

# JAVASCRIPT

Guida completa  
per lo sviluppatore

Marijn Haverbeke



**HOEPLI**  
INFORMATICA

# JAVASCRIPT

**Marijn Haverbeke**

# **JAVASCRIPT**

**Guida completa  
per lo sviluppatore**



**EDITORE ULRICO HOEPLI MILANO**

Titolo originale: *Eloquent JavaScript*, 2<sup>nd</sup> edition

Copyright © 2015 by Marijn Haverbeke

All rights reserved

Originally published by No Starch Press

Per l'edizione italiana

## **Copyright © Ulrico Hoepli Editore S.p.A. 2016**

via Hoepli 5, 20121 Milano (Italy)

tel. +39 02 864871 – fax +39 02 8052886

e-mail [hoepli@hoepli.it](mailto:hoepli@hoepli.it)

**[www.hoepli.it](http://www.hoepli.it)**

Seguici su Twitter: @Hoepli\_1870

Tutti i diritti sono riservati a norma di legge  
e a norma delle convenzioni internazionali

## **ISBN EBOOK 978-88-203-7341-2**

Progetto editoriale e realizzazione: Maurizio Vedovati - Servizi editoriali ([info@iltrio.it](mailto:info@iltrio.it))

Traduzione: Marina Tadiello

Redazione: Marinella Luft

Impaginazione e copertina: Sara Taglialegne

Realizzazione digitale: Promedia, Torino

Per Lotte e Jan

## [Introduzione](#)

---

[La programmazione](#)

[L'importanza del linguaggio](#)

[Che cos'è JavaScript](#)

[Il codice e le sue meraviglie](#)

[Struttura del libro](#)

[Convenzioni tipografiche](#)

## [PARTE I - LINGUAGGIO](#)

---

### [Capitolo 1. Valori, tipi e operatori](#)

---

[Valori](#)

[Numeri](#)

[Aritmetica](#)

[Numeri speciali](#)

[Stringhe](#)

[Operatori unari](#)

[Valori booleani](#)

[Comparazioni](#)

[Operatori logici](#)

[Valori indefiniti](#)

[Conversione automatica dei tipi](#)

[Il corto circuito degli operatori logici](#)

[Riepilogo](#)

### [Capitolo 2. Struttura dei programmi](#)

---

[Espressioni e dichiarazioni](#)

[Variabili](#)

[Parole chiave e parole riservate](#)

[L'ambiente](#)

[Funzioni](#)

[La funzione console.log](#)

[Valori restituiti](#)

**Prompt e confirm**

**Struttura di controllo**

**Esecuzione condizionale**

**Cicli while e do**

**I rientri del codice**

**Cicli for**

**Uscire da un ciclo**

**Aggiornamento rapido delle variabili**

**Scelta di un valore con switch**

**Capitalizzazione**

**Commenti**

**Riepilogo**

**Esercizi**

Un ciclo per un triangolo

FizzBuzz

Scacchiera

## **Capitolo 3. Funzioni**

---

**Definire una funzione**

**Parametri e ambiti di visibilità**

**Ambiti di visibilità (scope) nidificati**

**Funzioni come valori**

**Notazione della dichiarazione**

**Pila delle chiamate**

**Argomenti facoltativi**

**Chiusura**

**Ricorsività**

**Funzioni che crescono**

**Funzioni ed effetti collaterali**

**Riepilogo**

**Esercizi**

Minimo

Ricorsività

## Capitolo 4. Strutture dati: oggetti e array

---

**Lo scoiattolo mannaro**

**Insiemi di dati**

**Proprietà**

**Metodi**

**Oggetti**

**Mutabilità**

**Il diario dello scoiattolo mannaro**

**Calcolo delle correlazioni**

**Gli oggetti come mappe**

**Analisi finale**

**Ancora a proposito di array**

**Le stringhe e le loro proprietà**

**L'oggetto arguments**

**L'oggetto Math**

**L'oggetto global**

**Riepilogo**

**Esercizi**

Somma di un intervallo

Invertire l'ordine di un array

Una lista

Confronto profondo

## Capitolo 5. Funzioni di ordine superiore

---

**Astrazione**

**Astrazioni per passare al setaccio gli array**

**Funzioni di ordine superiore**

**Passare gli argomenti**

**JSON**

**Filtrare gli array**

**Trasformazioni con map**

**Le somme con reduce**

## **Componibilità**

### **Il costo**

**Bis-bis-bis-bis...**

### **Il metodo bind**

### **Riepilogo**

### **Esercizi**

Appiattire degli array

Differenza di età tra madri e figli

Speranza di vita storica

Tutti e qualcuno

---

## **Capitolo 6. La vita segreta degli oggetti**

### **Storia**

### **Metodi**

### **Prototipi**

### **Costruttori**

### **Prevalenza sulle proprietà derivate**

### **Interferenza dei prototipi**

### **Oggetti senza prototipo**

### **Polimorfismo**

### **Impostare una tabella**

### **Recuperare e impostare**

### **Ereditarietà**

### **L'operatore instanceof**

### **Riepilogo**

### **Esercizi**

Un tipo vettore

Un'altra cella

Interfaccia per sequenze

---

## **Capitolo 7. Progetto: Vita elettronica**

### **Definizione**

### **Rappresentazione dello spazio**

### **L'interfaccia di programmazione delle creature**

### **L'oggetto World**

**this e la sua visibilità**

**Animare la vita**

**Si muove**

**Altre forme di vita**

**Una simulazione più simile alla vita**

**Metodi di gestione delle azioni**

**Popolare il nuovo mondo**

**Far vivere il mondo**

**Esercizi**

Stupidità artificiale

Predatori

## **Capitolo 8. Bachi e gestione degli errori**

---

**Errori di programmazione**

**Modalità strict**

**Prove di esecuzione**

**Debugging**

**Propagazione degli errori**

**Eccezioni**

**Ripulire dopo le eccezioni**

**Catch selettivo**

**Asserzioni**

**Riepilogo**

**Esercizi**

Riprova

La scatola chiusa

## **Capitolo 9. Espressioni regolari**

---

**Impostare un'espressione regolare**

**Prove di corrispondenza**

**Trovare serie di caratteri**

**Ripetere parte di una sequenza**

**Raggruppare le sottoespressioni**

**Corrispondenze e gruppi**

**Il tipo data**

**Confini di parola e di stringa**

**Sequenze alternative**

**La meccanica delle corrispondenze**

**Backtracking**

**Il metodo replace**

**Operatori greedy**

**Creare dinamicamente oggetti RegExp**

**Il metodo search**

**La proprietà lastIndex**

Passare in ciclo i risultati

**Analizzare un file INI**

**Caratteri internazionali**

**Riepilogo**

**Esercizi**

Il golf

Stile delle citazioni

Ancora numeri

## **Capitolo 10. Moduli**

---

**Perché i moduli aiutano**

Lo spazio dei nomi

Riutilizzo

Sdoppiamento

**Le funzioni come spazi dei nomi**

**Oggetti come interfacce**

**Staccarsi dall'ambito globale**

**Elaborare i dati come codice**

**La funzione require**

**Moduli lenti da caricare**

**Progettazione delle interfacce**

Prevedibilità

Componibilità

Interfacce stratificate

## Riepilogo

### Esercizi

Nomi dei mesi

Ritorno alla vita elettronica

Dipendenze circolari

## Capitolo 11. Progetto: Un linguaggio di programmazione

---

### L'analisi

### L'interprete

### Modelli speciali

### L'ambiente

### Funzioni

### Compilazione

### Imbrogli

### Esercizi

Array

Chiusura

Commenti

Sistemare l'ambito di visibilità

## **PARTE II - IL BROWSER**

---

## Capitolo 12. JavaScript e il browser

---

### Le reti e Internet

### Il Web

### HTML

### HTML e JavaScript

### Nell'ambiente di prova

### Compatibilità e guerre tra browser

## Capitolo 13. Il modello a oggetti del documento: DOM

---

### Struttura dei documenti

### Alberi

### Lo standard

### Spostarsi lungo l'albero

### Trovare gli elementi

**Modificare il documento**

**Creare i nodi**

**Attributi**

**Layout**

**Gli stili**

**Stili a cascata**

**Selettori di query**

**Posizionamento e animazione**

**Riepilogo**

**Esercizi**

Costruire una tabella

Elementi per nome di tag

Il cappello del gatto

## **Capitolo 14. Gestire gli eventi**

---

**Gestori degli eventi**

**Eventi e nodi del DOM**

**Oggetti evento**

**Propagazione**

**Azioni predefinite**

**Eventi per i tasti**

**Clic del mouse**

**Movimenti del mouse**

**Eventi di scorrimento**

**Eventi focus**

**Eventi load**

**Tempi di esecuzione degli script**

**Impostare dei contatori**

**Debouncing**

**Riepilogo**

**Esercizi**

La tastiera censurata

La scia del mouse

## Capitolo 15. Progetto: Un videogioco a piattaforme

---

**Il gioco**

**La tecnologia**

**Livelli**

**Leggere un livello**

**Gli attori**

**Il fardello dell'incapsulazione**

**Disegni**

**Movimenti e collisioni**

**Attori e azioni**

**Seguire i tasti**

**Eseguire il gioco**

**Esercizi**

Game Over

Interrompere il gioco

## Capitolo 16. Disegnare su un elemento canvas

---

**SVG**

**L'elemento canvas**

**Riempimento e spessore del bordo**

**Il tracciato**

**Curve**

**Disegnare un grafico a torta**

**Il testo**

**Immagini**

**Trasformazioni**

**Memorizzare ed eliminare le trasformazioni**

**Torniamo al gioco**

**Scegliere un'interfaccia grafica**

**Riepilogo**

**Esercizi**

Forme geometriche

Il grafico a torta  
Una palla che rimbalza  
Riflessioni precalcolate

## **Capitolo 17. HTTP**

---

**Il protocollo**

**I browser e HTTP**

**XMLHttpRequest**

**Inviare una richiesta**

**Richieste asincrone**

**Recuperare i dati XML**

**Protezioni sandbox in HTTP**

**Astrazione delle richieste**

**Promesse**

**Il valore del protocollo HTTP**

**La sicurezza e HTTPS**

**Riepilogo**

**Esercizi**

Negoziazione dei contenuti  
Aspettare più promesse

## **Capitolo 18. Moduli e campi di moduli**

---

**I campi**

**Campi attivi**

**Campi non abilitati**

**Il modulo nel suo insieme**

**Campi di testo**

**Caselle di spunta e pulsanti di opzione**

**Campi selezionabili**

**Campi file**

**Registrare i dati sul lato client**

**Riepilogo**

**Esercizi**

Un laboratorio per JavaScript

[Autocompletamento](#)

[Il Gioco della vita di Conway](#)

## **Capitolo 19. Progetto: Un programma per dipingere**

---

### **Realizzazione**

#### **Costruire la struttura DOM**

#### **Le fondamenta**

#### **Selezione degli strumenti**

#### **Colore e dimensioni del pennello**

#### **Salvare**

#### **Caricare file di immagini**

#### **Rifiniture**

#### **Esercizi**

[Rettangoli](#)

[Tavolozza per i colori](#)

[Secchiello e riempimento](#)

## **PARTE III - OLTRE**

---

## **Capitolo 20. Node.js**

---

### **Un po' di storia**

### **Asincronicità**

### **Il comando node**

### **Moduli**

#### **Installazione con NPM**

#### **Il modulo filesystem**

#### **Il modulo HTTP**

#### **Flussi di dati**

#### **Un semplice server di file**

#### **Gestione degli errori**

#### **Riepilogo**

#### **Esercizi**

[Ancora negoziazione dei contenuti](#)

[Tappare una falla](#)

[Creare directory](#)

[Uno spazio pubblico sul Web](#)

## **Capitolo 21. Progetto: Un sito Web di skill-sharing**

---

**Piano del progetto**

**Modello di comunicazione long polling**

**Interfaccia HTTP**

**Il server**

Routing

Servire i file

Presentazioni come risorse

Supporto per long polling

**Il client**

HTML

Si parte

Visualizzazione delle presentazioni

Aggiornamenti del server

Rilevamento delle modifiche

**Esercizi**

Persistenza su disco

Reimpostare i campi per i commenti

Modelli migliori

Senza script

## **Capitolo 22. JavaScript e prestazioni**

---

**Compilazione per gradi**

**Layout di un grafo**

**Definizione di un grafo**

**Una prima funzione di layout basata sulla forza**

**Analisi dinamica del programma**

**Funzioni in linea**

**Ritorno ai cicli di vecchia memoria**

**Evitare lavoro inutile**

**Creare meno rifiuti**

**La raccolta dei rifiuti**

**Scrivere sugli oggetti**

**Tipi dinamici**

**Riepilogo**

## **Esercizi**

Trovare un percorso

Misura del tempo

Ottimizzare

## **Suggerimenti per gli esercizi**

---

### **Struttura dei programmi**

Un ciclo per un triangolo

FizzBuzz

Scacchiera

### **Funzioni**

Minimo

Ricorsività

Conteggi

### **Strutture dati: oggetti e array**

La somma di un intervallo

Invertire l'ordine di un array

Liste

Confronto profondo

### **Funzioni di ordine superiore**

Differenza di età tra madri e figli

Speranza di vita storica

Tutti e qualcuno

### **La vita segreta degli oggetti**

Un tipo vettore

Un'altra cella

Interfacce per sequenze

### **Progetto: vita elettronica**

Stupidità artificiale

Predatori

### **Bachi e gestione degli errori**

Riprova

La scatola chiusa

### **Espressioni regolari**

Stile delle citazioni

Ancora numeri

### **Moduli**

Nomi dei mesi

Ritorno alla vita elettronica

Dipendenze circolari

## Progetto: un linguaggio di programmazione

Array

Chiusura

Commenti

Sistemare l'ambito di visibilità

## Il modello a oggetti del documento: DOM

Costruire una tabella

Elementi per nome di tag

## Gestire gli eventi

La tastiera censurata

La scia del mouse

Schede

## Progetto: un videogioco a piattaforme

Game Over

Interrompere il gioco

## Disegnare su un foglio

Forme geometriche

Il grafico a torta

Una palla che rimbalza

Riflessioni precalcolate

## HTTP

Negoziazione dei contenuti

Aspettare più promesse

## Moduli e relativi campi

Un laboratorio per JavaScript

Autocompletamento

Il Gioco della vita di Conway

## Progetto: Un programma per dipingere

Rettangoli

Tavolozza per i colori

Secchiello e riempimento

## Node.js

Ancora negoziazione dei contenuti

Tappare una falla

Creare directory

Uno spazio pubblico sul Web

## **Progetto: Un sito Web di skill-sharing**

Persistenza su disco

Reimpostare i campi per i commenti

Modelli migliori

Senza script

## **JavaScript e prestazioni**

Trovare un percorso

Ottimizzare

## **Indice analitico**

---

## **Circa l'autore**

---

## **Informazioni sul Libro**

---

# INTRODUZIONE

Questo libro spiega come far fare ai computer quel che vogliamo che facciano. Al giorno d'oggi, i computer sono comuni quanto un cacciavite, ma molto più complessi e quindi difficili da utilizzare e comprendere. Per molti, rimangono degli oggetti misteriosi e vagamente minacciosi.



Abbiamo trovato due valide soluzioni per colmare il baratro della comunicazione tra noi, organismi biologici, mollicci e ricchi di talento ragionativo in senso sociale e spaziale, e i computer, freddi manipolatori di dati senza significato. La prima è di fare appello al nostro senso del mondo fisico per realizzare interfacce che lo imitino e ci consentano di manipolare le forme sullo schermo con le dita: cosa che funziona molto bene per le interazioni casuali tra uomo e macchina.

Ma non abbiamo ancora trovato una soluzione efficace per utilizzare l'approccio punta-e-clicca quando si tratta di comunicare al computer cose non previste da chi aveva realizzato l'interfaccia. Per le interfacce aperte, come dare al computer istruzioni per svolgere dei compiti arbitrari, abbiamo avuto più fortuna con un approccio che sfrutta il nostro talento per il linguaggio: ossia, quello di insegnare al computer una lingua, un linguaggio di programmazione.

Il linguaggio umano permette di combinare con una certa libertà parole e frasi, consentendoci di esprimere concetti diversi. I linguaggi per i computer seguono principi simili, sebbene abbiano una grammatica meno flessibile.

Negli ultimi vent'anni si è diffuso un uso meno formale dei computer, dove le interfacce basate sul linguaggio, che erano la norma in passato, sono state in gran parte sostituite da interfacce grafiche. Ma esistono ancora: basta sapere dove cercarle. Uno di questi linguaggi, JavaScript, fa parte di quasi tutti i browser ed è pertanto disponibile su praticamente tutti i dispositivi destinati al grande pubblico.

Questo libro si pone l'obiettivo di darvi gli strumenti per imparare abbastanza di questo linguaggio da poter far fare al computer quel che volete.

## La programmazione

*Non inseguo a chi non vuole imparare e non fornisco stimoli a coloro che non sono ansiosi di dare essi stessi spiegazioni. Se ho presentato l'angolo di un quadrato e chi mi ascolta non è in grado di riportarmi gli altri tre, non ritornerò sugli stessi punti.*

Confucio

Oltre a trattare JavaScript, spiegherò anche dei concetti fondamentali di programmazione. In effetti, programmare è difficile. In generale, le regole di base sono semplici e chiare. Ma i programmi realizzati sulla base di quelle regole finiscono spesso coll'essere tanto complicati da introdurre nuove regole e nuovi livelli di difficoltà. In un certo senso, programmando costruiamo un labirinto, dove rischiamo noi stessi di perderci.

A volte, leggendo questo libro, potrete sentirvi frustrati. Se siete dei novizi, dovrete digerire un sacco di materiale sconosciuto. Mettere insieme questi materiali richiederà ulteriori sforzi di comprensione.

Decidete voi quanto impegno volete metterci. Se fate fatica a seguire, però, non saltate a conclusioni affrettate sulle vostre capacità: fermatevi e riconsiderate la situazione. Prendetevi una pausa, andate a rileggere quel che non era chiaro e verificate sempre di aver letto e compreso tutti i programmi di esempio e gli esercizi. Si fa fatica a imparare, ma quel che imparate vi appartiene e semplifica l'apprendimento di nuove nozioni.

*Il programmatore è il creatore di universi dei quali è l'unico responsabile. Si possono creare universi di complessità praticamente illimitata sotto forma di programmi per computer.*

Joseph Weizenbaum, “Il potere del computer e la ragione umana”

Un programma è tante cose. È un brano di testo digitato da un programmatore, è la forza motrice che fa agire il computer, è l'insieme dei dati nella memoria del computer e nello stesso tempo è quel che controlla le azioni svolte sulla stessa memoria. Non è facile paragonare il computer ad altri oggetti di uso quotidiano. Un paragone che potrebbe calzare, almeno superficialmente, è quello con una macchina: ci sono tante parti collegate tra loro e per farle funzionare dobbiamo stabilire le interconnessioni, il funzionamento di ciascuna parte e il modo in cui ciascuna contribuisce al risultato finale.

Un computer è una macchina costruita per ospitare delle macchine virtuali. Di per sé, i computer possono svolgere solo operazioni stupidamente semplici. Il motivo per cui sono tanto utili è che le svolgono a velocità altissime. I programmi sono in grado di combinare una quantità enorme di azioni semplici per svolgere operazioni estremamente complicate.

Per alcuni di noi, scrivere programmi per computer è un gioco appassionante. Un programma è una costruzione di pensieri. Non ha costi di materiali, non ha peso e cresce senza sforzo man mano che digitiamo sulla tastiera.

Ma se non prestiamo attenzione, le dimensioni e la complessità del programma possono sfuggire di mano e confondere anche la persona che lo ha creato. Mantenere il controllo sul programma è il problema centrale della programmazione. È meraviglioso quando un programma funziona. L'arte del programmatore sta nella sua capacità di

controllare la complessità: un programma per essere grande viene domato e semplificato.

Molti programmatore credono che la complessità si possa controllare grazie ad alcune tecniche ben note. Si affidano a regole stringenti (“best practice”) che ne dettano la forma e i più esaltati non esitano a considerare cattivi programmatore quelli che escono dalla zona di sicurezza.

Che condanna per la ricchezza del programmare, cercare di ridurlo a qualcosa di semplice e prevedibile, di mettere al bando tutti i programmi strani e bellissimi! Il panorama delle tecniche di programmazione è vastissimo e affascinante nella sua diversità e ancora in larga parte inesplorato. Se è sicuramente pericoloso per i programmatore inesperti lasciarsi incantare da troppa confusione, basta procedere con cautela e mantenere la testa sulle spalle. Come vedrete, ci sono sempre nuove sfide da affrontare e territori inviolati da scoprire. I programmatore che smettono di esplorare perdono l’ispirazione e la gioia e finiscono col disamorarsi della loro arte.

## L’importanza del linguaggio

Nel principio, alla nascita dei computer, non esistevano linguaggi di programmazione. I programmi avevano un aspetto come questo:

---

```
00110001 00000000 00000000  
00110001 00000001 00000001  
00110011 00000001 00000010  
01010001 00001011 00000010  
00100010 00000010 00001000  
01000011 00000001 00000000  
01000001 00000001 00000001  
00010000 00000010 00000000  
01100010 00000000 00000000
```

---

Questo è un programma che somma i numeri da 1 a 10 e ne stampa il risultato:  $1 + 2 + \dots + 10 = 55$ . Potrebbe girare su una ipotetica macchina molto semplice. Per programmare i primi computer, era necessario impostare grandi quadri di interruttori sulle posizioni giuste o forare delle strisce di cartone da far passare nel computer. Non è difficile immaginare quanto questa procedura fosse noiosa e soggetta a errori. Persino scrivere dei semplici programmi richiedeva grande abilità e molta disciplina. Programmi complessi non si potevano neppure immaginare!

