = this; // memorizzalo in una variabile intermedia
 setTimeout(function() {
 f.apply(savedThis, args); // usalo qui
 }, ms);
 };
}
```

[Apri la soluzione con i test in una sandbox.](#) ↗

[Alla formulazione](#)

## Debounce decorator

```
function debounce(func, ms) {
 let timeout;
 return function() {
 clearTimeout(timeout);
 timeout = setTimeout(() => func.apply(this, arguments), ms);
 };
}
```

Una chiamata a `debounce` restituisce un wrapper. Quando viene chiamato, pianifica la chiamata della funzione originale dopo `tot ms` e annulla il precedente timeout.

[Apri la soluzione con i test in una sandbox.](#) ↗

[Alla formulazione](#)

## Throttle decorator

```
function throttle(func, ms) {
```

```

let isThrottled = false,
 savedArgs,
 savedThis;

function wrapper() {

 if (isThrottled) { // (2)
 savedArgs = arguments;
 savedThis = this;
 return;
 }
 isThrottled = true;

 func.apply(this, arguments); // (1)

 setTimeout(function() {
 isThrottled = false; // (3)
 if (savedArgs) {
 wrapper.apply(savedThis, savedArgs);
 savedArgs = savedThis = null;
 }
 }, ms);
}

return wrapper;
}

```

La chiamata a `throttle(func, ms)` ritorna `wrapper`.

1. Durante la prima chiamata, il `wrapper` semplicemente esegue `func` ed imposta lo stato cooldown (`isThrottled = true`).
2. In questo stato, tutte le chiamate vengono memorizzate in `savedArgs/savedThis`. Va notato che sia il contesto che gli argomenti sono ugualmente importanti e dovrebbero essere memorizzati. Ne abbiamo bisogno contemporaneamente per riprodurre la chiamata.
3. Dopo che sono passati `ms` millisecondi, `setTimeout` scatta. Lo stato cooldown viene rimosso (`isThrottled = false`) e, nel caso fossero state ignorate delle chiamate, `wrapper` viene eseguito con gli ultimi argomenti e contesto memorizzati.

Il terzo passaggio non esegue `func`, ma `wrapper`, perché non abbiamo bisogno solo di eseguire `func`, ma anche di impostare nuovamente lo stato di cooldown ed il timeout per resettarlo.

[Apri la soluzione con i test in una sandbox.](#)

[Alla formulazione](#)

## Function binding

### Funzione associata come metodo

Risposta: `null`.

```
function f() {
 alert(this); // null
}

let user = {
 g: f.bind(null)
};

user.g();
```

Il contesto di una funzione associata è fisso. Non esiste alcun modo di cambiarlo successivamente.

Quindi, anche se eseguiamo `user.g()`, la funzione originale verrà chiamata con `this=null`.

[Alla formulazione](#)

---

## Secondo bind

Risposta: `John`.

```
function f() {
 alert(this.name);
}

f = f.bind({name: "John"}).bind({name: "Pete"});

f(); // John
```

L' *exotic object bound function* ↪ restituito da `f.bind(...)` memorizza il contesto (e gli argomenti, se forniti) solo in fase di creazione.

Una funzione non può essere riassegnata.

[Alla formulazione](#)

---

## Proprietà della funzione dopo il bind

Risposta: `undefined`.

Il risultato di `bind` è un altro oggetto, che non ha la proprietà `test`.

[Alla formulazione](#)

---

## Correggi una funzione che ha perso "this"

L'errore si verifica perché `askPassword` riceve le funzioni `loginOk/loginFail` senza l'oggetto.

Quando le chiamiamo, naturalmente assumono `this=undefined`.

Usiamo `bind` per associare il contesto:

```
function askPassword(ok, fail) {
 let password = prompt("Password?", '');
 if (password == "rockstar") ok();
 else fail();
}

let user = {
 name: 'John',

 loginOk() {
 alert(`${this.name} logged in`);
 },

 loginFail() {
 alert(`${this.name} failed to log in`);
 },
};

askPassword(user.loginOk.bind(user), user.loginFail.bind(user));
```

Ora funziona.

Una soluzione alternativa potrebbe essere:

```
//...
askPassword(() => user.loginOk(), () => user.loginFail());
```

Di solito anche questo funziona e appare come una buona soluzione.

Tuttavia è un po' meno affidabile in situazioni più complesse, in cui la variabile `user` potrebbe cambiare dopo la chiamata di `askPassword`, ma *prima* che il visitatore risponda e venga chiamata `( ) => user.loginOk()`.

## Alla formulazione

### Applicazione parziale per login

1.

Puoi sia utilizzare una funzione wrapper, che una arrow per essere concisi:

```
askPassword(() => user.login(true), () => user.login(false));
```

Ora riceve `user` dalla variabile esterna ed esegue la funzione in maniera corretta.

2.

Oppure creare una funzione parziale da `user.login` che utilizzi `user` come contesto ed abbia il giusto primo argomento:

```
askPassword(user.login.bind(user, true), user.login.bind(user, false));
```

[Alla formulazione](#)

## Prototypal inheritance

### Lavorare con prototype

1. `true`, preso da `rabbit`.
2. `null`, preso da `animal`.
3. `undefined`, non esiste più quella proprietà.

[Alla formulazione](#)

### Algoritmo di ricerca

1.

Aggiungiamo `__proto__`:

```
let head = {
 glasses: 1
};

let table = {
 pen: 3,
 __proto__: head
};

let bed = {
 sheet: 1,
 pillow: 2,
 __proto__: table
};

let pockets = {
 money: 2000,
 __proto__: bed
};

alert(pockets.pen); // 3
alert(bed.glasses); // 1
alert(table.money); // undefined
```

2.

Nei moderni engine, che valutano la performance, non c'è alcuna differenza tra il prelevare una proprietà dall'oggetto oppure direttamente dal suo prototype. Sono in grado di ricordare da dove è stata presa la proprietà e riutilizzarla alla prossima richiesta.

Ad esempio, per `pockets.glasses` ricordano dove hanno trovato `glasses` (in `head`), quindi la prossima volta la cercheranno proprio lì. Sono anche abbastanza intelligenti da aggiornare la cache interna nel caso qualcosa cambi, quindi questa ottimizzazione è sicura.

[Alla formulazione](#)

---

## Dove andrà a scrivere?

La risposta: `rabbit`.

Questo perché `this` fa riferimento all'oggetto prima del punto, quindi `rabbit.eat()` modifica `rabbit`.

La ricerca della proprietà e la sua esecuzione sono cose differenti.

Il metodo `rabbit.eat` viene prima cercato nel prototype, e successivamente eseguito con `this=rabbit`.

[Alla formulazione](#)

---

## Perché entrambi i criceti sono sazi?

Guardiamo attentamente cosa succede nella chiamata `speedy.eat("apple")`.

1.

Il metodo `speedy.eat` viene trovato nel prototype (`=hamster`), eseguito con `this=speedy` (l'oggetto prima del punto).

2.

Successivamente `this.stomach.push()` deve trovare la proprietà `stomach` ed invocare `push`. Cerca `stomach` in `this` (`=speedy`), ma non trova nulla.

3.

Allora segue la catena del prototype e trova `stomach` in `hamster`.

4.

Invoca `push` in `hamster`, aggiungendo il cibo nello *stomaco del prototype*.

Quindi tutti i criceti condividono un unico stomaco!

Per entrambi `lazy.stomach.push(...)` e `speedy.stomach.push()`, la proprietà `stomach` viene trovata nel prototype (poiché non si trova negli oggetti), quindi i cambiamenti avvengono lì.

Da notare che questo non accade nel caso di una semplice assegnazione `this.stomach=`:

```
let hamster = {
 stomach: [],

 eat(food) {
 // assegnamo a this.stomach invece di this.stomach.push
 this.stomach = [food];
 }
};

let speedy = {
 __proto__: hamster
};

let lazy = {
 __proto__: hamster
};

// Speedy trova il cibo
speedy.eat("apple");
alert(speedy.stomach); // apple

// lo stomaco di Lazy è vuoto
alert(lazy.stomach); // <nothing>
```

Ora tutto funziona bene, perché `this.stomach=` non deve andare alla ricerca di `stomach`. Il valore è scritto direttamente nell'oggetto `this`.

Possiamo anche evitare completamente il problema, facendo in modo che ogni criceto abbia il suo stomaco:

```
let hamster = {
 stomach: [],

 eat(food) {
 this.stomach.push(food);
 }
};

let speedy = {
 __proto__: hamster,
 stomach: []
};

let lazy = {
 __proto__: hamster,
 stomach: []
};

// Speedy trova il cibo
```

```
speedy.eat("apple");
alert(speedy.stomach); // apple

// lo stomaco di Lazy è vuoto
alert(lazy.stomach); // <nothing>
```

Come soluzione comune, tutte le proprietà che descrivono un particolare stato dell'oggetto, come `stomach`, dovrebbero essere memorizzate nell'oggetto. In questo modo eviteremo il problema.

[Alla formulazione](#)

## F.prototype

### Cambiare "prototype"

Riposte:

1.

`true`.

L'assegnazione a `Rabbit.prototype` imposta `[[Prototype]]` per i nuovi oggetti, ma non influenza gli oggetti già esistenti.

2.

`false`.

Gli oggetti vengono assegnati per riferimento. L'oggetto in `Rabbit.prototype` non viene duplicato, è sempre un oggetto riferito sia da `Rabbit.prototype` che da `[[Prototype]]` di `rabbit`.

Quindi quando cambiamo il suo contenuto tramite un riferimento, questo sarà visibile anche attraverso l'altro.

3.

`true`.

Tutte le operazioni di `delete` vengono applicate direttamente all'oggetto. Qui `delete rabbit.eats` prova a rimuovere la proprietà `eats` da `rabbit`, ma non esiste. Quindi l'operazione non avrà alcun effetto.

4.

`undefined`.

La proprietà `eats` viene rimossa da prototype, non esiste più.

## Crea un oggetto con lo stesso costruttore

Possiamo utilizzare questo approccio se siamo sicuri che il "constructor" possiede il valore corretto.

Ad esempio, se non tocchiamo il "prototype" di default, allora il codice funzionerà di sicuro:

```
function User(name) {
 this.name = name;
}

let user = new User('John');
let user2 = new user.constructor('Pete');

alert(user2.name); // Pete (ha funzionato!)
```

Ha funzionato, poiché `User.prototype.constructor == User`.

...Ma se qualcuno, per un qualsiasi motivo, sovrascrivesse `User.prototype` e dimenticasse di ricreare il `constructor` di riferimento a `User`, allora fallirebbe.

Ad esempio:

```
function User(name) {
 this.name = name;
}
User.prototype = {}; // (*)

let user = new User('John');
let user2 = new user.constructor('Pete');

alert(user2.name); // undefined
```

Perché `user2.name` è `undefined`?

Ecco come `new user.constructor('Pete')` funziona:

1. Prima, controlla se esiste `constructor` in `user`. Niente.
2. Successivamente segue la catena di prototype. Il prototype di `user` è `User.prototype`, e anche qui non c'è un `constructor` (perché ci siamo "dimenticati" di impostarlo!).
3. Seguendo la catena, `User.prototype` è un oggetto semplice, il suo prototype è `Object.prototype`.
4. Infine, per `Object.prototype`, c'è `Object.prototype.constructor == Object`. Quindi verrà utilizzato.

In conclusione, abbiamo `let user2 = new Object('Pete')`.

Probabilmente, non è quello che avremmo voluto, ossia creare `new User`, non `new Object`. Questo è il risultato del `costruttore` mancante.

(Nel caso tu sia curioso, la chiamata `new Object(...)` converte il suo argomento in un oggetto. Questa è una cosa teorica, in pratica nessuno chiama `new Object` con un valore, e generalmente non usiamo mai `new Object` per creare oggetti.)

[Alla formulazione](#)

## Native prototypes

### Aggiungi il metodo "f.defer(ms)" alle funzioni

```
Function.prototype.defer = function(ms) {
 setTimeout(this, ms);
}

function f() {
 alert("Hello!");
}

f.defer(1000); // mostra "Hello!" dopo 1 secondo
```

[Alla formulazione](#)

### Aggiungi il decorator "defer()" alle funzioni

```
Function.prototype.defer = function(ms) {
 let f = this;
 return function(...args) {
 setTimeout(() => f.apply(this, args), ms);
 }
};

// controlla
function f(a, b) {
 alert(a + b);
}

f.defer(1000)(1, 2); // mostra 3 dopo 1 secondo
```

Da notare: utilizziamo `this` in `f.apply` per far sì che il nostro decorator funzioni con i metodi degli oggetti.

Quindi se la nostra funzione viene invocata come metodo di un oggetto, allora `this` viene passato al metodo originale `f`.