Certo, l’inserimento manuale di strutture arcane di bit (gli uno e gli zeri) dava al programmatore la sensazione di essere un mago potente. Cosa che aveva un certo suo fascino in termini di soddisfazione professionale.

Ogni riga del programma sopra riportato contiene una sola istruzione. In italiano, il programma si potrebbe tradurre come segue:

- 
1. Memorizza il numero 0 nella posizione di memoria 0.
  2. Memorizza il numero 1 nella posizione di memoria 1.
  3. Memorizza il valore della posizione di memoria 1 nella posizione di memoria 2.
  4. Sottrai il numero 11 dal valore nella posizione di memoria 2.
  5. Se il valore nella posizione di memoria 2 è il numero 0, continua con l'istruzione 9.
  6. Aggiungi il valore della posizione di memoria 1 alla posizione di memoria 0.
  7. Aggiungi il numero 1 al valore della posizione di memoria 1.
  8. Continua con l'istruzione 3.
  9. Stampa il valore della posizione di memoria 0.
- 

Anche se più leggibile del blocco di bit, il programma non è tanto bello. Si potrebbero usare dei nomi invece dei numeri per le istruzioni e le posizioni di memoria:

---

Imposta "totale" su 0.

Imposta "somma" su 1.

[ciclo]

Imposta "confronta" su "somma".

Sottrai 11 da "confronta".

Se "confronta" è zero, continua fino a [fine].

Aggiungi "somma" a "totale".

Aggiungi 1 a "somma".

Continua per tutto il [ciclo].

[fine]

Stampa "totale".

---

Capite adesso che cosa fa il programma? Nelle prime due righe si impostano due posizioni di memoria sui valori iniziali: totale viene utilizzato per calcolare il risultato, mentre somma tiene in memoria il numero corrente. Le righe con l'istruzione confronta sono forse le più strane: per sapere se il programma può concludersi, bisogna stabilire se somma è uguale a 11. Poiché la nostra ipotetica macchina è piuttosto primitiva, può solo stabilire se un numero è zero e prendere una decisione (o fare un salto) su questa base. Usa pertanto la posizione di memoria che ha l'etichetta confronta per calcolare il valore di somma - 11 e prende una decisione in base a questo valore. Nelle due righe seguenti, si aggiunge il valore di somma al risultato e si incrementa somma di 1 ogni volta che il programma ha stabilito che somma non è ancora arrivato a 11.

Lo stesso programma in JavaScript si scrive così:

---

```
var totale = 0, somma = 1;
```

```
while (somma <= 10) {  
    totale += somma;  
    somma += 1;  
}  
console.log(totale);  
// → 55
```

---

Questa versione offre alcuni miglioramenti. In particolare, non c'è bisogno di specificare come vogliamo che il programma vada avanti e indietro: di questo si occupa il costrutto `while`, che continua a eseguire il blocco di codice tra parentesi graffe che lo segue, fintanto che viene soddisfatta la condizione data. Tale condizione è `somma <= 10`, che significa “somma è minore o uguale a 10”. Non c'è più bisogno di creare un valore temporaneo da confrontare con zero, che era un dettaglio poco significativo. Parte della forza dei linguaggi di programmazione è proprio che si occupano al posto nostro dei dettagli insignificanti.

Alla fine del programma, una volta terminato il costrutto `while`, si applica al risultato l'operazione `console.log` per estrarre come risultato finale.

Infine, ecco come potrebbe essere il programma se avessimo a disposizione due altre utili operazioni: `range` e `sum`, che creano rispettivamente una raccolta di numeri all'interno di un intervallo dato (`range`) e calcolano la somma della raccolta:

```
console.log(sum(range(1, 10)));  
// → 55
```

---

La morale della storia è che lo stesso programma può essere espresso in modi diversi: lunghi e corti, leggibili e illeggibili. La prima versione del programma era molto oscura, mentre quest'ultima è praticamente inglese, tradotta risulta come: `registra la somma dell'intervallo di numeri da 1 a 10`. Nei prossimi capitoli vedremo come realizzare operazioni come `sum` (somma) e `range` (intervallo).

Un buon linguaggio di programmazione aiuta il programmatore a descrivere le azioni che il computer deve svolgere a un livello superiore. Aiuta a evitare dettagli trascurabili, offre comode operazioni (come `while` e `console.log`), consente di definirne altre (come `sum` e `range`) e ne semplifica la realizzazione.

## Che cos'è JavaScript

JavaScript fu introdotto nel 1995 come strumento per l'inserimento di programmini nelle pagine Web del browser Netscape Navigator. Quel linguaggio è stato via via adottato da tutti gli altri browser grafici, rendendo possibili moderne applicazioni Web con le quali si può interagire direttamente, senza dover ogni volta ricaricare la pagina. Ma viene usato anche in siti più tradizionali per offrire varie forme di interattività.

Va ricordato che JavaScript ha ben poco in comune col linguaggio di programmazione chiamato Java. La somiglianza nel nome si ispirò a considerazioni di marketing più che a ragioni sensate. Quando uscì JavaScript, Java era al massimo della popolarità e qualcuno ebbe l'idea di sfruttarne il nome per avere successo. E quel nome è rimasto.

Dopo la sua adozione in altri ambiti, oltre a Netscape, venne scritto un documento standard per descrivere il funzionamento di JavaScript, in modo da uniformare il linguaggio per tutti i vari software che promettevano di offrire supporto per JavaScript. Quel documento è lo standard ECMAScript e prende il nome dall'organizzazione internazionale che si occupò della standardizzazione. In pratica, si possono usare indifferentemente i due termini, ECMAScript e JavaScript, in quanto si riferiscono allo stesso linguaggio.

C'è chi dice cose terribili a proposito di JavaScript, e molte sono vere. La prima volta che dovetti scrivere qualcosa in JavaScript, lo detestai quasi subito: accettava quasi tutto quel che scrivevo, ma lo interpretava in maniera diversa da quel che intendeva. Certo, questo dipendeva in gran parte dal fatto che non sapevo ciò che stavo facendo, ma mette in evidenza una pecca grave: JavaScript pone pochissimi limiti a quel che si può fare. L'idea di fondo era di semplificare il lavoro per i principianti. Di fatto, questa impostazione rende quasi impossibile rilevare eventuali problemi, perché il sistema non li evidenzia.

La flessibilità di JavaScript ha comunque i suoi vantaggi. Lascia spazio a molte tecniche impossibili in linguaggi più rigidi. Come vedremo per esempio nel [Capitolo 10](#), può persino aiutare a superare alcune delle lacune del linguaggio. Dopo aver imparato bene JavaScript e averci lavorato per un po' di tempo, ho imparato anche ad apprezzarlo.

Sono esistite diverse versioni di JavaScript. La versione 3 di ECMAScript era quella più supportata all'epoca dell'affermazione di JavaScript, grossomodo tra il 2000 e il 2010. In questo periodo, si lavorava anche a un'ambiziosa versione 4, per la quale erano previsti miglioramenti ed estensioni radicali. Tuttavia, modificare così drasticamente un linguaggio vivo e largamente utilizzato si rivelò troppo difficile e il lavoro sulla versione 4 venne abbandonato nel 2008. La versione 5, molto meno ambiziosa, uscì nel 2009. Attualmente, tutti i browser più importanti offrono supporto per la versione 5, che è quella su cui si basa questo libro. La versione 6 è in corso di realizzazione e alcuni browser cominciano a offrire supporto per questa nuova versione.

I browser Web non sono le sole piattaforme su cui viene usato JavaScript. Alcuni database, tra cui MongoDB e CouchDB, utilizzano JavaScript per script e query. Diverse piattaforme per la programmazione di desktop e server, in particolare il progetto Node.js (di cui si parla nel [Capitolo 20](#)) offrono un potente ambiente di programmazione per JavaScript al di fuori dell'ambito browser.

## Il codice e le sue meraviglie

Il codice è il testo che forma i programmi. Nei capitoli di questo libro ne troverete parecchio. Secondo me, leggere e scrivere codice sono componenti indispensabili per

imparare a programmare: dovrete pertanto analizzare con attenzione gli esempi, leggendoli e sforzandovi di capirli. Se inizialmente vi sentite confusi, tenete duro: vi abituerete presto. La stessa cosa vale per gli esercizi: non date per scontato che li avete capiti, finché non sarete in grado di scrivere del codice che funziona.

È bene abituarsi a provare le soluzioni agli esercizi in un interprete di JavaScript. In questo modo, avrete immediatamente un'idea dei risultati; cosa che dovrebbe servire da stimolo per continuare a sperimentare anche al di là degli esercizi.

Il modo più semplice per eseguire i listati riportati nel libro e fare esperimenti col codice è quello di accedere alla versione online del libro (in inglese) <http://eloquentjavascript.net>. Qui potete cliccare sugli esempi di codici per modificarli ed eseguirli per vederne i risultati. Per lavorare sugli esercizi andate su <http://eloquentjavascript.net/code/>, che fornisce il codice di base per ogni esercizio e vi consente di vedere le soluzioni.

Se scegliete di scaricare i listati per eseguire i programmi riportati nel libro in un ambiente diverso, dovrete fare un po' di attenzione. Molti degli esempi sono fini a se stessi e dovrebbero funzionare in qualunque ambiente JavaScript. Ma il codice degli ultimi capitoli è stato scritto per lo più per un ambiente specifico, il browser o la struttura Node.js, e funziona solo lì. Inoltre, in molti capitoli si parla di programmi più vasti e i brani di codice estratti da questi programmi hanno dipendenze tra di loro o da file esterni. Lo spazio protetto del sito elenca i collegamenti a file .zip che contengono tutti gli script e ai file di dati necessari per eseguire il codice di ogni capitolo.

Poiché i listati disponibili dal sito del libro sono in inglese, abbiamo mantenuto in inglese anche le illustrazioni e le parti di codice che si riferiscono a quei listati. In alcuni casi, abbiamo tuttavia scelto di tradurre in italiano i commenti per facilitarne la comprensione. Per gli stessi motivi, abbiamo tradotto in italiano i listati riportati come esempi di forma sintattica dei programmi.

## Struttura del libro

Il libro si compone di tre parti. Nei primi 11 capitoli si parla del linguaggio JavaScript. Gli otto capitoli successivi trattano di browser Web e dell'uso di JavaScript in quegli ambienti. Gli ultimi due capitoli sono dedicati a Node.js, un altro ambiente di programmazione per JavaScript.

Il libro contiene cinque capitoli di progetti, che descrivono programmi di esempio più vasti per dare un'idea di quel che si può fare col linguaggio. I capitoli trattano, in quest'ordine, di simulazione di vita artificiale, di un linguaggio di programmazione, di un videogioco, di un'applicazione per disegnare e di un sito Web dinamico.

La parte dedicata al linguaggio si apre con quattro capitoli introduttivi alla struttura di JavaScript. Qui si parla di strutture di controllo (come la parola `while` già citata), funzioni (l'impostazione delle proprie operazioni) e strutture dati. Con questi elementi è già possibile scrivere dei programmi semplici. Nei [Capitoli 5 e 6](#) si passa alle tecniche di

manipolazione di funzioni e oggetti, per arrivare a scrivere codice più astratto e mantenere sotto controllo la complessità dei programmi.

Dopo il primo capitolo sui progetti, la prima parte del libro continua con capitoli su gestione e risoluzione degli errori, sulle espressioni regolari (strumento importantissimo quando si lavora con dati di testo) e sulla modularità, un altro strumento di protezione dalla complessità. Il capitolo col secondo progetto conclude la prima parte del libro.

Nella seconda parte, i Capitoli dal 12 al 19 descrivono gli strumenti a disposizione di JavaScript nei browser: si spiega come visualizzare oggetti sullo schermo ([Capitoli 13 e 16](#)), come rispondere all'input degli utenti ([Capitoli 14 e 18](#)) e come comunicare sulla rete ([Capitolo 17](#)). Questa parte comprende altri due capitoli dedicati ad altrettanti progetti.

Il [Capitolo 20](#) tratta di Node.js e nel [Capitolo 21](#) si spiega come realizzare un semplice sistema Web con Node.js.

Infine, il [Capitolo 22](#) riporta alcune delle considerazioni che nascono quando si vuole ottimizzare la velocità dei programmi in JavaScript.

## Convenzioni tipografiche

Il testo in carattere a larghezza fissa indica elementi di programma; a volte saranno frammenti indipendenti, altre volte possono essere componenti o singoli elementi di un programma. I programmi, come negli esempi riportati nelle pagine precedenti, sono rappresentati come segue:

---

```
function fac(n) {  
    if (n == 0)  
        return 1;  
    else  
        return fac(n - 1) * n;  
}
```

---

A volte, l'output di un programma viene rappresentato alla fine del programma stesso, preceduto da due segni di barra e da una freccia:

---

```
console.log(fac(8));  
// → 40320
```

---

Buona fortuna!

# **Parte I**

## **LINGUAGGIO**

# 1

## VALORI, TIPI E OPERATORI

*Sotto la superficie della macchina, il programma si muove. Senza sforzo, si espande e si contrae. In grande armonia, gli elettroni si disperdon e si raggruppano. Le forme sullo schermo sono solo increspature sulla superficie dell'acqua. L'essenza rimane al di sotto, invisibile.*

Master Yuan-Ma, *The Book of Programming*

Dentro il mondo dei computer ci sono solo dati. Leggete dati, modificate dati, create nuovi dati, tutto ciò che non è dati semplicemente non esiste. Tutti questi dati sono memorizzati come lunghe sequenze di bit e sono pertanto fondamentalmente uguali.

I bit sono qualunque cosa possa avere due valori, di solito espressi come zero e uno. All'interno del computer, prendono forme diverse: per esempio, cariche elettriche ad alto o a basso voltaggio, segnali forti o deboli, punti lucidi od opachi sulla superficie di un CD. Qualunque tipo di informazione può essere ridotto a una sequenza di zero e uno e, pertanto, rappresentato in bit.

Pensate per esempio a come si potrebbe rappresentare in bit il numero 13. L'impostazione è la stessa dei numeri decimali, solo che invece di 10 cifre, ne avete solo due; e il valore di ciascuna cresce in potenza di due, da destra a sinistra. Questi sono i bit che rappresentano il numero 13, con i rispettivi valori indicati sotto ciascuno:

---

0	0	0	0	1	1	0	1
128	64	32	16	8	4	2	1

---

Questo è il numero binario 00001101, ossia  $8 + 4 + 1$ , che corrisponde a 13.

### Valori

Immaginate un mare di bit, un oceano. Il tipico computer di oggi ha più di 30 milioni di bit nella memoria volatile. La memoria non volatile (disco rigido o equivalente) è di solito maggiore di diversi ordini di grandezza.



Per lavorare con tali quantità di bit senza perdersi, bisogna dividerli in blocchi che rappresentino informazioni diverse. In un ambiente JavaScript, questi blocchi si chiamano *valori*. Sebbene tutti i valori siano costituiti da bit, i loro ruoli sono diversi: a ogni valore è assegnato un tipo, che ne determina il ruolo. In JavaScript esistono sei tipi fondamentali di valori: numeri, stringhe, valori booleani, oggetti, funzioni e valori indefiniti.

Per creare un valore, è sufficiente invocarne il nome. Comodo! Non bisogna raccogliere materiali da costruzione e non si spendono soldi: basta chiamare un valore per nome ed eccolo lì. Tuttavia, i valori non sono fatti d'aria. Ognuno di essi deve essere memorizzato da qualche parte e quando ne servono quantità gigantesche, si rischia di rimanere senza bit. Per fortuna, questo problema si pone solo se i valori devono essere disponibili tutti insieme allo stesso tempo. Appena si smette di usare un valore, i bit che occupava ritornano a disposizione, come materiali da costruzione per la prossima generazione di valori.

Questo capitolo presenta gli elementi che compongono il nucleo dei programmi in JavaScript, ossia, i tipi di valore semplici e gli operatori che li muovono.

## Numeri

I valori del tipo *numero* sono valori numerici. Nei programmi JavaScript, si scrivono come il seguente:

---

13

---

Inserendo quel valore in un programma, si attiva nella memoria del computer lo schema di bit che rappresenta il numero 13.

JavaScript utilizza un numero fisso di bit e precisamente 64, per memorizzare ogni singolo valore di tipo numero. Poiché le combinazioni possibili con 64 bit non sono illimitate, anche la quantità di numeri rappresentabili ha un limite. Dato  $N$  per le cifre decimali, la quantità massima di numeri che si può rappresentare è  $10^N$ . Analogamente, con 64 cifre binarie si possono rappresentare  $2^{64}$  numeri diversi, che corrispondono a circa 18 quintilioni (18 seguito da 18 zeri): che sono comunque tanti numeri!

In passato, la memoria dei computer era molto più piccola e si usavano gruppi di 8 o

16 bit per rappresentare i numeri. Con una quantità di numeri tanto ridotta, capitava spesso di *eccedere* il limite e ritrovarsi con numeri che non rientravano nei bit a disposizione. Oggi, anche i personal computer hanno moltissima memoria: si può pertanto lavorare con blocchi da 64 bit e non c'è più bisogno di preoccuparsi di eccedenze di dati quando si ha a che fare con numeri astronomici.

Tuttavia, non tutti i numeri interi entro i 18 quintilioni possono essere espressi in JavaScript. Infatti, poiché i bit a disposizione servono anche per i numeri negativi, uno dei bit è riservato al segno. Inoltre, vanno previsti numeri non interi (decimali) e, per questo, alcuni dei bit sono riservati alla posizione della virgola. Il numero intero più grande memorizzabile in JavaScript sta pertanto nell'ordine dei 9 quadrillioni: ossia, un numero di 15 cifre, che è comunque sempre un numero bello grosso. I decimali si esprimono con il punto:

---

9.81

---

Per numeri molto grandi o molto piccoli, si può utilizzare anche la notazione scientifica, aggiungendo un “e” (per “esponente”), seguito dall'esponente del numero:

---

2.998e8

---

Ossia,  $2,998 \times 10^8 = 299.800.000$ .

I calcoli con i numeri *interi* più piccoli di 9 quadrillioni saranno sempre precisi. Purtroppo, i calcoli con i decimali non sempre lo sono. Proprio come non si può esprimere con precisione  $\pi$ , il pi greco, con un numero finito di cifre decimali, molti altri numeri perdono precisione quando si hanno solo 64 bit di memoria a disposizione. Pertanto è importante tenerlo presente e trattare i numeri decimali come approssimazioni e non come valori precisi.

## Aritmetica

L'aritmetica è lo strumento principale per lavorare con i numeri. Le operazioni aritmetiche come l'addizione e la moltiplicazione prendono due valori di tipo numerico e producono un nuovo numero. In JavaScript si impostano come segue:

---

100 + 4 \* 11

---

I simboli + e \* si chiamano *operatori*. Il primo serve per l'addizione, il secondo per la moltiplicazione. Mettendo un operatore tra due valori, lo si applica a quei valori per produrne uno nuovo.

Come va interpretato l'esempio in termini di precedenza? Così com'è scritta, l'operazione svolge per prima la moltiplicazione e aggiunge poi 100 al risultato. Come si fa abitualmente in matematica, per cambiare impostazione basta inserire l'addizione tra parentesi:

---

(100 + 4) \* 11

---

Per la sottrazione, si usa l'operatore `-`; la barra `/` è l'operatore per la divisione.

Quando gli operatori sono indicati senza parentesi, l'ordine in cui sono utilizzati dipende dalla loro *precedenza*. Come si è visto nell'esempio, la moltiplicazione ha precedenza sull'addizione. L'operatore `/` per la divisione ha la stessa precedenza di `*`, quello per la moltiplicazione. Gli operatori `+` e `-` hanno uguale precedenza tra loro. Quando l'operazione indica operatori con la stessa precedenza, come in `1 - 2 + 1`, sono applicati nell'ordine in cui appaiono, da sinistra a destra:  $(1 - 2) + 1$ .

Le regole sulla precedenza non sono comunque un problema: basta ricordarsi di aggiungere le parentesi quando si è in dubbio.

Esiste un ultimo operatore aritmetico, che non sempre viene immediatamente riconosciuto: il simbolo `%`, che serve per rappresentare il *resto*. Ciò significa che  $x \% Y$  indica il resto della divisione di  $x$  per  $Y$ . Per esempio,  $314 \% 100$  dà  $14$  e  $144 \% 12$  dà  $0$ . La precedenza dell'operatore per il resto è uguale a quella di moltiplicazione e divisione. Quest'operatore è chiamato anche *modulo*, sebbene tecnicamente *resto* sia più accurato.

## Numeri speciali

In JavaScript esistono tre valori speciali, considerati di tipo numerico, che tuttavia non si comportano come gli altri numeri. I primi due sono `Infinity` e `-Infinity`, che rappresentano il positivo e il negativo dell'infinito. Per definizione, questi valori non cambiano mai: `Infinity - 1` è sempre `Infinity`, e così via. Non conviene comunque affidarsi troppo ai calcoli su base infinito. Non sono matematicamente affidabili e conducono al terzo numero speciale: `NaN`.

`NaN` sta per “not a number” (non un numero), e questo anche se si tratta di un valore di tipo numerico. Si ottiene questo risultato quando, per esempio, si cerca di calcolare la divisione di  $0$  per  $0$  (`0 / 0`) o `Infinity - Infinity` o qualunque altro risultato di operazioni numeriche che non diano un risultato preciso e significativo.

## Stringhe

I valori di tipo *stringa* servono per rappresentare del testo e si scrivono inserendoli tra virgolette:

---

`"Toppa la falla con la gomma da masticare"`

`'Le scimmie fanno ciao'`

---

Si usano indifferentemente virgolette semplici o doppie, purché aperte e chiuse correttamente.

Si può inserire praticamente di tutto tra virgolette: JavaScript lo intenderà come valore stringa. Ci sono tuttavia alcuni caratteri problematici, per esempio, se si volessero inserire delle virgolette tra virgolette. Non si possono neppure inserire degli *a capo* premendo semplicemente il tasto `INVIO`, perché le stringhe vanno sempre scritte su un'unica riga, non

importa quanto lunga.

Per consentire l'inserimento in una stringa di tali caratteri, si usa una notazione che fa ricorso alla barra rovesciata. Ogni volta che si trova una barra \ all'interno del testo tra virgolette, il carattere che segue immediatamente la barra rovesciata prende un significato speciale. In questa sua funzione speciale, la barra rovesciata si chiama *carattere di escape*. Così per esempio, una virgola preceduta da una barra rovesciata non chiude la stringa ma ne fa parte; quando la barra rovesciata è seguita da n, la combinazione \n viene intesa come a capo; analogamente, \t indica un segno di tabulazione. Ecco un esempio di stringa:

---

```
"Questa è la prima riga\nE questa è la seconda"
```

---

Il testo viene letto come segue:

```
Questa è la prima riga
```

```
E questa è la seconda
```

---

Quando si vuole che la barra rovesciata in una stringa sia letta come barra rovesciata e non come segnale speciale, basta usarne due di seguito. La stringa "L'a capo si scrive "\n." si esprime come segue:

---

```
"L'a capo si scrive \"\\n\"."
```

---

Non è possibile dividere, moltiplicare o sottrarre stringhe, ma tra una stringa e l'altra si può usare l'operatore +. In questo caso, non viene svolta un'addizione, ma un *concatenamento*: in pratica, il segno + incolla le stringhe una dopo l'altra. Il codice seguente darà la stringa "concatenare":

---

```
"con" + "cat" + "e" + "nare"
```

---

Esistono altri modi per manipolare le stringhe, come vedremo quando si parla di metodi nel [Capitolo 4](#).

## Operatori unari

Non tutti gli operatori sono simboli. Alcuni sono parole: per esempio, typeof. Quest'operatore produce un valore stringa che indica il tipo di dati assegnato. Per esempio:

---

```
console.log(typeof 4.5)
// → numero
console.log(typeof "x")
// → stringa
```

---

Nei frammenti di codice di esempio, console.log indica che si vuole visualizzare il

risultato di una certa operazione. Eseguendo il codice, il valore prodotto viene visualizzato sullo schermo, anche se l'aspetto dipende dal tipo di ambiente JavaScript dove viene eseguito il codice.

Mentre gli operatori descritti finora agivano tutti su due valori, `typeof` si applica a un solo valore. Gli operatori che utilizzano due valori si chiamano operatori *binari*, mentre quelli che ne utilizzano uno solo si chiamano operatori *unari*. L'operatore meno può essere utilizzato sia come operatore unario, sia come operatore binario.

---

```
console.log(- (10 - 2))  
// → -8
```

---

## Valori booleani

In alcuni casi, c'è bisogno di valori che si limitino a distinguere due possibilità, come "sì" e "no" o "acceso" e "spento". Per questi casi, JavaScript offre un tipo *booleano* che ha solo due valori: `true` e `false` (vero e falso).

### Comparazioni

Ecco un esempio che dà come risultato dei valori booleani:

```
console.log(3 > 2)  
// → true  
console.log(3 < 2)  
// → false
```

---

I segni `>` e `<` significano rispettivamente "è maggiore di" ed "è minore di", e sono operatori binari. Applicando questi operatori ai valori, il risultato è espresso da un valore booleano che indica se la condizione è vera o falsa per i numeri dati.

Allo stesso modo si possono mettere a confronto le stringhe:

```
console.log("Aardvark" < "Zoroaster")  
// → true
```

---

L'ordine di grandezza per le stringhe è grossomodo alfabetico, con la differenza che le lettere maiuscole sono sempre "minori" delle minuscole. Pertanto, `"z" < "a"` dà come risultato `true`. L'ordinamento delle stringhe prende in considerazione anche i caratteri non alfabetici, come i segni di punteggiatura: in effetti, la comparazione prende come riferimento lo standard Unicode. In questo standard è assegnato un codice numerico a praticamente tutti i caratteri possibili, compresi quelli degli alfabeti greco, arabo, giapponese, Tamil e tutti gli altri. Il ricorso ai caratteri Unicode è molto utile per memorizzare stringhe, perché consente di rappresentarle come sequenze di numeri. Per

confrontare più stringhe, JavaScript procede da sinistra a destra confrontando uno per uno i codici dei singoli caratteri.

Altri operatori di comparazione sono `>=` (maggiore o uguale a), `<=` (minore o uguale a), `==` (uguale a), and `!=` (non uguale a).

---

```
console.log("Punge" != "Graffia")
// → true
```

---

In JavaScript, solo un valore non è uguale a se stesso e quel valore è `NaN` (“not a number”).

---

```
console.log(NaN == NaN)
// → false
```

---

`NaN` è utilizzato per indicare il risultato di un’operazione assurda o impossibile e come tale non è uguale al risultato di *nessun’altra* operazione assurda o impossibile.

## Operatori logici

Per le operazioni che si possono applicare ai valori booleani, JavaScript offre tre operatori logici: `and`, `or` e `not`. Gli operatori logici servono per “ragionare” su dati booleani.

L’operatore `&&` rappresenta l’*and* logico. Si tratta di un operatore binario il cui risultato è `true` solo se sono `true` entrambi i valori passati:

---

```
console.log(true && false)
// → false
console.log(true && true)
// → true
```

---

L’operatore `||` indica l’*or* logico. Il suo risultato è `true` quando è `true` uno dei due valori passati:

---

```
console.log(false || true)
// → true
console.log(false || false)
// → false
```

---

Il simbolo per l’operatore logico *not* è un punto esclamativo `!`. Si tratta di un operatore unario che inverte il valore passato: `!true` dà come risultato `false` e `!false` dà come risultato `true`.

Quando si usano insieme operatori booleani, operatori aritmetici e altri operatori, la necessità delle parentesi non è sempre ovvia. In pratica, potete attenervi al principio di massima che, di tutti gli operatori esaminati sinora, `||` è quello che ha la precedenza inferiore; seguono `&&`, poi gli operatori di comparazione (`>`, `==` eccetera) e poi gli altri. La

scelta di quest'ordine è stata dettata dall'esigenza di mantenere al minimo il numero delle parentesi in espressioni complesse, come la seguente:

---

```
1 + 1 == 2 && 10 * 10 > 50
```

---

L'ultimo operatore logico non è né unario, né binario, bensì ternario. Opera, infatti, su tre valori e utilizza come simboli il punto di domanda e i due punti:

---

```
console.log(true ? 1 : 2);  
// → 1  
console.log(false ? 1 : 2);  
// → 2
```

---

Questo è l'operatore *condizionale*, o semplicemente operatore *ternario* in quanto è l'unico di questo tipo. Il valore alla sinistra del punto di domanda “determina” la scelta del risultato tra gli altri due valori. Quando il risultato è `true`, la scelta è caduta sul valore che sta in mezzo; quando è `false`, il valore del risultato è quello del valore a destra.

## Valori indefiniti

I due valori speciali `null` e `undefined` sono utilizzati per indicare l'assenza di un valore significativo. Sono valori di per sé, eppure non contengono informazioni o dati.

Anche le operazioni che non producono un valore significativo (di cui vedrete qualche esempio più avanti) devono comunque avere un qualche valore, che sarà `undefined`.

La differenza di significato tra `undefined` e `null` è una svista nella progettazione di JavaScript e in generale non ha importanza. Quando dovete davvero preoccuparvi di valori di questo tipo, vi consiglio di trattare `null` e `undefined` come intercambiabili, come si vedrà tra poco.

## Conversione automatica dei tipi

Nell'introduzione, ho detto che JavaScript accetta di tutto, compresi programmi che fanno cose strane. L'espressione che segue dimostra questo concetto:

---

```
console.log(8 * null)  
// → 0  
console.log("5" - 1)  
// → 4  
console.log("5" + 1)  
// → 51  
console.log("cinque" * 2)
```

```
// → NaN  
console.log(false == 0)  
// → true
```

---

Quando si applica un operatore al tipo di dati “sbagliato”, JavaScript si limita a convertirne il valore nel tipo richiesto, impiegando una serie di regole che non sempre seguono la nostra logica. Questo comportamento si chiama *forzatura del tipo*. Pertanto, il risultato `null` della prima espressione diventa `0` e il `"5"` della seconda diventa `5` (passando da stringa a numero). Poiché nella terza espressione l’operatore `+` tenta di concatenare delle stringhe prima di effettuare un’addizione, l’`'1` viene convertito in `"1"` (da numero a stringa).

Quando si tenta di convertire in numero qualcosa che non ha un’ovvia corrispondenza con un numero (come `"cinque"` o `undefined`), si produce il valore `Nan`. Poiché tutte le operazioni su `Nan` danno come risultato `Nan`, se incontrate risultati di questo tipo dove non ve li aspettate, il primo indizio da cercare sta nelle conversioni forzate.

Se si confrontano valori dello stesso tipo con `==`, il risultato dovrebbe essere facilmente prevedibile: otterrete `true` quando i valori sono uguali, fatta eccezione per valori `Nan`. Quando il tipo è diverso, JavaScript usa una serie di regole complicate e confuse per stabilire come procedere. In molti casi, tenta di convertire uno dei valori nel tipo dell’altro. Ma quando da una parte o dall’altra dell’operatore si trovano `null` o `undefined`, darà `true` solo se entrambi i valori sono `null` o `undefined`.

---

```
console.log(null == undefined);  
// → true  
console.log(null == 0);  
// → false
```

---

L’ultimo aspetto di questo comportamento risulta spesso utile: volendo verificare se un certo valore ha un valore vero invece di `null` o `undefined`, dovete impostare una comparazione con `null` attraverso l’operatore `==` (o `!=`).

Che cosa succede quando volete verificare se un certo dato fa riferimento proprio al valore `false`? Le regole per convertire stringhe e numeri in valori booleani dicono che `0`, `Nan` e la stringa vuota `("")` sono calcolati come `false`, mentre tutti gli altri valori sono `true`. Per questo, danno risultato `true` anche operazioni come `0 == false` e `"" == false`. Nei casi dove non volete la conversione di tipo automatica, esistono altri due operatori: `==` e `!=`. Il primo verifica se un certo valore è identico a un altro, mentre il secondo indica se non è esattamente identico. Pertanto, `"" == false` dà `false`, proprio come previsto.

In generale, gli operatori di comparazione di tre caratteri andrebbero sempre usati quando volete evitare possibili problemi con le conversioni di tipo. Ma quando siete certi che il tipo degli operandi sia lo stesso, saranno sufficienti gli operatori più brevi.

## Il corto circuito degli operatori logici

Gli operatori logici `&&` e `||` si comportano in maniera differente con valori di tipi diversi. Per esempio, convertono il valore di sinistra in booleano per stabilire come procedere, ma restituiscono il valore originario di sinistra o quello di destra secondo l'operatore e il risultato di quella conversione.

L'operatore `||` restituisce il valore di sinistra quando lo si può convertire in `true`; diversamente, restituisce il valore di destra. Questa conversione dà il risultato più prevedibile per i valori booleani e si comporta in modo analogo per gli altri tipi.

---

```
console.log(null || "utente")
// → utente
console.log("Carlo" || "utente")
// → Carlo
```

---

Questa funzionalità consente di utilizzare l'operatore `||` come scorciatoia per restituire un valore predefinito. Se gli viene assegnata un'espressione che potrebbe produrre un valore vuoto a sinistra, viene utilizzato al suo posto il valore sulla destra.

L'operatore `&&` ha un comportamento simile, ma invertendo le posizioni. Quando il valore sulla sinistra potrebbe essere convertito in `false`, restituisce quel valore; altrimenti restituisce il valore sulla destra.

Un'altra importante proprietà di questi due operatori è che l'espressione alla loro destra viene presa in considerazione solo quando serve. Per esempio, con `true || x` non ha importanza che cosa sia `x`. Se anche `x` fosse un'espressione sbagliata, il risultato sarebbe `true` e `x` non sarebbe mai calcolata. Lo stesso vale per `false && x`, che dà come risultato `false` e pertanto ignora `x`. Questo comportamento si chiama *valutazione in corto circuito*.

L'operatore condizionale si comporta in modo simile. La prima espressione viene sempre calcolata, mentre non è calcolato il valore che non viene scelto tra il secondo e il terzo.

## Riepilogo

In questo capitolo abbiamo esaminato quattro tipi di valori in JavaScript: numeri, stringhe, valori booleani e indefiniti.

Per impostare i valori, è sufficiente digitare il nome (`true`, `null`) o il valore (13, `"abc"`). Attraverso gli operatori, si possono combinare e trasformare i valori. JavaScript offre operatori binari per l'aritmetica (+, -, \*, / e %), per la concatenazione di stringhe (+), per la comparazione (==, !=, ===, !==, <, >, <=, >=) e per le operazioni logiche (&&, ||). Offre inoltre alcuni operatori unari (- per rendere negativo un numero, ! per negare logicamente e `typeof` per trovare il tipo di un certo valore) e un operatore ternario (?:) per estrarre uno solo di due valori in base a un terzo valore.

Con queste basi, è già possibile far uso di JavaScript come di un calcolatore tascabile, ma non molto di più. Nel prossimo capitolo si vedrà come mettere insieme queste espressioni per realizzare dei piccoli programmi.

## STRUTTURA DEI PROGRAMMI

*E il mio cuore s'illumina di rosso sotto la pelle traslucida e devono amministrarmi 10cc di JavaScript per farmi riprendere conoscenza (rispondo bene alle tossine nel sangue). Ehi, quella roba ti fa proprio scoppiare le branchie!*

\_why, *Why's (Poignant) Guide to Ruby*

In questo capitolo, cominciamo a occuparci della *programmazione* vera e propria. Impareremo a controllare il linguaggio ben al di là dei frammenti che abbiamo visto sinora, fino a esprimere della prosa ricca di significati.

### Espressioni e dichiarazioni

Nel [Capitolo 1](#), abbiamo visto come impostare dei valori e come applicarvi degli operatori per ottenere nuovi valori. Questa è una parte essenziale di tutti i programmi JavaScript, ma è solo una parte.

Un frammento di codice che produce un valore si chiama *espressione*. Tutti i valori espressi letteralmente (per esempio, 22 o "psicoanalisi") sono espressioni. Anche un'espressione tra parentesi è un'espressione, così come lo è un operatore binario applicato a due espressioni o un operatore unario applicato a una sola.

Quest'aspetto illustra uno dei vantaggi dell'interfaccia basata sul linguaggio. Le espressioni si possono nidificare proprio come si impostano le frasi all'interno dei periodi nella lingua parlata: ogni periodo può contenere frasi, che a loro volta possono contenere proposizioni subordinate e così via. In questo modo, possiamo combinare più espressioni per rappresentare calcoli di varia complessità.

Se l'espressione corrisponde a una proposizione della lingua parlata, una *dichiarazione* in JavaScript corrisponde al periodo. Un programma non è altro che un elenco di dichiarazioni.

La forma più semplice per le dichiarazioni è un'espressione seguita da un punto e virgola. Quello che segue è un programma:

---

```
1;
!false;
```

---

Peccato che sia un programma inutile. Lo scopo di un'espressione può essere

semplicemente di produrre un valore, che sarà poi usato dall'espressione che la contiene. La dichiarazione invece è fine a se stessa e ha significato solo se genera un cambiamento. Potrebbe far comparire qualcosa sullo schermo (che sarà comunque "cambiato") o potrebbe modificare lo stato interno della macchina in modo da cambiare anche le dichiarazioni che la seguono. Le modifiche di questo tipo si chiamano *effetti collaterali*. Ma le dichiarazioni nell'esempio riportato qui sopra si limitano a produrre i valori 1 e true e a buttarli via immediatamente: ciò non lascia alcun tipo di segno e pertanto non è un cambiamento. Quando si esegue il programma, non succede nulla che si possa rilevare.

In alcuni casi, in JavaScript potete omettere il punto e virgola alla fine della dichiarazione. Il punto e virgola è tuttavia sempre necessario per dividerla da quella che segue. Le regole sull'uso del punto e virgola sono abbastanza confuse e possono provocare errori inutili. Nel libro, per evitare problemi si usano punti e virgola in chiusura di tutte le dichiarazioni e consiglio di fare lo stesso finché non avrete assimilato tutti i concetti di questa sintassi.

## Variabili

Come fa il programma a mantenere il suo stato interno? Come fa a "ricordarsi" delle cose? Abbiamo visto come produrre nuovi valori da valori vecchi, ma ciò non modifica i vecchi valori; peraltro, i nuovi valori vanno utilizzati immediatamente, altrimenti spariscono. Per catturare e mantenere i valori, JavaScript mette a disposizione le *variabili*.

---

```
var caught = 5 * 5;
```

---

Quest'esempio illustra il secondo tipo di dichiarazioni, dove la parola speciale (*parola chiave*) var indica che nella frase si definisce una variabile. La parola chiave è seguita dal nome della variabile, alla quale possiamo assegnare un valore iniziale aggiungendo un operatore = e un'espressione.

Nella dichiarazione di esempio, si crea una variabile caught, che registrerà in memoria il numero prodotto moltiplicando 5 per 5.

Una volta definita, la variabile può essere utilizzata come un'espressione, il cui valore è quello della variabile stessa. Per esempio:

---

```
var ten = 10;  
console.log(ten * ten);  
// → 100
```

---

Alle variabili si può assegnare qualunque nome, esclusi quelli riservati dal linguaggio (come var). I nomi delle variabili non devono contenere spazi. Possono contenere numeri, come per esempio in abc123, purché il numero non sia il loro primo carattere. Simboli e segni di punteggiatura non sono validi, con l'esclusione dei simboli \$ e \_.

Il valore assegnato alle variabili non è immutabile. Per assegnare un nuovo valore, si

può utilizzare in qualunque momento l'operatore =:

---

```
var umore = "buono";
console.log(umore);
// → buono
umore = "cattivo";
console.log(umore);
// → cattivo
```

---

Le variabili in realtà sono più dei tentacoli che delle scatole: non *contengono* valori, ma li *recuperano* e due variabili possono fare riferimento allo stesso valore. Il programma può accedere soltanto ai valori che rimangono “in possesso” delle variabili. Quando volete memorizzare un certo dato, dovete far crescere o riutilizzare un tentacolo che lo vada a recuperare.



Nell'esempio che segue, create una variabile per tenere a mente il numero di euro che Luigi vi deve. Quando Luigi paga 35€, assegnate un nuovo valore alla variabile:

---

```
var luigiDare = 140;
luigiDare = luigiDare - 35;
console.log(luigiDare);
// → 105
```

---

Se definite una variabile senza assegnarvi un valore, non c'è nulla da afferrare e la variabile sparisce. Se chiedete al programma il valore di una variabile vuota, viene restituito il valore `undefined`.

Una singola dichiarazione `var` può definire più variabili, separando le definizioni con la virgola:

---

```
var one = 1, two = 2;
```

```
console.log(one + two);
// → 3
```

---

## Parole chiave e parole riservate

Le parole che hanno un significato speciale, come `var`, si chiamano *parole chiave* e non possono essere usate come nomi di variabili. Esistono inoltre altre parole “riservate” per eventuali versioni future di JavaScript. Nemmeno queste parole sono valide come nomi di variabili, anche se alcuni ambienti JavaScript le consentono. L’elenco completo di parole chiave e riservate è piuttosto lungo:

---

```
break case catch class const continue debugger default delete do else enum
export extends false finally for function if implements import in
instanceof interface let new null package private protected public return
static super switch this throw true try typeof var void while with yield
```

---

Non dovete impararle a memoria, purché vi ricordiate di verificare la validità del nome delle variabili se qualcosa non funzionasse come vi aspettate.

## L’ambiente

L’insieme delle variabili e dei rispettivi valori che esistono in un determinato momento si chiama *ambiente*. Quando si avvia un programma, l’ambiente non è vuoto: contiene sempre le variabili che fanno parte del linguaggio e, il più delle volte, delle variabili che consentono di interagire col sistema circostante. Per esempio, in un browser si trovano variabili e funzioni che ispezionano e influenzano il sito Web caricato, e che rilevano l’input da mouse e tastiera.

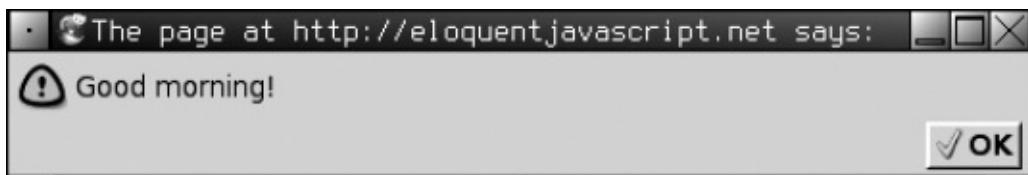
## Funzioni

Molti dei valori disponibili nell’ambiente predefinito sono di tipo *funzione*. Una funzione è un brano di programma racchiuso in un valore. Assegnando questi valori, si esegue il programma in essi racchiuso. Per esempio, nell’ambiente browser la variabile `alert` contiene una funzione che mostra una finestra di dialogo con un messaggio. La si invoca così:

---

```
alert("Good morning!");
```

---



Per eseguire una funzione, la si *invoca*, la si *richiama* o la si *applica*, mettendo delle

parentesi dopo un'espressione che produce un valore di funzione. Di solito, userete direttamente il nome della variabile che contiene la funzione. Il valore tra parentesi viene passato al programma all'interno della funzione. Nell'esempio sopra riportato, la funzione `alert` prende la stringa tra parentesi come testo da visualizzare nella finestra. I valori assegnati alle funzioni si chiamano *argomenti*. La funzione `alert` ne accetta uno solo, ma altre funzioni possono richiedere più argomenti o più tipi di argomento.