```
Function.prototype.defer = function(ms) {
 let f = this;
```

```

 return function(...args) {
 setTimeout(() => f.apply(this, args), ms);
 }
};

let user = {
 name: "John",
 sayHi() {
 alert(this.name);
 }
}

user.sayHi = user.sayHi.defer(1000);

user.sayHi();

```

Alla formulazione

## Metodi di prototype, objects senza \_\_proto\_\_

### Aggiungi `toString` al dizionario

Il metodo `Object.keys` mostra tutte le proprietà enumerabili di un oggetto.

Per rendere `toString` non-enumerabile, dobbiamo definirlo utilizzando un property descriptor. La sintassi che ci permette di farlo è `Object.create`, che ci consente di fornire dei property descriptors come secondo argomento.

```

let dictionary = Object.create(null, {
 toString: { // definiamo la proprietà toString
 value() { // il valore è una funzione
 return Object.keys(this).join();
 }
 }
});

dictionary.apple = "Apple";
dictionary.__proto__ = "test";

// apple e __proto__ appaiono nel ciclo
for(let key in dictionary) {
 alert(key); // "apple", poi "__proto__"
}

// vengono elencate le proprietà separate da una virgola
alert(dictionary); // "apple,__proto__"

```

Possiamo creare una proprietà utilizzando un descriptor. Di default i flag vengono impostati a `false`. Quindi nel codice sopra, `dictionary.toString` è non-enumerabile.

Vedi il capitolo [property descriptors](#) se hai bisogno di ripassare l'argomento.

[Alla formulazione](#)

## La differenza tra chiamate

La prima chiamata ha `this == rabbit`, le altre hanno `this` uguale a `Rabbit.prototype`, perché è l'oggetto prima del punto.

Quindi, solamente la prima chiamata mostra `Rabbit`, le altre mostrano `undefined`:

```
function Rabbit(name) {
 this.name = name;
}
Rabbit.prototype.sayHi = function() {
 alert(this.name);
}

let rabbit = new Rabbit("Rabbit");

rabbit.sayHi(); // Rabbit
Rabbit.prototype.sayHi(); // undefined
Object.getPrototypeOf(rabbit).sayHi(); // undefined
rabbit.__proto__.sayHi(); // undefined
```

[Alla formulazione](#)

## Sintassi base delle classi

### Rewrite to class

```
class Clock {
 constructor({ template }) {
 this.template = template;
 }

 render() {
 let date = new Date();

 let hours = date.getHours();
 if (hours < 10) hours = '0' + hours;

 let mins = date.getMinutes();
 if (mins < 10) mins = '0' + mins;

 let secs = date.getSeconds();
 if (secs < 10) secs = '0' + secs;

 let output = this.template
 .replace('h', hours)
 .replace('m', mins)
 .replace('s', secs);

 console.log(output);
 }
}
```

```

 }

 stop() {
 clearInterval(this.timer);
 }

 start() {
 this.render();
 this.timer = setInterval(() => this.render(), 1000);
 }
 }

let clock = new Clock({template: 'h:m:s'});
clock.start();

```

Apri la soluzione in una sandbox. ↗

Alla formulazione

## Ereditarietà delle classi

### Error creating an instance

That's because the child constructor must call `super()`.

Here's the corrected code:

```

class Animal {

 constructor(name) {
 this.name = name;
 }
}

class Rabbit extends Animal {
 constructor(name) {
 super(name);
 this.created = Date.now();
 }
}

let rabbit = new Rabbit("White Rabbit"); // ok now
alert(rabbit.name); // White Rabbit

```

Alla formulazione

### Extended clock