## La funzione `console.log`

La funzione `alert` può sostituire un risultato di output in fase sperimentale, ma non è sempre la soluzione ideale. Negli esempi precedenti, per produrre dei risultati abbiamo usato `console.log`. Praticamente tutti i sistemi JavaScript, compresi i browser moderni e Node.js, mettono a disposizione una funzione `console.log` che visualizza i suoi argomenti come output di una qualsiasi interfaccia di testo. Nei browser, l'output si trova nella console di JavaScript. Questa parte dell'interfaccia del browser è di solito nascosta; in genere, vi si accede premendo **F12**, oppure **CMD+OPT+I** sui Mac. Se non funzionano le scorciatoie da tastiera, accedete alla console dai menu del browser, dove la trovate sotto Strumenti per sviluppatori o Sviluppo o altro secondo il browser.

Negli esempi del libro, l'output di `console.log`, che verrebbe visualizzato nella console JavaScript del browser, viene riportato subito dopo l'esempio:

---

```
var x = 30;  
console.log("il valore di x è", x);  
// → il valore di x è 30
```

---

In genere, i nomi delle variabili non possono contenere punti, ma `console.log` ne ha uno. Questo perché non si tratta di una semplice variabile, ma di un'espressione che recupera la proprietà `log` dal valore mantenuto dalla variabile `console`. Nel [Capitolo 4](#) scopriremo esattamente di che cosa si tratta.

## Valori restituiti

Mostrare del testo sullo schermo o aprire una finestra di dialogo sono utili *effetti collaterali* delle funzioni. Le funzioni possono anche produrre valori e, in questi casi, non sempre hanno degli effetti collaterali. Per esempio, la funzione `Math.max` esamina dei dati di tipo numerico e restituisce il maggiore:

---

```
console.log(Math.max(2, 4));  
// → 4
```

---

Quando una funzione produce un valore, si dice che lo *restituisce*. In JavaScript, tutto ciò che produce un valore è un'espressione, il che significa che si possono richiamare funzioni all'interno di espressioni più grandi. Nell'esempio, si richiama `Math.min`, che è

l'opposto di `Math.max`, come input per l'operatore di addizione:

---

```
console.log(Math.min(2, 4) + 100);  
// → 102
```

---

Il prossimo capitolo spiega come scrivere le vostre funzioni.

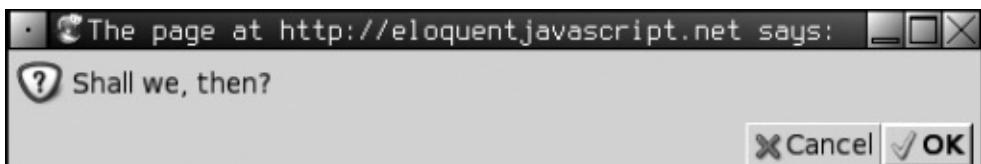
## Prompt e confirm

Oltre ad `alert`, gli ambienti browser contengono altre funzioni. Usate la funzione `confirm` per chiedere all'utente di premere OK o Cancel per confermare o annullare una certa azione. Il valore restituito in questo caso è di tipo booleano: `true` se l'utente fa clic su OK, `false` se fa clic su Cancel.

---

```
confirm("Shall we, then?");
```

---

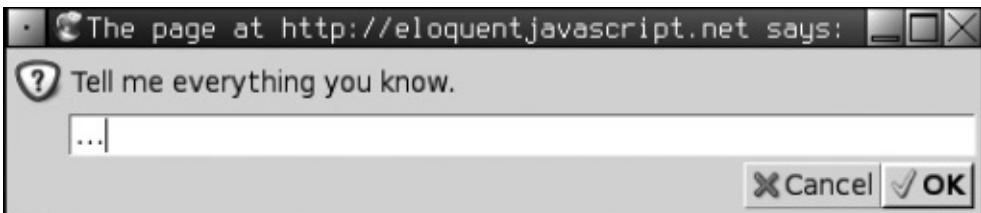


La funzione `prompt` può essere usata per impostare una domanda “aperta” e ha due argomenti: il primo è la domanda, il secondo il testo proposto inizialmente all'utente. La finestra di dialogo consente di inserire del testo, che la funzione restituisce come stringa.

---

```
prompt("Tell me everything you know.", "...");
```

---



Queste due funzioni non sono molto usate nella programmazione moderna per il Web, sostanzialmente perché non avrete controllo sull'aspetto delle finestre, ma sono utili per piccoli programmi di prova e in sede sperimentale.

## Struttura di controllo

Quando il vostro programma contiene più di una dichiarazione, l'esecuzione avviene dall'alto al basso in modo lineare. L'esempio più semplice è dato da un programma con due dichiarazioni. Nella prima si chiede all'utente un numero, nella seconda, che viene eseguita dopo la prima, si mostra il quadrato di quel numero:

---

```
var theNumber = Number(prompt("Scegli un numero", ""));
```

```
alert("Il numero scelto è la radice quadrata di " +  
      theNumber * theNumber);
```

---

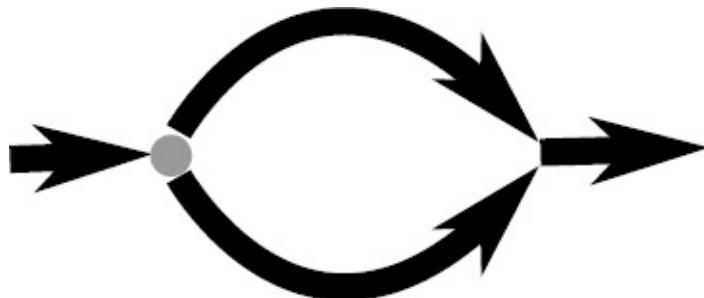
La funzione `Number` converte un valore in un numero. La conversione è necessaria in questo caso perché il risultato di `prompt` è un valore di tipo stringa, ma noi vogliamo un numero. Altre funzioni di questo tipo sono `String` e `Boolean`, che convertono i valori rispettivamente in tipo stringa e tipo booleano.

La rappresentazione schematica più semplice della struttura di controllo è la seguente:



## Esecuzione condizionale

L'esecuzione lineare delle dichiarazioni non è l'unica possibilità a nostra disposizione. Esiste anche *l'esecuzione condizionale*, dove scegliamo la strada da seguire tra le due alternative offerte da un valore booleano. Eccone una rappresentazione schematica:



L'esecuzione condizionale si indica in JavaScript con la parola chiave `if`. Nel caso più semplice, vogliamo solo che il codice sia eseguito se, e soltanto se, viene soddisfatta una certa condizione. Per esempio, nel programma precedente, possiamo decidere di mostrare il quadrato del numero dato soltanto se il carattere inserito è di tipo numerico:

---

```
var theNumber = Number(prompt("Scegli un numero", ""));  
if (!isNaN(theNumber))  
    alert("Il numero scelto è la radice quadrata di " +  
          theNumber * theNumber);
```

---

Con questa variante, non si ottiene un output se invece di un numero digitate “formaggio”.

La parola chiave `if` esegue o salta una dichiarazione secondo il valore di un'espressione booleana. L'espressione determinante viene specificata dopo la parola chiave, tra parentesi, e prima della dichiarazione da eseguire.

La funzione `isNaN` è una funzione standard di JavaScript che restituisce `true` solo se l'argomento passato è `Nan`. La funzione `Number` restituisce `Nan` quando riceve una stringa che non rappresenta un numero valido. Pertanto, questa condizione si traduce in “a meno che `theNumber` sia non-un-numero, fai questo”.

In generale, dovete prevedere del codice non solo quando è soddisfatta una certa condizione, ma anche in caso contrario. Il percorso alternativo è rappresentato in figura dalla seconda freccia. Per questo, si usa la parola chiave `else` insieme a `if` per impostare due percorsi di esecuzione separati.

---

```
var theNumber = Number(prompt("Scegli un numero", ""));
if (!isNaN(theNumber))
    alert("Il numero scelto è la radice quadrata di " +
          theNumber * theNumber);
else
    alert("Quel che mi hai dato non è un numero!");
```

---

Se abbiamo più di due possibilità di scelta, possiamo “concatenare” più coppie di `if/else`. Per esempio:

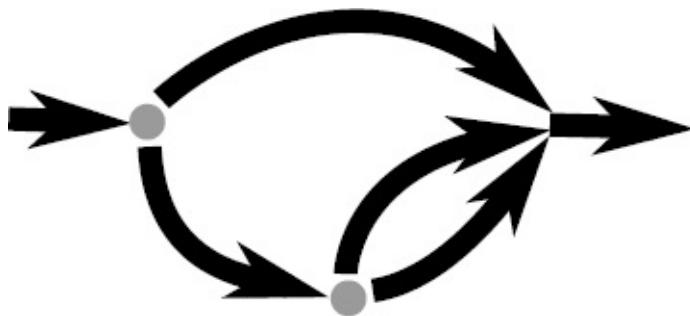
---

```
var num = Number(prompt("Scegli un numero", "0"));
if (num < 10)
    alert("Piccolo");
else if (num < 100)
    alert("Medio");
else
    alert("Grande");
```

---

Per prima cosa, il programma verifica se `num` è inferiore a 10. Se lo è, segue quel ramo della dichiarazione, visualizza "Piccolo" e termina. Altrimenti, segue il ramo `else`, che contiene a sua volta un secondo `if`. Se è vera la seconda condizione ( $<100$ ), il numero sarà compreso tra 10 e 100 e viene pertanto visualizzato "Medio". Altrimenti, segue l'ultimo ramo `else`.

La struttura di controllo del programma può essere rappresentata così:



## Cicli while e do

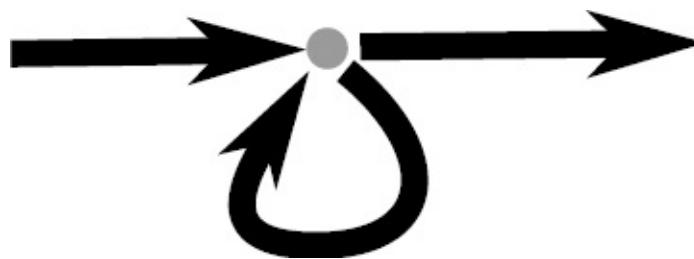
L'esempio che segue è un programma che stampa tutti i numeri pari, da 0 a 12:

---

```
console.log(0);
console.log(2);
console.log(4);
console.log(6);
console.log(8);
console.log(10);
console.log(12);
```

---

Sebbene questo programma funzioni, l'idea è di creare qualcosa che riduca il lavoro, non lo aumenti. Se l'intervallo di numeri salisse a 1000, una soluzione come questa sarebbe assurda. Dovremo pertanto trovare una struttura di controllo che consenta di ripetere il codice. Questo tipo di struttura si chiama *ciclo*:



La struttura di controllo ciclica consente di tornare a un punto precedente del programma e ripetere le istruzioni per lo stato corrente. Combinando quest'impostazione con una variabile che tiene conto della somma, possiamo scrivere quanto segue:

---

```
var number = 0;
while (number <= 12) {
    console.log(number);
    number = number + 2;
}
// → 0
// → 2
// ... eccetera
```

---

Le dichiarazioni che iniziano con la parola chiave `while` impostano un ciclo. Dopo `while`, si trova un'espressione tra parentesi, seguita da una dichiarazione, in modo molto simile all'impostazione di `if`. Il ciclo esegue l'istruzione finché l'espressione produce un valore che rimane `true` quando viene convertito in tipo booleano.

In questo ciclo, vogliamo stampare il numero corrente e aggiungere due alla variabile. Per eseguire più dichiarazioni in un ciclo, dobbiamo chiuderle tra parentesi graffe, `{` e `}`. Le parentesi graffe hanno nelle dichiarazioni lo stesso ruolo che le parentesi tonde hanno nelle espressioni: le raggruppano e le fanno diventare una singola dichiarazione. Una sequenza di dichiarazioni tra parentesi graffe si chiama blocco.

In molti casi, si preferisce mettere tra parentesi graffe i singoli cicli `while` e

dichiarazioni `if`. Ciò soddisfa criteri di coerenza ed evita di ricontrizzare il numero di parentesi se si dovessero aggiungere o togliere dichiarazioni in un secondo tempo. Per amore di brevità, in questo libro non ho inserito tra parentesi i blocchi con una sola dichiarazione. Voi siete liberi di regolarvi come credete.

La variabile `number` dimostra come si può seguire il progresso di un programma. Ogni volta che si ripete il ciclo, `number` aumenta di 2. All'inizio di ogni ripetizione, il valore viene confrontato col numero 12 per stabilire se il programma ha terminato.

Nel prossimo esempio, impostiamo un programma che calcoli e visualizzi il valore di  $2^{10}$  (2 alla decima potenza). Impostiamo due variabili: una che tenga in memoria il risultato e una che memorizzi quante volte si è moltiplicato il risultato per 2. Il ciclo consente di verificare se la seconda variabile è arrivata a 10 prima di aggiornarle entrambe.

---

```
var result = 1;
var counter = 0;
while (counter < 10) {
  result = result * 2;
  counter = counter + 1;
}
console.log(result);
// → 1024
```

---

Si potrebbe partire anche da 1 e verificare se è `<= 10`, ma, come si vedrà nel [Capitolo 4](#), è preferibile abituarsi a iniziare il conteggio da 0.

Il ciclo `do` è una struttura di controllo simile al ciclo `while`, con una sola differenza: il ciclo `do` esegue sempre almeno una volta il corpo della dichiarazione e verifica se deve interrompersi solo alla fine di quella dichiarazione. Per questo, la prova viene dopo il corpo del ciclo:

---

```
do {
  var name = prompt("Chi sei?");
} while (!name);
console.log(name);
```

---

Il programma vi costringe a inserire un nome e continua a porre la domanda finché non riceve qualcosa di diverso da una stringa vuota. L'operatore `!` converte il valore in tipo booleano prima di rifiutarlo e tutte le stringhe che non sono vuote ("") vengono convertite in `true`.

## I rientri del codice

Avrete notato che davanti ad alcune dichiarazioni c'è dello spazio vuoto. In JavaScript, lo spazio vuoto non è necessario e i computer accettano anche programmi che non ne contengono. In effetti, sono facoltativi anche gli a capo e tutti i programmi si potrebbero scrivere su una sola riga. Lo scopo dei rientri è semplicemente quello di far risaltare la struttura del codice. Quando si scrive codice complesso e si aprono blocchi all'interno di altri blocchi, può essere difficile individuare immediatamente dove finisce un blocco e dove ne inizia un altro. Facendo rientrare le righe del programma in maniera corretta, la forma del programma "a colpo d'occhio" corrisponde alla forma dei blocchi in esso contenuti. Nel libro, si inseriscono due spazi per ogni blocco aperto, ma ciascuno segue le proprie consuetudini: alcuni usano sequenze di quattro caratteri, altri preferiscono i tabulatori.

## Cicli for

In generale, i cicli seguono lo schema riportato nell'esempio precedente. Per prima cosa, si crea una variabile "contatore" che segue il progresso del ciclo. Segue poi un ciclo `while`, la cui espressione di prova verifica se il contatore ha raggiunto il limite stabilito. Alla fine del corpo del ciclo, si aggiorna il contatore.

Poiché questo schema è di uso molto comune, JavaScript e altri linguaggi simili ne offrono una forma abbreviata ma più completa nei cicli `for`.

---

```
for (var number = 0; number <= 12; number = number + 2)
    console.log(number);
// → 0
// → 2
// ... eccetera
```

---

Questo programma è del tutto equivalente all'esempio precedente. La sola cosa che è cambiata è che tutte le dichiarazioni relative allo "stato" del ciclo sono state raggruppate insieme.

Le parentesi dopo la parola chiave `for` devono contenere due punti e virgola. La parte che precede il primo punto e virgola *inizializza* il ciclo, di solito con la definizione di una variabile. La seconda parte è l'espressione che *verifica* se il ciclo deve continuare. La parte finale *aggiorna* lo stato del ciclo dopo ogni iterazione. Nella maggior parte dei casi, questa impostazione è più breve e più chiara di un costrutto `while`.

Ecco il codice per calcolare  $2^{10}$ , definito in un ciclo `for` invece di `while`:

---

```
var result = 1;
for (var counter = 0; counter < 10; counter = counter + 1)
    result = result * 2;
console.log(result);
```

Notate che, sebbene nessun blocco si apra con una parentesi graffa, la dichiarazione del ciclo ha ancora un rientro di due spazi, per evidenziare che “appartiene” alla riga che la precede.

## Uscire da un ciclo

I cicli non finiscono solo se la loro condizione dà come risultato `false`. Esiste una dichiarazione speciale, `break`, che ha l’effetto di interrompere immediatamente il ciclo che la contiene.

Questo programma illustra la dichiarazione `break`, che viene eseguita appena si trova un numero che sia contemporaneamente maggiore di 0 e uguale a 20 e divisibile per 7.

```
for (var current = 20; ; current++) {  
    if (current % 7 == 0)  
        break;  
}  
console.log(current);  
// → 21
```

---

L’operatore resto (%) offre un semplice modo per verificare se un numero sia divisibile per un altro: in questi casi, il resto della divisione è zero.

Il costrutto `for` nell’esempio qui sopra non contiene codice che verifichi la fine del ciclo. Ciò significa che il ciclo non finisce finché non viene eseguita la dichiarazione `break` al suo interno.

Se lasciate fuori la dichiarazione `break` o, per sbaglio, impostate una condizione che dà sempre `true` come risultato, il programma continua a girare in un *ciclo infinito*. I programmi che eseguono cicli infiniti non terminano mai, il che di solito è piuttosto negativo.

La parola chiave `continue` è simile a `break`, in quanto influenza il progresso del ciclo. Quando incontra `continue` nel corpo di un ciclo, il controllo del flusso esce dal corpo del ciclo e passa all’iterazione successiva.

## Aggiornamento rapido delle variabili

In particolar modo nei cicli, può esserci bisogno di “aggiornare” una variabile per memorizzare un valore basato sul valore precedente.

---

```
counter = counter + 1;
```

---

In JavaScript, il concetto si esprime più brevemente così:

---

```
counter += 1;
```

---

Esistono scorciatoie simili anche per altri operatori, tra cui `result *= 2` per raddoppiare il valore di `result` o `counter -= 1` per contare alla rovescia.

L'esempio si può pertanto abbreviare ancora:

---

```
for (var number = 0; number <= 12; number += 2)  
    console.log(number);
```

---

Gli equivalenti per `counter += 1` e `counter -= 1` sono ancora più brevi: `counter++` and `counter--`.

## Scelta di un valore con switch

Di solito il codice appare come il seguente:

---

```
if (variable == "valore1") azione1();  
else if (variable == "valore2") azione2();  
else if (variable == "valore3") azione3();  
else azionePredefinita();
```

---

L'alternativa disponibile col costrutto `switch` consente di “scegliere” la strada da seguire in maniera più diretta. Purtroppo, la sintassi di JavaScript, che deriva dai linguaggi di programmazione di tipo C/Java, può essere poco chiara, se non addirittura meno preferibile di una serie di dichiarazioni.

Ecco un esempio:

---

```
switch (prompt("Com'è il tempo?")) {  
    case "piove":  
        console.log("Portati l'ombrelllo.");  
        break;  
    case "fa caldo":  
        console.log("Vestiti leggero.");  
    case "nuvoloso":  
        console.log("Esci.");  
        break;  
    default:  
        console.log("Non so che tempo fa!");  
        break;  
}
```

---

All'interno dei blocchi aperti da `switch` potete inserire tutti i casi necessari. Il programma passa direttamente all'etichetta del valore assegnato a `switch o`, se non trova corrispondenze, al valore predefinito, `default`. Il codice inizia a eseguire le dichiarazioni in quella posizione, anche se sono sotto un'etichetta diversa, finché non trova una dichiarazione `break`. In alcuni casi, come per "fa caldo" nell'esempio, i dati possono essere condivisi da più casi ("Esci." vale sia quando "fa caldo", sia quando è "nuvoloso"). Ma attenzione: se si dimentica la dichiarazione `break`, l'esecuzione del programma non seguirà il percorso voluto.

## Capitalizzazione

Poiché i nomi delle variabili non possono contenere spazi, quando il nome della variabile utilizza più parole potete scegliere una delle alternative seguenti:

---

`coloredeliocchi`

`colore_degli_occhi`

`ColoreDegliOcchi`

`coloreDegliOcchi`

---

La prima forma può essere difficile da leggere. Personalmente apprezzo i segni di sottolineatura, ma sono più lunghi da scrivere. Le funzioni standard di JavaScript, e la maggioranza dei programmati, optano per l'ultimo stile: ogni parola inizia con una lettera maiuscola, tranne la prima. Non è difficile abituarsi e poiché mischiare convenzioni di denominazione rende il codice inutilmente confuso, seguiremo questa convenzione.

In alcuni casi, come per la funzione `Number`, si preferisce l'iniziale maiuscola perché questa funzione va contrassegnata come costruttore. Di costruttori si parla in dettaglio nel [Capitolo 6](#). Per ora, la cosa importante da ricordare è di accettare quest'apparente incoerenza.

## Commenti

Spesso, il codice di per sé non comunica tutte le informazioni che vorreste trasmettere a chi lo legge o lo fa in maniera talmente misteriosa, da risultare incomprensibile. Altre volte, potreste semplicemente voler inserire nel codice annotazioni o appunti. Per questo, si usano i *commenti*.

I commenti sono brani di testo inseriti in un programma, che vengono completamente ignorati dal computer. JavaScript ha due tipi di sintassi per l'inserimento di commenti. Per i commenti che stanno su una sola riga, potete inserire due barre (`//`) seguite dal testo:

---

```
var accountBalance = calculateBalance(account);
// calcolo il saldo del conto
accountBalance.adjust();
```

```
// rettifico il saldo
var report = new Report();
// estratto conto
addToReport(accountBalance, report);
// aggiungo il saldo all'estratto conto
```

---

I commenti aperti con // possono arrivare solo fino alla fine della riga. I brani di testo inseriti tra i delimitatori /\* e \*/ sono ignorati, anche se vanno a capo. Questa notazione è utile quando si aggiungono commenti più lunghi e brani che descrivono file o parti di programma.

---

```
/*
Ho notato questo numero per la prima volta quando lo trovai scarabocchiato
sulla copertina dei miei quaderni di scuola. Da allora, mi è capitato di
vederlo molte altre volte: come numero di telefono, o come numero seriale
di prodotti che ho acquistato. Evidentemente gli piaccio, e ho deciso
pertanto di tenermelo.
*/
var myNumber = 11213;
```

---

## Riepilogo

Sapete che un programma si compone di dichiarazioni, che a loro volta possono contenere altre dichiarazioni. Le dichiarazioni possono contenere espressioni, che a loro volta possono essere composte da espressioni più piccole.

Mettendo una dichiarazione dopo l'altra, ottenete programmi che vengono eseguiti dall'inizio alla fine. Nel flusso (o struttura) di controllo potete tuttavia inserire delle "interferenze" attraverso dichiarazioni condizionali (if, else e switch) e cicliche (while, do e for).

Le variabili consentono di tenere in memoria dei dati e vi sono utili per seguire lo stato del programma. L'ambiente è l'insieme delle variabili definite. I sistemi JavaScript mettono a disposizione del vostro ambiente diverse variabili standard.

Le funzioni sono valori speciali, che encapsulano un brano di programma. Potete invocarle con nomeFunzione(argomento1, argomento2). Questo tipo di chiamata di funzione è un'espressione, che può produrre un valore.

## Esercizi

Per informazioni su come verificare i risultati degli esercizi, fate riferimento all'Introduzione.

Ogni esercizio si apre con la descrizione del problema. Se non trovate la soluzione dopo aver letto questa parte introduttiva, provate a leggere i suggerimenti riportati alla fine del libro. Le soluzioni complete non si trovano nel libro, ma sul sito <http://eloquentjavascript.net/code>. Gli esercizi vi servono per imparare nuovi concetti. Per questo, vi suggerisco di guardare le soluzioni solo come verifica finale o almeno dopo averci sbattuto la testa per un bel po'.

## **Un ciclo per un triangolo**

Scrivete un ciclo che effettui sette chiamate a `console.log` per produrre il seguente triangolo:

---

```
#  
##  
###  
####  
#####  
######  
######
```

---

Nota: per trovare la lunghezza di una stringa, potete aggiungere `.length` dopo di essa.

---

```
var abc = "abc";  
console.log(abc.length);  
// → 3
```

---

## **FizzBuzz**

Scrivete un programma che utilizzi `console.log` per stampare tutti i numeri da 1 a 100, con due eccezioni:

1. stampate "Fizz" invece del numero per i numeri divisibili per 3,
2. stampate "Buzz" invece del numero per i numeri divisibili per 5.

Dopo averlo fatto funzionare, modificate il programma in modo che stampi "FizzBuzz" per tutti i numeri che siano divisibili sia per 3, sia per 5.

(Questo esercizio è in effetti un test di selezione per scartare un buon numero di candidati programmati. Se riuscite risolverlo potete considerarvi promossi!).

## **Scacchiera**

Scrivete un programma che crei una stringa che rappresenti una griglia 8x8, separando le righe con degli a capo. Ogni posizione della griglia deve essere occupata da uno spazio o da un carattere "#", in modo da simulare l'aspetto di una scacchiera.

Passando la stringa a `console.log`, si dovrebbe ottenere qualcosa di simile a quanto segue:

---

```
# # # #
# # # #
# # # #
# # # #
# # # #
# # # #
# # # #
# # # #
```

---

Quando avete un programma che genera questa struttura, definite una variabile `size = 8` e modificate lo in modo che valga per qualunque dimensione (`size`), producendo griglie dell'altezza e larghezza date.

# 3

## FUNZIONI

*Molti credono che la scienza informatica sia arte da geni. In realtà non lo è affatto: è solo un sacco di gente che costruisce nuove cose su quel che hanno creato altri, come un muro di piccole pietre.*

Donald Knuth

Avete già visto i valori di funzione, come alert, e come richiamarli. Le funzioni sono il pane quotidiano del programmatore JavaScript. Il concetto di avvolgere un brano di programma in un valore ha grande utilità. Si tratta di uno strumento che consente di impostare programmi più vasti, ridurre le ripetizioni, associare dei nomi a dei programmi secondari e isolarli individualmente.

L'applicazione più ovvia per le funzioni è di definire un nuovo vocabolario. Con la lingua parlata, inventare nuovi termini non è una buona idea: ma quando si programma, ciò è indispensabile.

Gli adulti di discreta cultura hanno in media un vocabolario di circa 47.000 parole diverse. Ben pochi linguaggi di programmazione dispongono di 47.000 comandi integrati. E il vocabolario *a disposizione* ha in genere definizioni più precise e pertanto molto meno flessibili della lingua parlata. Pertanto, è normale *dover* ampliare il vocabolario aggiungendo nuove parole per evitare troppe ripetizioni.

### Definire una funzione

Definire una funzione è come definire una variabile, dove il valore assegnato alla variabile è una funzione. Per esempio, nel codice che segue, si definisce la variabile square per fare riferimento a una funzione che produce il quadrato di un numero dato:

---

```
var square = function(x) {  
    return x * x;  
};  
console.log(square(12));  
// → 144
```

---

La funzione si imposta con un'espressione che inizia con la parola chiave function.

Ogni funzione ha un certo numero di *parametri* (in questo caso, soltanto `x`) e un *corpo*, che contiene le dichiarazioni che saranno eseguite quando si richiama la funzione. Il corpo della funzione deve sempre essere inserito tra parentesi graffe, anche quando consiste di una sola dichiarazione (come nell'esempio precedente).

Una funzione può avere più parametri o anche nessuno. Nell'esempio che segue, la funzione `makeNoise` non riporta parametri, mentre per `power` se ne indicano due:

---

```
var makeNoise = function() {
  console.log("Pling!");
};

makeNoise();
// → Pling!

var power = function(base, exponent) {
  var result = 1;
  for (var count = 0; count < exponent; count++)
    result *= base;
  return result;
};
console.log(power(2, 10));
// → 1024
```

---

Alcune funzioni, come `power` e `square`, producono un valore; altre, come `makeNoise`, non producono valori ma effetti collaterali. La dichiarazione `return` determina il valore restituito dalla funzione. Quando incontra questa dichiarazione, la struttura di controllo balza fuori dalla funzione corrente e passa il valore restituito al codice che aveva richiamato la funzione. Quando la parola chiave `return` non è seguita da un'espressione, la funzione restituisce `undefined`.

## Parametri e ambiti di visibilità

I parametri delle funzioni si comportano come le variabili, ma i loro valori iniziali vengono assegnati dalla *chiamata* alla funzione e non dal codice della funzione stessa.

Un'importante proprietà delle funzioni è che le variabili create al loro interno, compresi i parametri, sono *locali* alla funzione. Ciò significa, per esempio, che la variabile `result` nel listato sopra riportato per `power` viene ricreata ogni volta che si richiama la funzione e queste "incarnazioni" separate non interferiscono tra loro.

La proprietà delle variabili di essere "locali" si applica soltanto ai parametri e alle variabili dichiarate con la parola chiave `var` all'interno del corpo della funzione. Le variabili dichiarate al di fuori delle funzioni si chiamano *globali*, perché sono disponibili per tutto il programma. È possibile accedere a quelle variabili dall'interno di una funzione,

a condizione che non abbiate una variabile locale con lo stesso nome.

Il codice che segue illustra questo concetto. Si definiscono e si richiamano due funzioni, ciascuna delle quali assegna un valore alla variabile `x`. La prima dichiara la variabile come locale e modifica pertanto solo i valori della variabile locale. La seconda non dichiara `x` come variabile locale; pertanto, i riferimenti a `x` all'interno di questa funzione sono in relazione alla variabile globale `x`, definita all'inizio dell'esempio.

---

```
var x = "fuori";
var f1 = function() {
    var x = "dentro f1";
}
f1();
console.log(x);
// → fuori
var f2 = function() {
    x = "dentro f2";
}
f2();
console.log(x);
// → dentro f2
```

---

Questo comportamento serve a prevenire interferenze accidentali tra funzioni. Se tutte le variabili fossero ugualmente condivise dal programma, diventerebbe impossibile avere la certezza di non usare lo stesso nome per scopi diversi. Se voleste *riutilizzare* il nome di una variabile, potrebbero verificarsi comportamenti strani quando altre parti del codice andassero a modificare il valore della variabile. Il vantaggio delle variabili locali alla funzione, che esistono solo e soltanto all'interno della funzione dove sono definite, è che ciascuna funzione rimane isolata dal resto del programma, come un piccolo universo a sé stante, senza che dobbiate preoccuparvi degli altri universi che la circondano.

## Ambiti di visibilità (scope) nidificati

JavaScript non distingue solo tra variabili *globali* e *locali*. Si possono creare funzioni all'interno di altre funzioni, perciò esistono diversi livelli di località.

Per esempio, questa funzione, per quanto poco significativa, contiene al suo interno due funzioni:

---

```
var paesaggio = function() {
    var result = "";
    var pianura = function(size) {
```

```

    for (var count = 0; count < size; count++)
        result += "_";
};

var montagna = function(size) {
    result += "/";
    for (var count = 0; count < size; count++)
        result += "'";
    result += "\\\";

};

pianura(3);
montagna(4);
pianura(6);
montagna(1);
pianura(1);
return result;
};

console.log(paesaggio());
// → __/_'''\_\_\_/_'\\_

```

---

Le funzioni `pianura` e `montagna` hanno accesso alla variabile `result`, in quanto sono all'interno della funzione che la definisce, ma non possono accedere alle rispettive variabili `count`, che sono al di fuori dei rispettivi ambiti di visibilità. L'ambiente al di fuori della funzione `paesaggio` non “vede” nessuna delle variabili definite al suo interno.

In breve, ogni ambito di visibilità locale ha accesso agli ambiti locali che lo contengono. L'insieme delle variabili accessibili dall'interno di una funzione dipende dalla posizione che questa ha nel programma. Le variabili definite nei blocchi intorno alla definizione di una variabile sono tutte visibili/accessibili: e questo vale sia per quelle nei corpi della funzione che le contiene, sia per quelle definite al livello principale del programma. Questo concetto di visibilità delle variabili si chiama *scoping (contesto) lessicale*.

Chi ha esperienza con altri linguaggi di programmazione probabilmente si aspetta che ogni blocco di codice tra parentesi graffe imposta un nuovo ambiente locale. Invece, in JavaScript, solo le funzioni creano un nuovo ambito di visibilità, anche se potete impostare blocchi di codice indipendenti:

```

var something = 1;
{
    var something = 2;
    // Fai qualcosa con la variabile something...
}

```

```
// Torna fuori dal blocco...
```

---

La variabile `something` all'interno del blocco è la stessa che vale al suo esterno. In effetti, questi blocchi di codice servono solo a raggruppare il corpo di una dichiarazione `if` o di un ciclo.

Se trovate strana questa impostazione, non siete gli unici. Nella versione successiva di JavaScript, viene introdotta una parola chiave `let`, simile a `var` ma che crea una variabile locale al blocco che la racchiude e non alla funzione che la contiene.

## Funzioni come valori

Le variabili funzione agiscono semplicemente da nomi per determinati parti del programma. Queste variabili si definiscono una volta e non cambiano mai, il che contribuisce a far confondere la funzione col suo nome.

Invece si tratta di due cose diverse. Un valore funzione può fare tutto quel che fanno gli altri valori: si può utilizzare in qualunque espressione e si può richiamare. Si può memorizzare un valore funzione in una nuova posizione, passarlo come argomento a un'altra funzione e così via. In parallelo, una variabile che contiene una funzione non è altro che una variabile alla quale si può assegnare un nuovo valore, come segue:

```
var launchMissiles = function(value) {  
    missileSystem.launch("adesso");  
};  
if (safeMode)  
    launchMissiles = function(value) /* non fa nulla */;
```

---

Nel [Capitolo 5](#), si vedrà che cosa si riesce a ottenere passando valori funzione ad altre funzioni.

## Notazione della dichiarazione

Esiste una scorciatoia per esprimere “`var square = function...`”. La parola chiave `function` si può usare anche all'inizio della dichiarazione, come nell'esempio seguente:

```
function square(x) {  
    return x * x;  
}
```

---

Questa è la forma che prende la *dichiarazione* di una funzione. L'istruzione definisce la variabile `square` e la fa puntare alla funzione data. Fin qui tutto chiaro. Esiste però un piccolo problema con questo modo di dichiarare una funzione.

```
console.log("Il futuro dice:", future());
```

```
function future() {  
    return "Continuiamo a NON avere automobili volanti.";  
}
```

---

Questo codice va bene, anche se la funzione è stata definita *al di sotto* del codice che la utilizza. Questo perché le dichiarazioni di funzione non seguono la solita struttura di controllo dall'alto al basso: nell'esecuzione del programma, vengono innalzate al massimo livello di accessibilità e possono essere utilizzate da tutto il codice a quel livello. Questa impostazione può essere utile, poiché ci consente di dare al codice un ordine sensato senza bisogno di definire tutte le funzioni prima di incontrarle (nel programma) per la prima volta.

Il problema nasce quando inserite la definizione della funzione in un blocco condizionale (`if`) o in un ciclo. Non fatelo. In quei casi, il comportamento delle piattaforme per JavaScript varia da browser a browser e da piattaforma a piattaforma; lo standard più recente proibisce quel tipo di sintassi. Per essere sicuri che i programmi siano eseguiti correttamente, questa sintassi per la dichiarazione di funzioni va seguita solo nel blocco più esterno della funzione o del programma.

---

```
function example() {  
    function a() {} // Va bene  
    if (something) {  
        function b() {} // Pericolo!  
    }  
}
```

---

## Pila delle chiamate

Conviene a questo punto esaminare come procede la struttura di controllo attraverso le funzioni. Ecco un semplice programma che effettua delle chiamate a funzioni:

---

```
function greet(who) {  
    console.log("Salve " + who);  
}  
greet("Luigi");  
console.log("Arrivederci");
```

---

Il programma viene eseguito come segue: la chiamata a `greet` trasferisce il controllo alla riga di inizio di quella funzione (riga 2). Si chiama ora `console.log`, una funzione integrata nei browser, che assume il controllo, svolge il suo lavoro e restituisce il controllo alla riga 2. Il flusso giunge poi al termine della funzione `greet`, tornando alla posizione dove ha avuto la prima chiamata, la riga 4. La riga seguente chiama di nuovo

```
console.log.
```

Possiamo rappresentare la struttura di controllo come segue:

---

```
top
  greet
    console.log
  greet
top
  console.log
top
```

---

Poiché, quando restituisce un valore, la funzione deve tornare indietro alla posizione dove era stata chiamata, il computer deve ricordare il contesto della chiamata. In un caso, `console.log` ritorna alla funzione `greet`. Nell'altro, torna alla fine del programma.

La posizione dove il computer memorizza questo contesto si chiama *stack* (pila) *delle chiamate*. Ogni volta che si chiama una funzione, il contesto corrente viene messo in cima a questa pila. Appena restituito il valore, la funzione toglie il contesto dalla cima e lo utilizza per continuare l'esecuzione.

Tenere in memoria lo stack richiede spazio. Quando la pila cresce troppo, il computer dà un errore di spazio “out of stack” o di eccesso di ricorsività. Il codice che segue illustra questa situazione: la domanda posta al computer è molto difficile e provoca un avanti-e-indietro infinito tra due funzioni. In effetti *sarebbe* infinito, se il computer avesse uno stack senza fine. Così com'è, a un certo punto resteremo senza spazio, o “faremo esplodere lo stack”.

---

```
function gallina() {
  return uovo();
}

function uovo() {
  return gallina();
}

console.log(gallina() + " venne prima.");
// → ??
```

---

## Argomenti facoltativi

Il codice seguente è ammesso e viene eseguito senza errori:

---

```
alert("Salve", "Buonasera", "Come va?");
```

---

La funzione `alert` accetta ufficialmente un solo argomento, ma non provoca errori quando la chiamate come nell'esempio: si limita a ignorare gli argomenti che seguono il primo e vi restituisce "Salve".

JavaScript è molto tollerante quando si tratta del numero di argomenti che passate a una funzione. Se ne passate troppi, vengono ignorati quelli in eccesso. Se ne passate troppo pochi, al parametro mancante viene assegnato il valore `undefined`.

Il rovescio della medaglia è che non sarete avvertiti quando accidentalmente, cosa che capita piuttosto spesso, passerete il numero di argomenti sbagliato.

Il vantaggio è che con questo comportamento si possono passare alle funzioni degli argomenti "facoltativi". Per esempio, la versione che segue di `power` si può chiamare con due o con un solo argomento; in questo caso, l'esponente viene letto come due e la funzione si comporta come il quadrato:

---

```
function power(base, exponent) {  
    if (exponent == undefined)  
        exponent = 2;  
    var result = 1;  
    for (var count = 0; count < exponent; count++)  
        result *= base;  
    return result;  
}  
console.log(power(4));  
// → 16  
console.log(power(4, 3));  
// → 64
```

---

Nel prossimo capitolo, vedremo come il corpo di una funzione può accedere alla lista esatta degli argomenti passati, per fare in modo che le funzioni accettino qualunque numero di argomenti. Nell'esempio che segue, `console.log` utilizza questa impostazione per produrre tutti i valori assegnati:

---

```
console.log("R", 2, "D", 2);  
// → R 2 D 2
```

---

## Chiusura

La possibilità di trattare le funzioni come valori, combinata col fatto che le variabili locali vengono "ricreate" ogni volta che si chiama una funzione, fa sorgere un interessante interrogativo: che cosa succede alle variabili locali quando la funzione che le ha create non è più attiva?

Un esempio è riportato nel codice seguente: si definisce una funzione, `wrapValue`, che crea una variabile locale. Il valore restituito è una funzione che accede a e restituisce la variabile locale.

---

```
function wrapValue(n) {  
    var localVariable = n;  
    return function() { return localVariable; };  
}  
  
var wrap1 = wrapValue(1);  
var wrap2 = wrapValue(2);  
console.log(wrap1());  
// → 1  
console.log(wrap2());  
// → 2
```

---

Il codice è corretto e funziona come volevate: è ancora possibile accedere alla variabile. In effetti, possono essere attive contemporaneamente più istanze della variabile: il che illustra bene il concetto che le variabili locali vengono davvero ricreate per ogni chiamata e nessuna chiamata può “disturbare” le variabili locali delle altre.

La possibilità di far riferimento a un’istanza specifica delle variabili locali in una funzione che le avvolge si chiama *chiusura* (closure). Una funzione che si chiude “sopra” una variabile locale viene definita una *chiusura*. Questo comportamento vi libera dal dovervi preoccupare della durata delle variabili e vi offre degli usi creativi per i valori funzione.

Con una leggera modifica, possiamo trasformare l’esempio precedente in modo da creare delle funzioni che moltiplicano qualunque cifra.

---

```
function multiplier(factor) {  
    return function(number) {  
        return number * factor;  
    };  
}  
  
var twice = multiplier(2);  
console.log(twice(5));  
// → 10
```

---

La variabile esplicita `localVariable` dell’esempio `wrapValue` non è necessaria, in quanto i parametri sono di per sé variabili locali.

Programmare in questo modo richiede un po’ di pratica. Come modello mentale, si può pensare alla parola chiave `function` come a un modo per “congelare” il codice all’interno

del suo corpo e avvolgerlo in un pacchetto (il valore funzione). Così, quando si trova `return function(...){...}` è come ottenere la maniglia di un pezzo di programma che si era “congelato” per uso futuro.

Nell'esempio, `multiplier` restituisce un blocco di codice che viene memorizzato nella variabile `twice`. In ultima riga, si chiama il valore di questa variabile (`return number * factor;`), attivando il codice congelato. Questo ha ancora accesso alla variabile `factor` dalla chiamata a `multiplier` che l'aveva creato; inoltre, attraverso il suo parametro `number`, ha accesso all'argomento passato quando è stato scongelato: 5.

## Ricorsività

Le funzioni possono richiamare se stesse, purché non facciano saltare lo stack. Una funzione che richiama se stessa si chiama *ricorsiva*. La ricorsività consente di impostare alcune funzioni in maniera diversa dal solito. Prendete per esempio la seguente forma di definizione di `power`:

---

```
function power(base, exponent) {  
    if (exponent == 0)  
        return 1;  
    else  
        return base * power(base, exponent - 1);  
}  
console.log(power(2, 3));  
// → 8
```

---

Questa sintassi segue abbastanza fedelmente la definizione matematica dell'innalzamento a potenza ed esprime il concetto in modo forse più elegante di quello offerto dai cicli. La funzione richiama se stessa più volte, con diversi argomenti, per ottenere la ripetizione della moltiplicazione.