```

class ExtendedClock extends Clock {

```

```
constructor(options) {
 super(options);
 let { precision = 1000 } = options;
 this.precision = precision;
}

start() {
 this.render();
 this.timer = setInterval(() => this.render(), this.precision);
}
};
```

Apri la soluzione in una sandbox. ↗

Alla formulazione

## Proprietà e metodi statici

### Class extends Object?

Come prima cosa, cerchiamo di capire perché il codice non funziona.

La motivazione appare piuttosto ovvia se proviamo ad eseguire il codice. Un classe che eredita, deve invocare `super()`. Diversamente, il valore di `"this"` non sarà "definito".

Vediamo come sistemarlo:

```
class Rabbit extends Object {
 constructor(name) {
 super(); // dobbiamo chiamare il costruttore padre della classe da cui stiamo ereditando
 this.name = name;
 }
}

let rabbit = new Rabbit("Rab");

alert(rabbit.hasOwnProperty('name')); // true
```

Ma non è tutto.

Anche dopo questo fix, c'è ancora un grande differenza tra `"class Rabbit extends Object"` e `class Rabbit`.

Come già sappiamo, la sintassi "extends" imposta due prototype:

1. Tra `"prototype"` del costruttore (per i metodi).
2. Tra i costruttori stessi (per i metodi statici).

Nel nostro caso, `class Rabbit extends Object` significa:

```
class Rabbit extends Object {}
```

```
alert(Rabbit.prototype.__proto__ === Object.prototype); // (1) true
alert(Rabbit.__proto__ === Object); // (2) true
```

In questo modo, tramite `Rabbit` abbiamo accesso ai metodi statici di `Object`, come nell'esempio:

```
class Rabbit extends Object {}

// normalmente invochiamo Object.getOwnPropertyNames
alert (Rabbit.getOwnPropertyNames({a: 1, b: 2})); // a,b
```

Ma se non estendiamo l'oggetto, con `extends Object`, allora `Rabbit.__proto__` non sarà impostato a `Object`.

Qui una demo:

```
class Rabbit {}

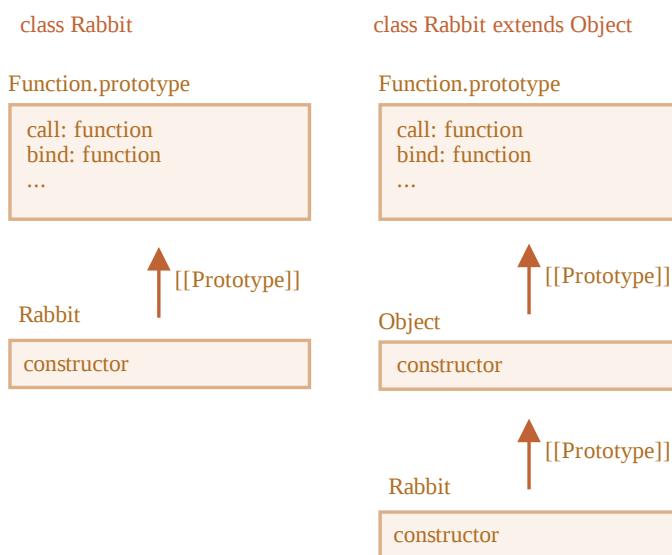
alert(Rabbit.prototype.__proto__ === Object.prototype); // (1) true
alert(Rabbit.__proto__ === Object); // (2) false (!)
alert(Rabbit.__proto__ === Function.prototype); // come qualsiasi funzione di default

// errore, funzione non esistente in Rabbit
alert (Rabbit.getOwnPropertyNames({a: 1, b: 2})); // Error
```

Quindi `Rabbit`, in questo caso, non fornisce l'accesso ai metodi statici di `Object`.

In ogni caso, `Function.prototype` possiede metodi "generici", come `call`, `bind` etc. Questi saranno disponibili in entrambi i casi, grazie al costruttore di `Object`, `Object.__proto__ === Function.prototype`.

Come mostrato in figura:



Quindi, per riassumere, ci sono due principali differenze:

<b>class Rabbit</b>	<b>class Rabbit extends Object</b>
-	dobbiamo invocare <code>super()</code> nel costruttore
<code>Rabbit.__proto__ === Function.prototype</code>	<code>Rabbit.__proto__ === Object</code>

[Alla formulazione](#)

## Verifica delle classi: "instanceof"

### Uno strano instanceof

Sì, sembra strano.

Ma `instanceof` non prende in considerazione la funzione, ma piuttosto il suo `prototype`, che trova riscontro nella catena dei prototye.

In questo caso `a.__proto__ == B.prototype`, quindi `instanceof` ritorna `true`.

Quindi, secondo la logica di `instanceof`, è il `prototype` a definire il tipo, non il costruttore.

[Alla formulazione](#)

## Gestione degli errori, "try...catch"

### Finally o solamente il codice?

La differenza diventa ovvia quando inseriamo il codice all'interno di una funzione.

Il comportamento è diverso se c'è una "uscita anticipata" dal `try...catch`.

Per esempio, quando c'è un `return` all'interno del `try...catch`. La clausola `finally` funziona *qualunque* sia la causa dell'uscita dal `try...catch`, anche tramite l'istruzione "return": appena il `try...catch` è terminato, ma prima che il codice richiamato prenda il controllo.