Questa sintassi, però, presenta un problema: in JavaScript, arriva a essere dieci volte più lenta della versione ciclica. Questo perché l'esecuzione di un ciclo semplice è meno onerosa delle chiamate multiple a una funzione.

Scegliere tra velocità ed eleganza è un bel dilemma: in un certo senso, rappresenta il conflitto tra “va bene per noi” e “va bene per il computer”. Moltissimi programmi possono essere resi più veloci in esecuzione aggiungendo complicazione e codice: chi programma deve scegliere fino a che punto ciò sia ragionevole.

Con l'esempio riportato in precedenza per la funzione `power`, la versione ciclica, seppur poco elegante, rimane semplice e facile da leggere. Non ha molto senso sostituirla con una versione ricorsiva. Tuttavia, in altri casi il programma deve prevedere aspetti talmente complessi, che può convenire rinunciare a un po' di efficienza per semplificarne la lettura.

La regola di base, alla quale si attiene la maggioranza dei programmati, e con cui concordo assolutamente, è di non preoccuparsi dell'efficienza del programma fintanto che non si ha la certezza che il programma sia in effetti troppo lento. Quando questo succede, scoprite quali sono le parti del programma che impiegano più tempo e, per quelle situazioni, rinunciate all'eleganza in favore dell'efficienza.

Naturalmente questo non significa che bisogna sempre e comunque ignorare l'efficienza del programma. In molti casi, come per la funzione power, non si ottiene un granché scegliendo la strada dell'eleganza. E a volte, anche i programmati più esperti devono ammettere che la strada più semplice non sarà mai la più veloce.

Se sottolineo questi concetti, è perché sono molti i programmati alle prime armi che si preoccupano, in maniera quasi fanatica, dell'efficienza anche nei più piccoli particolari. In questi casi, i programmi sono sempre più grandi e complicati e, spesso, anche meno accurati; oltre a essere più lunghi da scrivere di versioni equivalenti formalmente meno accurate, non è detto che siano eseguiti in maniera significativamente più veloce.

La ricorsività, comunque, non è sempre un'alternativa meno efficiente dei cicli. Alcuni problemi, anzi, si risolvono più semplicemente con la ricorsività. In genere, si tratta di problemi che richiedono di analizzare o elaborare diversi "rami", ciascuno dei quali può essere ulteriormente ramificato.

Considerate per esempio questo problema: partendo dal numero 1 e aggiungendo 5 o moltiplicando per 3, si ottiene un numero infinito di numeri (risultati). Come scrivereste una funzione che, dato un numero, trova la sequenza di addizioni e moltiplicazioni che lo producono? Per esempio, al numero 13 si potrebbe arrivare prima moltiplicando per 3 e poi aggiungendo due volte 5; ma quella sequenza non serve per arrivare al numero 15.

La soluzione ricorsiva si potrebbe scrivere come segue:

---

```
function findSolution(target) {
    function find(start, history) {
        if (start == target)
            return history;
        else if (start > target)
            return null;
        else
            return find(start + 5, "(" + history + " + 5)") ||
                   find(start * 3, "(" + history + " * 3)");
    }
    return find(1, "1");
}
console.log(findSolution(24));
// → (((1 * 3) + 5) * 3)
```

---

Notate che il programma non trova necessariamente la sequenza di operazioni *più breve*: basta infatti che trovi una sequenza qualunque.

Quest'esempio potrebbe non essere immediatamente comprensibile. Esaminiamolo passo per passo, perché è un ottimo esercizio per abituarsi a ragionare ricorsivamente.

La funzione interna `find` svolge il calcolo ricorsivo. Accetta due argomenti: il numero corrente e una stringa che memorizza il percorso seguito per arrivare al numero. Il valore restituito è o una stringa che mostra il percorso, o il valore `null`.

Per arrivare a questo, la funzione compie una di tre azioni possibili. Se il numero corrente è quello cercato, il valore corrente di `history` è la strada per arrivare a destinazione e basta restituire quel valore. Se il numero corrente è maggiore di quello cercato, non ha senso continuare perché qualunque nuova operazione lo renderà solo più grande. Infine, se non si è ancora arrivati al numero, la funzione prova entrambe le strade che partono dal numero corrente e richiama se stessa due volte, una per ciascuno dei passi consentiti. Se la prima chiamata restituisce un valore diverso da `null`, restituisce quel valore. Altrimenti, restituisce il valore della seconda chiamata, sia che si tratti di una stringa, sia che si tratti di un valore `null`.

Per capire meglio come la funzione produce l'effetto desiderato, esaminiamo le chiamate a `find` effettuate per trovare una soluzione per il numero 13:

---

```
find(1, "1")
find(6, "(1 + 5)
find(11, "((1 + 5) + 5)")
find(16, "(((1 + 5) + 5) + 5)")
    troppo grande
find(33, "(((1 + 5) + 5) * 3)")
    troppo grande
find(18, "((1 + 5) * 3)")
    troppo grande
find(3, "(1 * 3)")
find(8, "((1 * 3) + 5)")
find(13, "(((1 * 3) + 5) + 5)")
    trovato!
```

---

I rientri dal margine danno un'indicazione visiva della profondità della pila. La prima volta che si chiama `find`, la funzione si richiama due volte per esplorare le soluzioni che iniziano con  $(1 + 5)$  e  $(1 * 3)$ . La prima di queste chiamate cerca una soluzione che inizia con  $(1 + 5)$  e continua ricorsivamente, esplorando *tutte* le soluzioni che portano a un numero uguale o minore del numero dato. Poiché non trova una soluzione adatta, restituisce `null` alla prima chiamata. Qui, l'operatore `||` fa scattare la chiamata che esplora la strada di  $(1 * 3)$ . Questo dà un risultato migliore: infatti, dopo la prima

chiamata ricorsiva, *un'altra* chiamata ricorsiva trova il numero dato, 13. Quest'ultima chiamata restituisce una stringa, che viene passata dagli operatori || delle chiamate intermedie e fornisce la soluzione.

## Funzioni che crescono

Esistono due sistemi, più o meno naturali, per introdurre delle funzioni nei programmi.

Il primo è che vi troviate a scrivere più e più volte del codice molto simile. Cosa che vorremmo evitare, perché far crescere il codice significa avere più spazio per gli errori e più pagine da leggere per chi cerca di capire il programma. Prenderemo dunque la soluzione ripetuta, le assegneremo un nome adeguato e la inseriremo in una funzione.

Il secondo sistema è che scopriate di aver bisogno di funzionalità che non avete ancora scritto, ma che hanno buone probabilità di meritare una propria funzione. In questi casi, per prima cosa scegliete il nome della funzione e poi impostatene il corpo. In alcune situazioni, potreste trovarvi a scrivere del codice che utilizza la funzione ancor prima di averla chiaramente definita.

Tanto più oscuro è il concetto che state tentando di programmare, tanto più sarà difficile trovare il nome per la funzione. Vediamo un esempio.

Immaginiamo di dover scrivere un programma che stampa due numeri, la quantità di vacche e di polli di un'ipotetica fattoria. I numeri devono essere seguiti rispettivamente dalle parole vacche e Polli, e vogliamo che siano sempre di tre cifre, imbottendoli con degli zeri iniziali quando il numero è inferiore a 100.

---

007 Vacche

011 Polli

---

Chiaramente, ci vuole una funzione con due argomenti. Per esempio, la seguente:

```
function printFarmInventory(vacche, polli) {  
    var cowString = String(vacche);  
    while (cowString.length < 3)  
        cowString = "0" + cowString;  
    console.log(cowString + " Vacche");  
    var chickenString = String(polli);  
    while (chickenString.length < 3)  
        chickenString = "0" + chickenString;  
    console.log(chickenString + " Polli");  
}  
printFarmInventory(7, 11);
```

---

Aggiungendo .length dopo un valore di tipo string si ottiene la lunghezza della

stringa. Pertanto, i cicli while continuano ad aggiungere zeri davanti alle stringhe col numero, finché sono lunghe almeno tre caratteri.

Missione compiuta! Peccato che, quando abbiamo il programma pronto per essere consegnato, il fattore ci chiami comunicandoci di aver cominciato ad allevare anche maiali e il programma va esteso per prevedere anche questo articolo.

Potremmo pensare di risolvere il problema semplicemente copiando e incollando ancora una volta le quattro righe di codice che servono, ma ci dev'essere un sistema migliore. Per esempio, si potrebbe scrivere:

---

```
function printZeroPaddedWithLabel(number, label) {  
    var numberString = String(number);  
    while (numberString.length < 3)  
        numberString = "0" + numberString;  
    console.log(numberString + " " + label);  
}  
  
function printFarmInventory(vacche, polli, maiali) {  
    printZeroPaddedWithLabel(vacche, "Vacche");  
    printZeroPaddedWithLabel(polli, "Polli");  
    printZeroPaddedWithLabel(maiali, "Maiali");  
}  
  
printFarmInventory(7, 11, 3);
```

---

Anche questo funziona, ma il nome di `printZeroPaddedWithLabel` non è il massimo della chiarezza, poiché comprime tre operazioni, stampare, imbottire di zeri e aggiungere un'etichetta, in una sola funzione.

Invece di ripetere l'intera parte del nostro programma, proviamo a isolare un singolo *conetto*:

---

```
function zeroPad(number, width) {  
    var string = String(number);  
    while (string.length < width)  
        string = "0" + string;  
    return string;  
}  
  
function printFarmInventory(vacche, polli, maiali) {  
    console.log(zeroPad(vacche, 3) + " Vacche");  
    console.log(zeroPad(polli, 3) + " Polli");  
    console.log(zeroPad(maiali, 3) + " Maiali");  
}
```

```
printFarmInventory(7, 16, 3);
```

---

Una funzione che si chiama `zeroPad` (imbottisci di zeri) semplifica la lettura del programma e aiuta a capire che cosa fa. E può essere utile anche in altre situazioni, oltre che in questo programma specifico. Per esempio, potete usarla anche per stampare delle tabelle di numeri allineandole ordinatamente.

Possiamo impostare funzioni di moltissimi gradi di complessità: dalle più semplici, che si limitano a imbottire un numero di zeri, fino a sistemi di formattazione numerica complicatissimi, in grado di gestire decimali, numeri negativi, allineamento, riempimento a sinistra con caratteri diversi e così via.

Tenete sempre a mente che non conviene aggiungere troppi livelli di complicazione finché non si è certi di averne bisogno. Evitate la tentazione di scrivere delle “strutture” per tutte le funzionalità che incontrate: rischiate di combinare ben poco, al di là della mole di codice che si produrrebbe e che nessuno userà mai.

## Funzioni ed effetti collaterali

Le funzioni si possono dividere grossomodo in due gruppi: da una parte, quelle che danno effetti collaterali; dall’altra, quelle che restituiscono dei valori. E questo, anche se è sicuramente possibile avere contemporaneamente degli effetti collaterali e dei valori di restituzione.

La prima versione della funzione nell’esempio della fattoria, `printZeroPaddedWithLabel`, viene chiamata per un effetto collaterale: quello di stampare una riga. La seconda versione, `zeroPad`, viene chiamata per il valore di restituzione. Non è casuale che la seconda versione abbia più applicazioni della prima. Le funzioni che danno valori sono più semplici da combinare in nuovi modi e riutilizzare delle funzioni che provocano degli effetti collaterali.

Una funzione *pura* è un tipo specifico di funzione che produce un valore, che non solo non ha effetti collaterali, ma non ha nemmeno bisogno di riceverli da altre parti del codice. Inoltre, non legge eventuali variabili globali che fossero modificate da altre parti del codice. Una funzione pura ha la proprietà speciale di produrre sempre lo stesso valore (senza fare altro) tutte le volte che viene richiamata con gli stessi argomenti. Pertanto, ogni chiamata a questo tipo di funzione può essere sostituita dal suo risultato senza che il codice cambi significato. Per verificare che una funzione pura operi correttamente, richiamatela in un qualunque contesto; se funziona lì, si sa che opererà ovunque. Le funzioni non pure possono invece restituire risultati differenti a seconda di diversi fattori e avere effetti collaterali difficili da rilevare.

Con tutto questo, anche le funzioni non pure hanno un loro ambito di applicazione. Gli effetti collaterali possono essere di grande utilità nei programmi: non c’è modo, per esempio, di scrivere una versione pura di `console.log`, benché `console.log` abbia utili applicazioni. Anche in altri casi può essere più efficiente far uso di effetti collaterali per esprimere alcune operazioni, come per esempio quando la velocità di calcolo è più

importante della purezza del codice.

## Riepilogo

In questo capitolo si è visto come impostare le funzioni. La parola chiave `function`, usata come espressione, può creare un valore funzione. Quando invece viene usata come dichiarazione, serve per dichiarare una variabile e assegnarle come valore una funzione.

---

```
// Crea un valore funzione f
var f = function(a) {
    console.log(a + 2);
};

// Dichiara g come funzione
function g(a, b) {
    return a * b * 3.5;
}
```

---

Un aspetto centrale quando si parla di funzioni sta nel comprendere il concetto di contesto locale. Parametri e variabili definiti all'interno di una funzione sono locali alla funzione, sono ri-creati ogni volta che la si chiama e non sono visibili all'esterno. Una funzione dichiarata all'interno di un'altra funzione ha accesso al contesto locale della funzione esterna.

Separate sempre le azioni svolte dal programma in funzioni diverse. In questo modo eviterete ripetizioni eccessive. Inoltre, quando il codice è raggruppato in brani di chiaro significato, il programma risulta più semplice da leggere, proprio come è più facile leggere un manuale quando il testo è organizzato in capitoli e paragrafi.

## Esercizi

### *Minimo*

Nel capitolo precedente si è accennato alla funzione standard `Math.min`, che restituisce il minore degli argomenti passati. Provate ora a scrivere una funzione `min` che accetti due argomenti e restituisca il minore.

### *Ricorsività*

Abbiamo visto che l'operatore resto (%) può essere utilizzato per stabilire se un numero è pari o dispari, impostando `% 2` per verificare se il numero è divisibile per due. Ecco un altro modo per definire se un numero intero positivo è pari o dispari:

- Zero è pari.

- Uno è dispari.
- Qualunque altro numero  $N$  è pari, se è pari il risultato di  $N - 2$ .

Definite una funzione ricorsiva `isEven` in base alle informazioni date. La funzione deve accettare un parametro di tipo numerico e restituire un valore di tipo booleano.

Provate il programma con 50 e 75. Osservate come si comporta con con -1. Perché? Riuscite a trovare una soluzione al problema?

## Conteggi

Per ricavare l' $N^{\text{mo}}$  carattere (o lettera) di una stringa, si può scrivere `"stringa".charAt(N)`, un po' come ne ottenete la lunghezza con `"s".length`. Il valore restituito deve essere una stringa di un solo carattere (per esempio, "b"). Poiché il primo carattere si trova nella posizione 0, l'ultimo si trova nella posizione `string.length - 1`. In altre parole, una stringa di due caratteri ha lunghezza 2 e i due caratteri si trovano rispettivamente in posizione 0 e 1.

Scrivete una funzione `countB`, che accetti la stringa come unico argomento e restituisca un numero che indichi quante lettere B maiuscole si trovano nella stringa.

Scrivete poi una funzione `countChar`, che si comporti come `countB`, ma accetti un secondo argomento che indichi il carattere da contare (invece di conteggiare soltanto le B maiuscole). Riscrivete `countB` in modo che faccia uso di questa nuova funzione.

## STRUTTURE DATI : OGGETTI E ARRAY

*In due occasioni mi hanno chiesto, ‘Dica, signor Babbage, se inserisce nella macchina le cifre sbagliate, ne usciranno risposte giuste?’ [...] Non riesco a capacitarmi della confusione di idee che potrebbe provocare tale domanda.*

Charles Babbage, *Passages from the Life of a Philosopher* (1864)

Numeri, valori booleani e stringhe sono i mattoni che servono per costruire le strutture dati, ma con un solo mattone non si erige una gran casa. Gli *oggetti* consentono di mettere insieme valori e altri oggetti, per costruire strutture più complesse.

I programmi degli esempi riportati finora sono gravemente limitati dal fatto che utilizzano solo tipi di dati semplici. In questo capitolo, a tali concetti si aggiungono gli elementi che servono per capire le strutture dati: con questi nuovi elementi, potrete cominciare a scrivere dei programmi davvero utili.

Il capitolo parte dall'illustrazione di un esempio di programmazione (più o meno realistico) per descrivere questi nuovi concetti man mano che si presentano. Il codice di esempio utilizza versioni più sofisticate delle funzioni e delle variabili trattate nei capitoli precedenti.

La piattaforma online per gli esempi di codice riportati nel libro (<http://eloquentjavascript.net/code/>) consente di eseguire il codice nel contesto specifico di ogni capitolo. Se voleste esercitarvi con gli esempi in ambienti diversi, assicuratevi di scaricare i listati completi dal sito.

### Lo scoiattolo mannaro

Ogni tanto, di solito tra le otto e le dieci di sera, Jacques si trasforma in un piccolo roditore dalla folta coda.

Da un lato, Jacques è contento di non soffrire di licantropia classica. Trasformarsi in uno scoiattolo gli crea meno problemi che diventare un lupo. Certo, non ha la preoccupazione di poter sbranare (per sbaglio) il vicino, il che sarebbe poco carino; in compenso, potrebbe essere divorato dal gatto del vicino. Da quando si è svegliato abbarbicato su un ramo in cima a una quercia, nudo e disorientato, ogni notte si impone di chiudere a chiave porta e finestra della camera e lascia un paio di noci sul pavimento casomai gli venisse fame.



Con questo, i problemi di gatto e quercia sono risolti, ma Jacques non è guarito. Il fatto che la trasformazione avvenga in modo irregolare gli fa pensare che sia causata da un qualche evento. Che cosa sarà a provocarla? All'inizio, aveva pensato che la trasformazione avvenisse solo se durante il giorno avesse toccato degli alberi, ma il problema continuò a ripresentarsi anche quando smise di toccare gli alberi e persino di avvicinarsi a essi.

Jacques decide pertanto di affrontare il problema in modo più scientifico, tenendo un diario dove annotare tutto quel che succede durante il giorno e se la sera ha luogo la trasformazione. Con questi dati, spera di isolare le condizioni che fanno scattare la metamorfosi.

Per prima cosa, imposta una struttura dati, dove registrare queste informazioni.

## Insiemi di dati

Per elaborare dei dati digitali, bisogna innanzitutto trovare un modo per rappresentarli nella memoria della macchina. Un esempio molto semplice è quando si vuole rappresentare una serie di numeri: 2, 3, 5, 7 e 11.

Si potrebbe ricorrere a una stringa: in fondo, le stringhe possono essere lunghe quanto si vuole e possono pertanto accogliere un sacco di dati. In questo caso, la rappresentazione dei dati avrebbe la forma "2 3 5 7 11". Questa soluzione, però, ha qualche problema: per avere accesso ai numeri, infatti, dovete prima trovare il modo per estrarli e riconvertirli in numeri.

Per fortuna, JavaScript mette a disposizione un tipo di dati specifico per memorizzare delle sequenze di valori: *l'array* o vettore. Gli array si scrivono come elenchi di valori, separati da virgole e inseriti tra parentesi quadre:

```
var listOfNumbers = [2, 3, 5, 7, 11];
console.log(listOfNumbers[1]);
// → 3
```

```
console.log(listOfNumbers[1 - 1]);  
// → 2
```

---

Anche la notazione per accedere agli elementi di un array utilizza le parentesi quadre. Una coppia di parentesi quadre, che contenga un'espressione e venga immediatamente dopo un'altra espressione, indica al programma di andare a cercare, nell'espressione di sinistra, l'elemento che corrisponde alla posizione specificata dall'espressione tra parentesi quadre.

La prima posizione in un array è zero, non uno. Il primo elemento si ricava pertanto con `listOfNumbers[0]`. Questa convenzione può sembrarvi strana se avete poca esperienza di programmazione, ma i conteggi da zero hanno una lunga tradizione (e un loro significato) in ambito tecnologico, purché siano applicati rigorosamente, come del resto si fa in JavaScript.

## Proprietà

In esempi precedenti, abbiamo incontrato espressioni come `myString.length` (per recuperare la lunghezza di una stringa) e `Math.max` (la funzione che recupera il valore più alto). Queste espressioni accedono a una *proprietà* di un dato valore. Nel primo caso, accediamo alla proprietà `length` del valore contenuto in `myString`. Nel secondo, accediamo alla proprietà `max` dell'oggetto `Math` (che è una collezione di funzioni e valori matematici).

In JavaScript, tutti i valori hanno delle proprietà, con la sola esclusione di `null` e `undefined`. Se tentate di accedere a una proprietà di uno di quei non-valori, ricevete un errore.

```
null.length;  
// → TypeError: Cannot read property 'length' of null  
// (errore di tipo: proprietà 'length' di null non accessibile)
```

---

I due modi più comuni per accedere alle proprietà in JavaScript utilizzano il punto e le parentesi quadre. Sia `value.x`, sia `value[x]` accedono a una proprietà di `value`, ma non necessariamente alla stessa. Tutto dipende da come viene interpretato `x`. Quando si usa il punto, la parte che lo segue deve essere un nome di variabile valido e riferirsi direttamente alla proprietà. Con le parentesi quadre, invece, il nome della proprietà viene *ricavato* dall'espressione tra parentesi. Mentre `value.x` recupera la proprietà "x" di `value`, `value[x]` tenta di calcolare l'espressione `x` e ne utilizza il risultato come nome della proprietà.

Pertanto, se sapete che la proprietà cercata si chiama "length", potete scrivere `value.length`. Se invece volette estrarre la proprietà il cui nome sta nel valore memorizzato nella variabile `i`, dovete scrivere `value[i]`. Poiché i nomi delle proprietà possono essere stringhe qualunque, se volette accedere a una proprietà che si chiama "0" o

“Mario Bianchi” dovete scegliere la notazione con le parentesi quadre: `value[0]` or `value["Mario Bianchi"]`. Questo anche se conoscete il nome della proprietà, perché né “0”, né “Mario Bianchi” sono nomi di variabile validi e pertanto non possono essere recuperati attraverso la notazione col punto.

Gli elementi degli array sono memorizzati in proprietà. Poiché i nomi di queste proprietà sono numeri, in generale dobbiamo ottenerne il nome da una variabile: pertanto, per accedervi dobbiamo utilizzare la sintassi con le parentesi quadre. La proprietà `length` di un array ne indica il numero di elementi. Il nome di questa proprietà è un nome di variabile valido, che conosciamo a priori. Pertanto, per trovare la proprietà `length` di un array, è di solito sufficiente che scriviate `array.length`, che è anche più facile da scrivere di `array["length"]`.

## Metodi

Gli oggetti di tipo `string` e `array` dispongono, oltre a `length`, di altre proprietà che fanno riferimento a valori funzione.

---

```
var doh = "Doh";
console.log(typeof doh.toUpperCase());
// → funzione
console.log(doh.toUpperCase());
// → DOH
```

---

Tutte le stringhe hanno una proprietà `toUpperCase`, che restituisce una copia della stringa dove tutte le lettere sono state convertite in maiuscole. Esiste anche una proprietà `toLowerCase`, che svolge l’operazione inversa.

Anche se la chiamata a `toUpperCase` non contiene argomenti, la funzione accede alla stringa “Doh”, ossia il valore di cui cerchiamo la proprietà. La logica di questo concetto è illustrata nel [Capitolo 6](#).

Le proprietà che contengono funzioni si chiamano *metodi* del valore a cui appartengono: per esempio, “`toUpperCase` è un metodo di un valore `stringa`”.

In questo esempio si dimostrano alcuni metodi degli oggetti `array`:

---

```
var mack = [];
mack.push("Mack");
mack.push("the", "Knife");
console.log(mack);
// → ["Mack", "the", "Knife"]
console.log(mack.join(" "));
// → Mack the Knife
```

```
console.log(mack.pop());
// → Knife
console.log(mack);
// → ["Mack", "the"]
```

---

Il metodo `push` consente di aggiungere valori alla fine di un array. Il metodo `pop` svolge il lavoro contrario: elimina il valore alla fine dell'array e lo restituisce. Col metodo `join`, si può ridurre a un'unica stringa un array di più stringhe. L'argomento passato a `join` determina il testo “incollato” tra gli elementi del vettore.

## Oggetti

Tornando all'esempio dello scoiattolo mannaro, si nota che una serie di appunti, nel diario di Jacques, può essere rappresentata come un array. Gli appunti, però, non sono costituiti solo da numeri o da stringhe: servono, infatti, per registrare attività e valori booleani che indicano se Jacques si è trasformato in scoiattolo. Idealmente, dovremmo raggruppare questi valori in un valore singolo, da inserire in un array per tutte le pagine del diario.

I valori di tipo *oggetto* sono collezioni arbitrarie di proprietà, che possiamo aggiungere o eliminare a piacere. Per creare un oggetto, si può usare una notazione tra parentesi graffe:

```
var day1 = {
  squirrel: false,
  events: ["work", "touched tree", "pizza", "running",
            "television"]
};

console.log(day1.squirrel);
// → false

console.log(day1.wolf);
// → undefined

day1.wolf = false;
console.log(day1.wolf);
// → false
```

---

Dentro le parentesi graffe, possiamo specificare elenchi di proprietà, separati da virgolette. Ciascuna proprietà è rappresentata col suo nome, seguito da un segno di due punti e da un'espressione che le assegna un valore. Spazi e a capo sono ignorati. Quando un oggetto occupa più righe, i rientri a margine, come nell'esempio, migliorano la leggibilità del codice. I nomi di proprietà che non sono nomi di variabile validi vanno inseriti tra virgolette.

---

```
var descriptions = {
  work: "Sono andato a lavorare",
  "touched tree": "Ho toccato un albero"
};
```

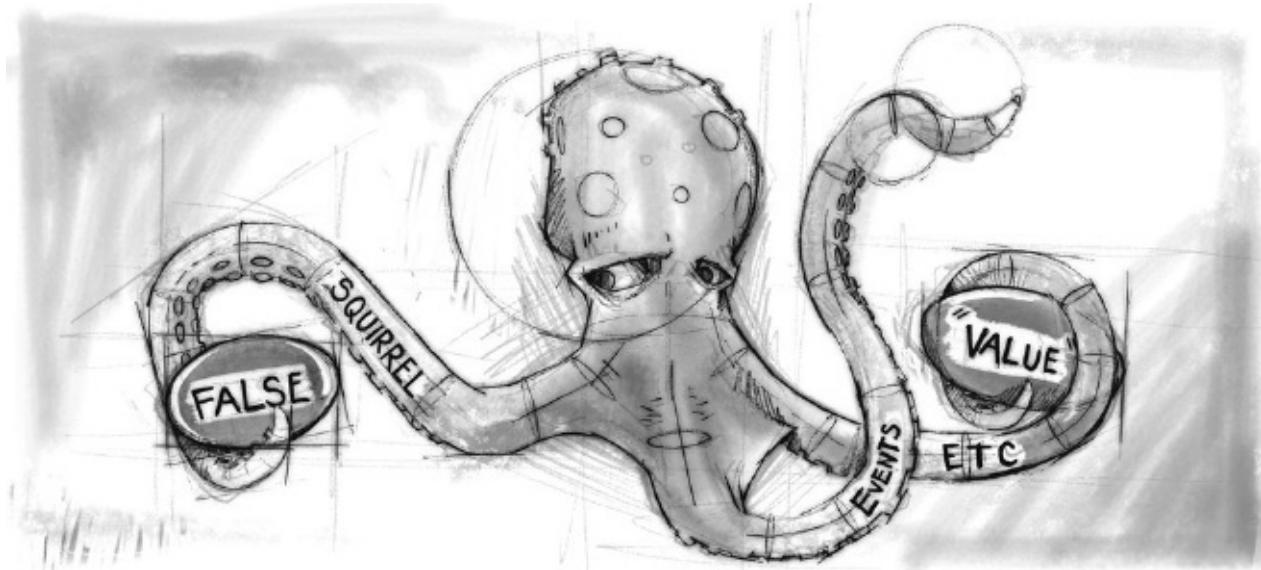
---

Ciò significa che le parentesi graffe hanno *due* significati in JavaScript. All'inizio di una dichiarazione, aprono un blocco di dichiarazioni. In qualunque altra posizione, descrivono un oggetto. Per fortuna, raramente c'è bisogno di iniziare una dichiarazione con un oggetto, il che evita ambiguità.

Quando si tenta di leggere una proprietà che non esiste, si ottiene come risultato il valore `undefined`: questo è quel che accade quando cerchiamo di leggere la proprietà `wolf` nell'esempio precedente.

Con l'operatore `=`, si assegna un valore all'espressione di una proprietà. Se la proprietà aveva già un valore, questo lo sostituisce; altrimenti si crea una nuova proprietà dell'oggetto.

I collegamenti tra proprietà ci riportano all'esempio delle variabili come tentacoli: le proprietà *catturano* valori, anche quando altre variabili e altre proprietà fossero collegate agli stessi valori. Gli oggetti sono come polipi che possono avere qualunque numero di tentacoli, ciascuno con un suo nome.



L'operatore `delete` tronca un tentacolo. Si tratta di un operatore unario che, applicato all'espressione di accesso di una proprietà, la elimina dall'oggetto. Anche se non comune, questa può essere un'esigenza di programmazione.

---

```
var anObject = {left: 1, right: 2};
console.log(anObject.left);
// → 1
delete anObject.left;
console.log(anObject.left);
```

```
// → undefined
console.log("left" in anObject);
// → false
console.log("right" in anObject);
// → true
```

---

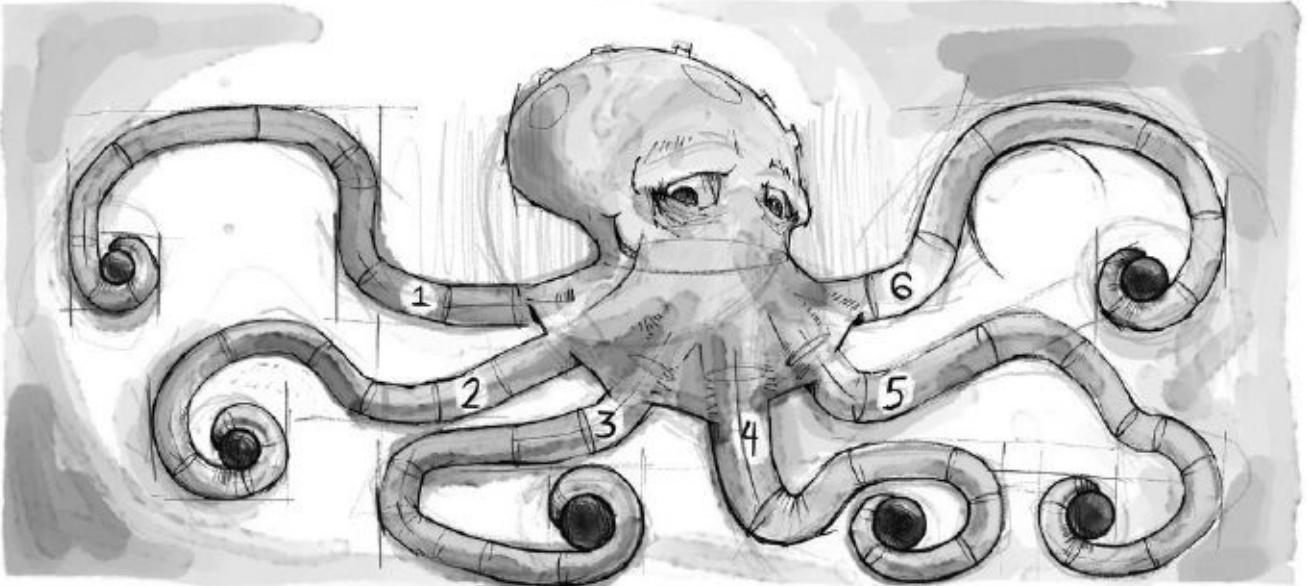
L'operatore binario `in`, applicato a una stringa e a un oggetto, restituisce un valore booleano che indica se quell'oggetto ha quella proprietà. La differenza tra impostare una proprietà su `undefined` ed eliminarla è che, nel primo caso, l'oggetto *mantiene* quella proprietà (sebbene con un valore poco interessante); nel secondo caso, la proprietà non è più presente e `in` restituirà `false`.

Gli array sono pertanto dei tipi speciali di oggetto, che servono proprio per memorizzare delle sequenze di dati. Se calcolate il risultato di `typeof [1, 2]`, ottenete "object": un tipo di polipo con tentacoli lunghi, piatti, messi tutti in fila, ciascuno col suo numero.

Anche il diario di Jacques può essere rappresentato come un array di oggetti:

```
var journal = [
  {events: ["work", "touched tree", "pizza",
            "running", "television"],
   squirrel: false},
  {events: ["work", "ice cream", "cauliflower",
            "lasagna", "touched tree", "brushed teeth"],
   squirrel: false},
  {events: ["weekend", "cycling", "break",
            "peanuts", "beer"],
   squirrel: true},
  /* e così via... */
];
```

---



## Mutabilità

Prima di passare alla programmazione *vera e propria*, dobbiamo comprendere un ultimo concetto di teoria.

Abbiamo visto come sia possibile modificare i valori degli oggetti. I tipi di valore discussi nei capitoli precedenti (numeri, stringhe, booleani) sono tutti *immutabili*: non è possibile modificare un valore esistente di quei tipi. Potete combinare tipi diversi per derivarne nuovi valori, ma il valore, per esempio, di una certa stringa rimarrà sempre lo stesso: il testo al suo interno non si può cambiare. Se avete un riferimento a una stringa che contenga "gatto", nessun altro codice potrà modificare i caratteri per trasformarla in "ratto".

Con gli oggetti, invece, è *possibile* cambiare il contenuto di un valore modificandone le proprietà.

Questo perché due numeri, per esempio 120 e 120, possono essere considerati esattamente lo stesso numero anche se non si riferiscono agli stessi bit fisici. Con gli oggetti, però, avere due riferimenti allo stesso oggetto non è lo stesso che avere due oggetti distinti contenenti le stesse proprietà. Prendete per esempio il seguente codice:

---

```
var object1 = {value: 10};
var object2 = object1;
var object3 = {value: 10};
console.log(object1 == object2);
// → true
console.log(object1 == object3);
// → false
object1.value = 15;
console.log(object2.value);
```

```
// → 15  
console.log(object3.value);  
// → 10
```

---

Le variabili `object1` e `object2` accedono allo stesso oggetto, il che spiega perché se si modifica `object1` cambia anche il valore di `object2`. La variabile `object3` punta a un altro oggetto, che inizialmente contiene le stesse proprietà di `object1`, ma vive di vita propria.

Nella comparazione tra oggetti, l'operatore `==` restituisce `true` solo se entrambi gli oggetti sono esattamente lo stesso valore. Confrontando oggetti diversi, il risultato sarà `false`, anche se i rispettivi contenuti fossero identici. JavaScript non offre operatori di confronto approfondito, ossia che esaminino i contenuti di un oggetto; tuttavia potete scrivere del codice per ottenere quella funzionalità, come illustra uno degli esercizi alla fine del capitolo.

## Il diario dello scoiattolo mannaro

A questo punto, Jacques lancia l'interprete JavaScript e imposta l'ambiente che gli servirà per tenere il diario.

---

```
var journal = [];  
function addEntry(events, didITurnIntoASquirrel) {  
    journal.push({  
        events: events,  
        squirrel: didITurnIntoASquirrel  
    });  
}
```

---

Ogni sera alle dieci, o la mattina dopo essere sceso dall'ultimo ripiano della libreria, prende nota degli avvenimenti della giornata appena conclusa.

---

```
addEntry(["work", "touched tree", "pizza", "running",  
         "television"], false);  
addEntry(["work", "ice cream", "cauliflower", "lasagna",  
         "touched tree", "brushed teeth"], false);  
addEntry(["weekend", "cycling", "break", "peanuts",  
         "beer"], true);
```

---

Una volta raccolta una quantità sufficiente di dati, Jacques calcolerà la correlazione tra le sue trasformazioni in scoiattolo e ciascuno degli eventi della giornata, nella speranza di imparare qualcosa di utile.

La correlazione è una misura di dipendenza tra variabili (“variabili” in senso statistico, non nel senso di JavaScript). Viene di solito espressa come coefficiente, in una scala che va da -1 a 1. Una correlazione di 0 indica che le variabili non sono in relazione tra loro, mentre una relazione di 1 indica una correlazione perfetta tra le due: conoscendone una, conoscete anche l’altra. Il valore negativo -1 indica che le variabili sono in correlazione perfetta, ma una è il contrario dell’altra: quando una è `true`, l’altra è `false`.

Per le variabili binarie (booleans), il coefficiente phi ( $\varphi$ ) offre una buona misura di correlazione ed è relativamente facile da calcolare. Per calcolare  $\varphi$ , dobbiamo impostare una tabella n che descriva quante volte si sono osservate le combinazioni possibili per le due variabili. Prendendo per esempio l’evento di mangiare una pizza in combinazione con la trasformazione in scoiattolo, si potrebbe avere quanto segue:

$$\varphi = \frac{n_{11}n_{00} - n_{10}n_{01}}{\sqrt{n_1 \cdot n_0 \cdot n_{\cdot 1} \cdot n_{\cdot 0}}} \quad (4.1)$$

Per calcolare  $\varphi$  si può usare la formula seguente, dove n fa riferimento alla tabella:

La notazione  $n_{01}$  indica quante volte la prima variabile (pizza) è `false` (0) e la seconda variabile (trasformazione in scoiattolo) è `true` (1). In questo esempio,  $n_{01}$  è 4.

No pizza, no scoiattolo	76
	Pizza, no scoiattolo
No pizza, scoiattolo	4
	Pizza, scoiattolo
	1

Il valore  $n_{1\cdot}$  indica quante volte la prima variabile è `true`, che nella tabella di esempio è 10. Analogamente,  $n_{\cdot 0}$  indica quante volte la seconda variabile è `false`.

Nella tabella per la pizza, dunque, il dividendo (la parte sopra la riga di divisione) è  $1 \times 76 - 9 \times 4 = 40$ , mentre il divisore (la parte sotto la riga) è la radice quadrata di  $10 \times 80 \times 5 \times 85$ , o  $\sqrt{340000}$ . Ciò corrisponde a  $\varphi \approx 0.069$ , il che indica una correlazione molto, molto piccola. In altre parole, mangiare la pizza non sembra influenzare la trasformazione in scoiattolo.

## Calcolo delle correlazioni

Possiamo rappresentare una tabella di due righe per due colonne in JavaScript con un array a quattro elementi ([76, 9, 4, 1]). Possiamo tuttavia utilizzare anche altre notazioni: per esempio, un array contenente due array a due elementi ([[76, 9], [4, 1]]) o un oggetto con nomi di proprietà "11" e "01", ma l'array piatto è il più semplice e semplifica grandemente le espressioni per accedere alla tabella. Gli indici dell'array sono rappresentati da un numero binario a due bit, dove la cifra più a sinistra, e la più significativa, fa riferimento alla variabile squirrel, mentre quella di destra (meno significativa) alla variabile event. Per esempio, il numero binario 10 indica il caso in cui Jacques si è trasformato in scoiattolo pur in assenza dell'evento (in questo caso, mangiare la pizza). Ciò si è verificato quattro volte. Poiché il numero binario 10 si scrive 2 nella notazione decimale, questo numero viene memorizzato nella posizione 2 dell'array.

Ecco la funzione che calcola il coefficiente  $\varphi$  da quell'array:

---

```
function phi(table) {
    return (table[3] * table[0] - table[2] * table[1]) /
        Math.sqrt((table[2] + table[3]) *
            (table[0] + table[1]) *
            (table[1] + table[3]) *
            (table[0] + table[2]));
}
console.log(phi([76, 9, 4, 1]));
// → 0.068599434
```

---

Questa è semplicemente la traduzione diretta in JavaScript della formula per  $\varphi$ . La funzione `Math.sqrt` che calcola la radice quadrata offerta come oggetto `Math` negli ambienti JavaScript normali. Dobbiamo inoltre sommare due campi della tabella per ottenere campi come  $n_{1\bullet}$ , in quanto la struttura dati non memorizza direttamente le somme delle righe e delle colonne.

Jacques ha tenuto il diario per tre mesi. I dati relativi sono disponibili sul sito Web, dove si trovano i listati, <http://eloquentjavascript.net/code/>, scegliendo 4. Data Structures: Objects and Arrays dal menu a discesa. I dati sono memorizzati nella variabile `JOURNAL` e in un file scaricabile.

Per estrarre una tabella di due righe per due colonne per un evento specifico del diario, dobbiamo far passare in ciclo tutti i dati e sommare quante volte si sono verificati gli eventi in corrispondenza delle trasformazioni in scoiattolo.

---

```
function hasEvent(event, entry) {
    return entry.events.indexOf(event) != -1;
}
function tableFor(event, journal) {
```

```

var table = [0, 0, 0, 0];
for (var i = 0; i < journal.length; i++) {
  var entry = journal[i], index = 0;
  if (hasEvent(event, entry)) index += 1;
  if (entry.squirrel) index += 2;
  table[index] += 1;
}
return table;
}
console.log(tableFor("pizza", JOURNAL));
// → [76, 9, 4, 1]

```

---

La funzione `hasEvent` verifica se le registrazioni del diario contengono l'evento cercato. Tutti gli array hanno un metodo `indexOf`, che va a cercare un valore dato (in questo caso, il nome dell'evento) e restituisce la posizione, dove si trova l'evento, oppure -1 se l'evento non viene trovato. Pertanto, se la chiamata a `indexOf` non restituisce -1, sappiamo che l'evento si trova nella registrazione corrente.

Il corpo del ciclo in `tableFor` stabilisce in quale delle celle della tabella rientra ciascuna registrazione del diario: per questo, verifica se la registrazione contiene l'evento cercato e se l'evento si è verificato in corrispondenza di una trasformazione in scoiattolo. Il ciclo aggiunge quindi un'unità al numero dell'array che corrisponde alla cella.

Ora abbiamo gli strumenti che servono per calcolare le singole correlazioni. L'unica questione che rimane da risolvere è trovare la correlazione per ciascun tipo di evento registrato e stabilire se ne escono dei risultati significativi. Nel prossimo paragrafo vedremo come calcolare e memorizzare le correlazioni.

## Gli oggetti come mappe

Una delle soluzioni possibili consiste nel memorizzare le correlazioni in un array, utilizzando oggetti con proprietà `name` (nome) e `value` (valore). Così facendo però si complica l'operazione di andare a trovare la correlazione per i singoli eventi: per trovare l'oggetto col `name` giusto, dovete, infatti, far passare in un ciclo tutto l'array. Potremmo incapsulare in una funzione questo procedimento di ricerca; dovremmo, tuttavia, scrivere altro codice e il computer farebbe più lavoro del necessario.

Meglio pertanto utilizzare delle proprietà degli oggetti, con nomi che indichino il tipo di evento. Possiamo utilizzare la notazione di accesso con le parentesi quadre per creare e leggere le proprietà, insieme all'operatore `in` per verificare se una data proprietà esiste.

```

var map = {};
function storePhi(event, phi) {

```

```
map[event] = phi;  
}  
storePhi("pizza", 0.069);  
storePhi("touched tree", -0.081);  
console.log("pizza" in map);  
// → true  
console.log(map["touched tree"]);  
// → -0.081
```

---

Tracciare *mappe* in questo modo consente di passare dai valori di un dominio (in questo caso, i nomi degli eventi) ai valori corrispondenti in un altro dominio (in questo caso, i coefficienti  $\varphi$ ).