```
function f() {
 try {
 alert('start');
 return "result";
 } catch (err) {
 /**
 */
 } finally {
 alert('cleanup!');
 }
}
```

```
f(); // cleanup!
```

...O quando si presenta un `throw`, come:

```
function f() {
 try {
 alert('start');
 throw new Error("an error");
 } catch (err) {
 // ...
 if("can't handle the error") {
 throw err;
 }
 } finally {
 alert('cleanup!')
 }
}

f(); // cleanup!
```

È `finally` che garantisce la pulizia qui. Se inseriamo del codice alla fine di `f`, in queste situazioni, non verrà eseguito.

[Alla formulazione](#)

## Errori personalizzati, estendere la classe Error

### Eredita da SyntaxError

```
class FormatError extends SyntaxError {
 constructor(message) {
 super(message);
 this.name = this.constructor.name;
 }
}

let err = new FormatError("errore di formattazione");

alert(err.message); // errore di formattazione
alert(err.name); // FormatError
alert(err.stack); // stack

alert(err instanceof SyntaxError); // vero
```

[Alla formulazione](#)

## Promise

## Ri-risolvere (re-resolve) una promise?

L'output è: '1'.

La seconda chiamata a 'resolve' è ignorata, perché solo la prima chiamata a `reject/resolve` viene presa in considerazione. Le chiamate successive sono ignorate.

Alla formulazione

## Delay with a promise

```
function delay(ms) {
 return new Promise(resolve => setTimeout(resolve, ms));
}

delay(3000).then(() => alert('Viene eseguita dopo 3 secondi'));
```

È da notare che in questo task, `resolve` è chiamato senza argomenti. Non ritorniamo alcun valore da `delay`, ci assicuriamo solo del ritardo.

Alla formulazione

## Animated circle with promise

Apri la soluzione in una sandbox. ↗

Alla formulazione

## Concatenamento di promise (promise chaining)

### Promise: then versus catch

La risposta breve è: **no, non sono uguali**:

La differenza è che se un errore accade in `f1`, allora è gestito da `.catch` qui:

```
promise
 .then(f1)
 .catch(f2);
```

...Ma non qui:

```
promise
 .then(f1, f2);
```

Questo perché un errore è passato giù nella catena, e nel secondo pezzo di codice non c'è una catena sotto `f1`.

In altre parole, `.then` passa i risultati/erri al prossimo `.then/catch`. Così nel primo esempio, c'è un `catch` sotto, e nel secondo – non c'è, così l'errore non è gestito.

[Alla formulazione](#)

## Gestione degli errori con le promise

### Error in setTimeout

La risposta è: **no, non sarà eseguito**:

```
new Promise(function(resolve, reject) {
 setTimeout(() => {
 throw new Error("Whoops!");
 }, 1000);
}).catch(alert);
```

Come detto nel capitolo, c'è un "try..catch implicito" attorno al codice della funzione. In questo modo tutti gli errori sincroni sono gestiti.

Tuttavia qui l'errore è generato non mentre sta venendo eseguito l'esecutore, ma dopo. Per questo motivo la promise non può gestirlo.

[Alla formulazione](#)

## Async/await

### Rewrite using async/await

Le note sono sotto il codice:

```
async function loadJson(url) { // (1)
 let response = await fetch(url); // (2)

 if (response.status == 200) {
 let json = await response.json(); // (3)
 return json;
 }

 throw new Error(response.status);
}

loadJson('no-such-user.json')
 .catch(alert); // Error: 404 (4)
```

Note:

1.

La funzione `loadJson` diventa `async`.

2.

Tutti i `.then` interni sono sostituiti con `await`.

3.

Possiamo ritornare `response.json()` invece di aspettarlo (awaiting for it), come qui:

```
if (response.status == 200) {
 return response.json(); // (3)
}
```

Poi il codice esterno avrebbe dovuto attendere (`await`) che la promise risolvesse. Nel nostro caso non è importante.

4.

L'errore sollevato da `loadJson` è gestito da `.catch`. Non possiamo usare `await loadJson(...)` qui, perchè non siamo in una funzione `async`.

[Alla formulazione](#)

## Riscrivere "rethrow" con `async/await`

Non ci sono trucchi qui. Basta sostituire `.catch` with `try...catch` dentro `demoGithubUser` e aggiungere `async/await` quando necessario:

```
class HttpError extends Error {
 constructor(response) {
 super(`${response.status} for ${response.url}`);
 this.name = 'HttpError';
 this.response = response;
 }
}

async function loadJson(url) {
 let response = await fetch(url);
 if (response.status == 200) {
 return response.json();
 } else {
 throw new HttpError(response);
 }
}

// Ask for a user name until github returns a valid user
async function demoGithubUser() {
```

```

let user;
while(true) {
 let name = prompt("Enter a name?", "iliakan");

 try {
 user = await loadJson(`https://api.github.com/users/${name}`);
 break; // no error, exit loop
 } catch(err) {
 if (err instanceof HttpError && err.response.status == 404) {
 // loop continues after the alert
 alert("No such user, please reenter.");
 } else {
 // unknown error, rethrow
 throw err;
 }
 }
}

alert(`Full name: ${user.name}.`);
return user;
}

demoGithubUser();

```

[Alla formulazione](#)

## Call async from non-async

That's the case when knowing how it works inside is helpful.

Just treat `async` call as promise and attach `.then` to it:

```

async function wait() {
 await new Promise(resolve => setTimeout(resolve, 1000));

 return 10;
}

function f() {
 // shows 10 after 1 second
 wait().then(result => alert(result));
}

f();

```

[Alla formulazione](#)

## I generatori

### Pseudo-random generator

```

function* pseudoRandom(seed) {
 let value = seed;

 while(true) {
 value = value * 16807 % 2147483647
 yield value;
 }
};

let generator = pseudoRandom(1);

alert(generator.next().value); // 16807
alert(generator.next().value); // 282475249
alert(generator.next().value); // 1622650073

```

Please note, the same can be done with a regular function, like this:

```

function pseudoRandom(seed) {
 let value = seed;

 return function() {
 value = value * 16807 % 2147483647;
 return value;
 }
}

let generator = pseudoRandom(1);

alert(generator()); // 16807
alert(generator()); // 282475249
alert(generator()); // 1622650073

```

That also works. But then we lose ability to iterate with `for .. of` and to use generator composition, that may be useful elsewhere.

[Apri la soluzione con i test in una sandbox.](#) ↗

[Alla formulazione](#)

## Proxy e Reflect

### Errore in lettura di una proprietà non esistente

```

let user = {
 name: "John"
};

function wrap(target) {
 return new Proxy(target, {
 get(target, prop, receiver) {
 if (prop in target) {

```

```

 return Reflect.get(target, prop, receiver);
 } else {
 throw new ReferenceError(`Property doesn't exist: "${prop}"`)
 }
}
});

user = wrap(user);

alert(user.name); // John
alert(user.age); // ReferenceError: Property doesn't exist: "age"

```

[Alla formulazione](#)

## Accesso ad un array[-1]

```

let array = [1, 2, 3];

array = new Proxy(array, {
 get(target, prop, receiver) {
 if (prop < 0) {
 // anche se vi accediamo come arr[1]
 // prop è una stringa, quindi dobbiamo convertirla a number
 prop = +prop + target.length;
 }
 return Reflect.get(target, prop, receiver);
 }
});

alert(array[-1]); // 3
alert(array[-2]); // 2

```

[Alla formulazione](#)

## Observable

La soluzione consiste di due parti:

1. Quando `.observe(handler)` viene invocato, dobbiamo memorizzare l'handler da qualche parte, per poter essere in grado di invocarlo più tardi. Possiamo memorizzare gli handler nell'oggetto, utilizzando un symbol come chiave della proprietà.
2. Abbiamo bisogno di un proxy con la trappola `set` per poter invocare gli handlers in caso di cambiamenti.

```

let handlers = Symbol('handlers');

function makeObservable(target) {
 // 1. Inizializziamo lo store per gli handlers
 target[handlers] = [];
}

```

```

// Memorizziamo l'handler nell'array per poterlo invocare successivamente
target.observe = function(handler) {
 this[handlers].push(handler);
};

// 2. Creiamo un proxy per gestire le modifiche
return new Proxy(target, {
 set(target, property, value, receiver) {
 let success = Reflect.set(...arguments); // inoltriamo l'operazione all'oggetto
 if (success) { // se non è stato generato alcun errore durante il cambiamento della proprietà
 // invochiamo tutti gli handlers
 target[handlers].forEach(handler => handler(property, value));
 }
 return success;
 }
});

let user = {};

user = makeObservable(user);

user.observe((key, value) => {
 alert(`SET ${key}=${value}`);
});

user.name = "John";

```

[Alla formulazione](#)

## Eval: eseguire una stringa di codice

### Eval-calculator

Utilizziamo `eval` per risolvere l'espressione matematica:

```

let expr = prompt("Type an arithmetic expression?", '2*3+2');

alert(eval(expr));

```

L'utente può inserire qualsiasi testo o codice.

Per rendere il tutto più sicuro e consentire solo caratteri aritmetici, possiamo verificare `expr` utilizzando un'[espressione regolare](#), in questo modo la stringa potrà contenere solamente cifre e operatori.

[Alla formulazione](#)

## Il tipo Reference

## Controllo di sintassi

**Errore!**

Provatelo:

```
let user = {
 name: "John",
 go: function() { alert(this.name) }
}

(user.go)() // error!
```

La maggior parte dei browser non vi darà informazioni necessarie per capire cosa è andato storto.

**L'errore viene causato dalla mancanza di un punto e virgola dopo `user = { ... }`.**

JavaScript non inserisce automaticamente un punto e virgola prima di `(user.go)()`, quindi leggerà il codice in questo modo:

```
let user = { go:... }(user.go)()
```

Possiamo anche vedere questa come una comune espressione, è sintatticamente una chiamata all'oggetto `{ go: ... }` come una funzione con argomento `(user.go)`. E questo avviene nella stessa riga di `let user`, quindi l'oggetto `user` non è ancora stato definito, quindi c'è un errore.

Se inseriamo un punto e virgola, tutto funziona correttamente:

```
let user = {
 name: "John",
 go: function() { alert(this.name) }
};

(user.go)() // John
```

Da notare che le parentesi su `(user.go)` non fanno nulla. Solitamente servono ad organizzare l'ordine delle operazioni, in questo caso è presente un `.` che verrebbe comunque eseguito per primo, non hanno quindi alcun effetto. L'unico errore stava nel punto e virgola.

[Alla formulazione](#)

## Spiegate il valore di "this"

Vediamo la spiegazione.

1.

Questa è una normale chiamata ad un metodo dell'oggetto.

2.

Stessa cosa, le parentesi non cambiano l'ordine delle operazioni, il punto viene eseguito per primo in ogni caso.

3.

Qui abbiamo una chiamata più complessa `(expression).method()`. La chiamata viene interpretata come fosse divisa in due righe:

```
f = obj.go; // calculate the expression
f(); // call what we have
```

Qui `f()` viene eseguita come una funzione, senza `this`.

4.

Molto simile a `(3)`, alla sinistra del punto `.` abbiamo un'espressione.

Per spiegare il comportamento di `(3)` e `(4)` dobbiamo ricordare che la proprietà di accesso (il punto o le parentesi quadre) ritornano un valore di tipo riferimento.

Qualsiasi operazione tranne la chiamata ad un metodo (come l'assegnazione `=` o `||`) trasforma questo riferimento in un valore ordinario, che non porta più le informazioni necessarie per impostare `this`.

[Alla formulazione](#)