Questo modo di utilizzare gli oggetti può dare qualche problema e ne parleremo nel [Capitolo 6](#); ma per ora possiamo continuare senza preoccuparcene troppo.

Che cosa accade se vogliamo trovare tutti gli eventi per i quali abbiamo memorizzato un coefficiente? Poiché le proprietà non formano serie prevedibili, come si possono avere negli array, non possiamo ricorrere a un normale ciclo `for`. JavaScript offre un costrutto di ciclo specifico per analizzare le proprietà di un oggetto: simile ai cicli `for`, si distingue per l'uso della parola `in`.

---

```
for (var event in map)  
  console.log("La correlazione per '" + event +  
             "' is " + map[event]);  
// → La correlazione per 'pizza' is 0.069  
// → La correlazione per 'touched tree' is -0.081
```

---

## Analisi finale

Per trovare tutti i tipi di eventi che sono presenti nell'insieme di dati, dobbiamo pertanto esaminare ogni registrazione e far passare un ciclo sugli eventi in essa contenuti. L'oggetto `phis` mantiene i coefficienti di correlazione per tutti i tipi di eventi esaminati finora. Ogni volta che troviamo un tipo di evento non ancora memorizzato in `phis`, ne calcoliamo il coefficiente di correlazione e lo aggiungiamo all'oggetto.

```
function gatherCorrelations(journal) {  
  var phis = {};  
  for (var entry = 0; entry < journal.length; entry++) {  
    var events = journal[entry].events;  
    for (var i = 0; i < events.length; i++) {
```

```
var event = events[i];
if (!(event in phis))
    phis[event] = phi(tableFor(event, journal));
}
}
return phis;
}

var correlations = gatherCorrelations(JOURNAL);
console.log(correlations.pizza);
// → 0.068599434
```

---

Ecco i risultati:

---

```
for (var event in correlations)
    console.log(event + ": " + correlations[event]);
// → carrot: 0.0140970969
// → exercise: 0.0685994341
// → weekend: 0.1371988681
// → bread: -0.0757554019
// → pudding: -0.0648203724
// e così' via...
```

---

Quasi tutte le correlazioni hanno coefficiente vicino allo zero. Mangiare carote (carrots), pane (bread) o budino (pudding) non sembra far scattare la trasformazione in scoiattolo. *Sembra* invece che ciò accada più spesso nei fine settimana: proviamo ora a filtrare i risultati per evidenziare solo quelli maggiori di 0,1 o minori di -0,1.

---

```
for (var event in correlations) {
    var correlation = correlations[event];
    if (correlation > 0.1 || correlation < -0.1)
        console.log(event + ": " + correlation);
}
// → weekend:          0.1371988681
// → brushed teeth:   -0.3805211953
// → candy:            0.1296407447
// → work:             -0.1371988681
// → spaghetti:        0.2425356250
// → reading:          0.1106828054
// → peanuts:          0.5902679812
```

---

Da qui si rileva che ci sono due fattori con un coefficiente di correlazione decisamente più alto degli altri. Mangiare noccioline (peanuts) sembra aumentare le probabilità di trasformazione in scoiattolo, mentre lavarsi i denti (brushed teeth) ha un effetto decisamente negativo.

Interessante. Vediamo qualche altra prova:

---

```
for (var i = 0; i < JOURNAL.length; i++) {  
  var entry = JOURNAL[i];  
  if (hasEvent("peanuts", entry) &&  
    !hasEvent("brushed teeth", entry))  
    entry.events.push("peanut teeth");  
}  
console.log(phi(tableFor("peanut teeth", JOURNAL)));  
// → 1
```

---

Questo risultato è incontrovertibile: il fenomeno si verifica proprio quando Jacques mangia noccioline e non si lava i denti. In effetti, se fosse stato più attento all'igiene orale, non avrebbe mai scoperto il suo “piccolo” problema.

Alla luce dei risultati, basta che Jacques smetta di mangiare noccioline per porre fine alle sue trasformazioni.

## Ancora a proposito di array

Prima di concludere il capitolo, esamineremo alcuni altri concetti legati agli oggetti. Per prima cosa, vediamo alcuni metodi di array piuttosto utili.

In precedenza in questo capitolo, avevamo parlato di `push` e `pop`, che aggiungono ed eliminano elementi alla fine di un array. I metodi corrispondenti per effettuare quelle operazioni all'inizio dell'array sono `unshift` e `shift`.

---

```
var todoList = [];  
function rememberTo(task) {  
  todoList.push(task);  
}  
function whatIsNext() {  
  return todoList.shift();  
}  
function urgentlyRememberTo(task) {  
  todoList.unshift(task);  
}
```

---

Come si è visto, il programma che analizza le trasformazioni in scoiattolo mannaro ha il compito di gestire degli elenchi di attività. Per aggiungere un’attività alla fine di un elenco, richiamate `rememberTo("eat")` e, quando siete pronti a svolgere l’attività in questione, richiamate `whatIsNext()` per recuperare (ed eliminare) il primo elemento dell’elenco. Anche la funzione `urgentlyRememberTo` immette un’attività, solo che la aggiunge all’inizio invece che alla fine dell’elenco.

Il metodo `lastIndexOf` è simile a `indexOf`, con la differenza che va a cercare l’elemento dato partendo dalla fine invece che dall’inizio dell’array.

---

```
console.log([1, 2, 3, 2, 1].indexOf(2));
// → 1
console.log([1, 2, 3, 2, 1].lastIndexOf(2));
// → 3
```

---

Sia `indexOf`, sia `lastIndexOf` accettano un secondo argomento facoltativo, che indica da che punto iniziare la ricerca.

Un altro metodo molto importante è `slice`, che accetta un indice di partenza e uno di arrivo per restituire un array che contiene solo gli elementi compresi tra quelle due posizioni. L’indice di partenza è inclusivo, mentre quello di arrivo è esclusivo.

---

```
console.log([0, 1, 2, 3, 4].slice(2, 4));
// → [2, 3]
console.log([0, 1, 2, 3, 4].slice(2));
// → [2, 3, 4]
```

---

Se non si specifica l’indice di arrivo, `slice` recupera tutti gli elementi da quello di partenza in poi. Anche le stringhe hanno un metodo `slice`, che ha effetti analoghi. Il metodo `concat` consente di incollare tra loro gli array, in modo simile a come l’operatore `+` fa per le stringhe. L’esempio che segue dimostra l’uso di `concat` e `slice`. La funzione accetta un array e un indice e restituisce un nuovo array, che è una copia di quello originario, ma senza l’elemento nella posizione indice specificata.

---

```
function remove(array, index) {
  return array.slice(0, index)
    .concat(array.slice(index + 1));
}
console.log(remove(["a", "b", "c", "d", "e"], 2));
// → ["a", "b", "d", "e"]
```

---

## Le stringhe e le loro proprietà

I valori stringa hanno proprietà come `length` e `toUpperCase`, che si possono leggere e recuperare. Se, però, tentate di aggiungere una nuova proprietà, l'operazione non ha successo.

---

```
var myString = "Fido";
myString.myProperty = "value";
console.log(myString.myProperty);
// → undefined
```

---

I valori di tipo stringa, numero e booleano, infatti, non sono oggetti e sebbene il linguaggio non dia errori, se si tenta di impostare nuove proprietà per quei valori, JavaScript non le memorizza. Quei valori sono immutabili e non possono essere modificati.

Questi tipi di dati, però, hanno alcune proprietà integrate. Tutti i valori stringa dispongono di un certo numero di metodi, i più utili dei quali sono probabilmente `slice` e `indexOf`, che ricordano i metodi degli array con lo stesso nome.

---

```
console.log("coconuts".slice(4, 7));
// → nut
console.log("coconut".indexOf("u"));
// → 5
```

---

Una differenza è che la proprietà `indexOf` di una stringa può accettare una stringa con più di un carattere, mentre il metodo corrispondente per gli array si limita a un solo elemento.

---

```
console.log("one two three".indexOf("ee"));
// → 11
```

---

Il metodo `trim` elimina lo spazio vuoto (spazi, a capo, tabulatori e simili) dall'inizio e dalla fine della stringa.

---

```
console.log(" okay \n ".trim());
// → okay
```

---

Della proprietà `length` del tipo stringa si è già parlato. Per accedere ai singoli caratteri di una stringa si può usare il metodo `charAt`; ma si possono semplicemente leggere le proprietà numeriche, come fate per gli array.

---

```
var string = "abc";
console.log(string.length);
// → 3
console.log(string.charAt(0));
```

```
// → a  
console.log(string[1]);  
// → b
```

---

## L'oggetto arguments

Ogni volta che si richiama una funzione, la variabile speciale `arguments` viene aggiunta all'ambiente, dove viene eseguito il corpo della funzione. Questa variabile fa riferimento a un oggetto che contiene tutti gli argomenti passati alla funzione. Ricordate che in JavaScript potete passare a una funzione più o meno argomenti del numero dei parametri dichiarati dalla funzione stessa.

---

```
function noArguments() {}  
noArguments(1, 2, 3); // questo va bene  
function threeArguments(a, b, c) {}  
threeArguments(); // e anche questo
```

---

L'oggetto `arguments` ha una proprietà `length` che indica il numero di argomenti effettivamente passati alla funzione. Dispone anche di una proprietà per ciascuno degli argomenti, chiamati `0`, `1`, `2` e così via.

Se vi ricorda molto gli array, in effetti è proprio così. Quest'oggetto, però, non ha purtroppo nessuno dei metodi degli array (come `slice` o `indexOf`) e risulta pertanto un po' più difficile da utilizzare di un vero array.

---

```
function argumentCounter() {  
  console.log("Mi hai dato", arguments.length, "argomenti.");  
}  
argumentCounter("Straw man", "Tautology", "Ad hominem");  
// → Mi hai dato 3 argomenti.
```

---

Alcune funzioni, come `console.log`, accettano qualunque numero di argomenti. Di solito, queste funzioni passano in ciclo i valori contenuti nel rispettivo oggetto `arguments` e possono essere utilizzate per creare delle interfacce interessanti. Per esempio, ecco come avevamo impostato il codice per le registrazioni nel diario di Jacques:

---

```
addEntry(["work", "touched tree", "pizza", "running",  
         "television"], false);
```

---

Visto che questa funzione sarà richiamata molte volte, possiamo impostarne una versione alternativa, più facile da richiamare:

---

```
function addEntry(squirrel) {
```

```
var entry = {events: [], squirrel: squirrel};
for (var i = 1; i < arguments.length;
    entry.events.push(arguments[i]);
journal.push(entry);
}
addEntry(true, "work", "touched tree", "pizza",
    "running", "television");
```

---

In questa versione, viene letto il primo argomento (`squirrel`) nel solito modo; la funzione passa poi in ciclo gli altri argomenti, partendo dalla posizione di indice 1 (e saltando il primo argomento), per raccoglierli in un array.

## L'oggetto Math

Come abbiamo visto, `Math` offre tutta una serie di funzioni per lavorare con i numeri, come `Math.max` (trova il massimo), `Math.min` (trova il minimo) e `Math.sqrt` (radice quadrata).

L'oggetto `Math` è sostanzialmente un contenitore che raggruppa funzionalità simili. Esiste un solo oggetto `Math`, che non ha alcuna utilità come valore: offre invece uno *spazio dei nomi* che consente di utilizzare funzioni e valori senza doverli dichiarare come variabili globali.

Questo perché avere troppe variabili globali “inquinava” lo spazio dei nomi. Più sono i nomi utilizzati, maggiori sono le probabilità che sovrascriviate accidentalmente il valore di qualche variabile. Per esempio, siete liberi di usare il nome `"max"` per qualche componente del programma senza paura che sia sovrascritto, in quanto la funzione `max` in JavaScript è “protetta” dall’oggetto `Math`.

In altri linguaggi, sareste interrotti o avvisati quando definite una variabile con un nome già utilizzato in precedenza. JavaScript non lo fa e pertanto dovete prestare particolare attenzione ai nomi.

Tornando all’oggetto `Math`, esso offre anche funzioni trigonometriche: `cos` (coseno), `sin` (seno) e `tan` (tangente) e le rispettive funzioni inverse, `acos`, `asin` e `atan`. L’oggetto `Math.PI` offre il numero  $\pi$  (pi greco), o la migliore approssimazione di pi greco che possa stare in un numero di JavaScript. Notate nel nome di questa funzione la vecchia tradizione di programmazione, che vuole che i nomi delle costanti siano scritti in maiuscole.

---

```
function randomPointOnCircle(radius) {
    var angle = Math.random() * 2 * Math.PI;
    return {x: radius * Math.cos(angle),
            y: radius * Math.sin(angle)};
}
console.log(randomPointOnCircle(2));
```

```
// → {x: 0.3667, y: 1.966}
```

---

Non preoccupatevi se non siete abituati a lavorare con seni e coseni. Questi concetti verranno spiegati al momento opportuno, nel [Capitolo 13](#).

Nell'esempio precedente si utilizza `Math.random`, una funzione che restituisce un diverso numero pseudo-casuale compreso tra zero (incluso) e uno (escluso) ogni volta che la richiamate.

---

```
console.log(Math.random());  
// → 0.36993729369714856  
console.log(Math.random());  
// → 0.727367032552138  
console.log(Math.random());  
// → 0.40180766698904335
```

---

Sebbene i computer siano macchine deterministiche, in quanto reagiscono sempre nello stesso modo alle stesse stimolazioni, è possibile indurli a produrre numeri che sembrino casuali. Per far ciò, il computer mantiene nel suo stato interno un numero (o molti numeri); ogni volta che serve un numero casuale, svolge dei calcoli deterministici molto complicati sul suo stato interno e restituisce parte del risultato. Il computer utilizza il risultato per modificare il suo stato interno, facendo così in modo che il successivo numero “casuale” sia diverso da quello appena calcolato.

Se invece di un decimale volessimo un numero intero, potremmo utilizzare `Math.floor`, che arrotonda per difetto il risultato di `Math.random`.

---

```
console.log(Math.floor(Math.random() * 10));  
// → 2
```

---

Moltiplicando per 10 il numero casuale, otteniamo un numero maggiore o uguale a zero e minore di 10. Poiché `Math.floor` arrotonda per difetto, questa espressione può produrre indifferentemente uno qualunque dei numeri da 0 a 9.

L'oggetto `Math` offre anche le funzioni `Math.ceil` (per “ceiling”, che arrotonda per eccesso) e `Math.round` (che arrotonda al numero intero più vicino).

## L'oggetto global

In JavaScript si può accedere anche all'ambito di visibilità globale, ossia lo spazio dove vivono le variabili globali, attraverso un oggetto. Tutte le variabili globali sono presenti come proprietà di quest'oggetto. Nei browser, l'ambito globale è memorizzato nella variabile `window`.

---

```
var myVar = 10;
```

```
console.log("myVar" in window);
// → true
console.log(window.myVar);
// → 10
```

---

## Riepilogo

Oggetti e array (che sono un tipo particolare di oggetto) consentono di raggruppare diversi valori in uno solo. Concettualmente, ciò ci consente di raccogliere in un unico contenitore più elementi, senza bisogno di doverli rincorrere e “catturare” uno per uno.

In JavaScript, tutti i valori hanno proprietà tranne `null` e `undefined`. Alle proprietà si accede con la notazione `valore.nomeProp` o `valore["nomeProp"]`. Gli oggetti utilizzano nomi per le proprietà, e dispongono di solito di un numero fisso di proprietà. Gli array, invece, contengono quantità variabili di valori concettualmente identici e utilizzano numeri (a partire da 0) come nomi delle proprietà.

Negli array ci sono tuttavia alcune proprietà con nome, come `length` e un certo numero di metodi. I metodi sono funzioni che vivono all’interno delle proprietà e (di solito) intervengono sul valore della proprietà che li contiene.

Gli oggetti possono essere utilizzati anche come mappe per associare dei valori a dei nomi. L’operatore `in` consente di stabilire se un determinato oggetto contiene una proprietà col nome dato. La stessa parola chiave può essere utilizzata anche in cicli `for` (`for (nome variabile in oggetto)`) per far passare in ciclo tutte le proprietà di un oggetto.

## Esercizi

### **Somma di un intervallo**

Nell’introduzione, si era dato il seguente esempio come soluzione elegante per calcolare la somma di un intervallo di numeri:

---

```
console.log(sum(range(1, 10)));
```

---

Impostate una funzione `range` che accetti due argomenti, `start` ed `end`, e restituisca un array che contenga tutti i numeri compresi tra `start` ed `end` (compreso).

Poi, impostate una funzione `sum` che accetti un array di numeri e ne restituisca la somma. Eseguite il programma precedente e verificate che il risultato sia 55.

Infine, modificate la funzione `range` in modo che accetti un terzo argomento che indichi il valore di incremento (`step`) utilizzato per realizzare l’array. In assenza di valori per `step`, gli elementi dell’array aumenteranno di uno, come nell’esempio originario. La

chiamata alla funzione `range(1, 10, 2)` deve restituire `[1, 3, 5, 7, 9]`. Assicuratevi che il codice funzioni anche con valori di incremento negativi e che pertanto `range(5, 2, -1)` dia `[5, 4, 3, 2]`.

## Invertire l'ordine di un array

Gli array offrono un metodo `reverse`, che li modifica invertendo l'ordine degli elementi. Per questo esercizio, impostate due funzioni, `reverseArray` e `reverseArrayInPlace`. La prima, `reverseArray`, accetta come argomento un *array* e produce un nuovo array che contiene gli stessi elementi in ordine inverso. La seconda, `reverseArrayInPlace`, si comporta come il metodo `reverse`: modifica l'array passato come argomento per invertirne gli elementi. Nessuna delle due funzioni deve far uso del metodo standard `reverse`.

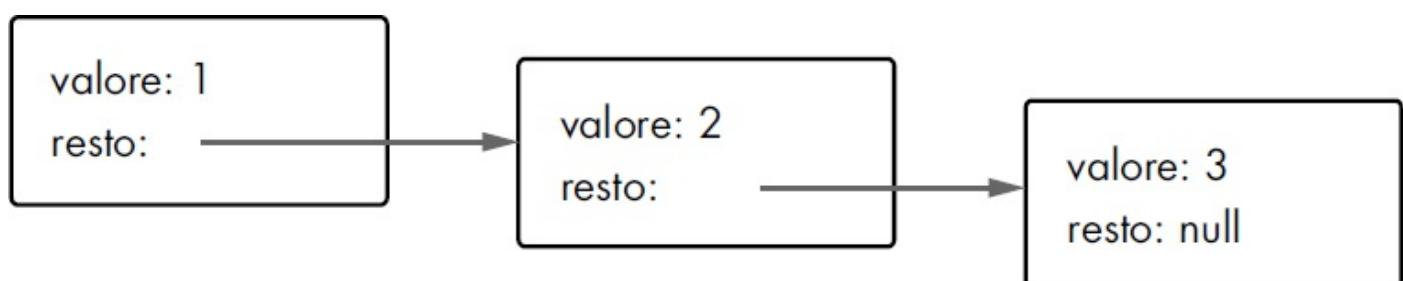
Ripensando a quel che si era detto nel capitolo precedente a proposito di funzioni pure ed effetti collaterali, quale forma sarà più utile in più situazioni? Quale sarà la più efficiente?

## Una lista

Gli oggetti, in quanto insiemi generici di valori, possono essere utilizzati per costruire qualunque tipo di struttura dati. Un tipo molto comune di struttura dati sono le liste (da non confondere con gli array). Una lista è un insieme nidificato di oggetti, dove il primo oggetto contiene un riferimento al secondo, il secondo al terzo e così via.

```
var list = {  
    value: 1,  
    rest: {  
        value: 2,  
        rest: {  
            value: 3,  
            rest: null  
        }  
    }  
};
```

Gli oggetti risultanti formano una catena come la seguente:



Un vantaggio delle liste è che possono condividere alcune parti della loro struttura. Per

esempio, se creo due nuovi valori `{value: 0, rest: list}` e `{value: -1, rest: list}` (dove `list` si riferisce alla variabile definita in precedenza), si avranno due liste indipendenti, che tuttavia condividono la struttura che costituisce gli ultimi tre elementi di ciascuna. Inoltre, anche la lista originaria resta valida come lista a tre elementi.

Scrivete una funzione `arrayToList` che imposti una struttura di dati come la precedente, dati `[1, 2, 3]` come argomento, e scrivete una funzione `listToArray` che produca un array partendo da una lista. Inoltre, impostate le funzioni ausiliarie `prepend`, che accettino un elemento e una lista per creare una nuova lista dove l'elemento è stato aggiunto all'inizio della lista di input, e `nth`, che accetti una lista e un numero per restituire l'elemento che si trova nella posizione data oppure `undefined` se l'elemento manca.

Se non lo avete già fatto, scrivete una versione ricorsiva di `nth`.

## **Confronto profondo**

L'operatore `==` consente di verificare se due oggetti sono identici, ma in alcuni casi può essere più utile confrontare i valori delle loro proprietà effettive.

Impostate una funzione, `deepEqual`, che accetti due valori e restituisca `true` solo se i due valori sono identici o se sono oggetti con le stesse proprietà e che risultano uguali anche quando confrontati con una chiamata ricorsiva a `deepEqual`.

Per stabilire se sia meglio verificare l'identità di due elementi con l'operatore `==`, oppure esaminandone le proprietà, potete utilizzare l'operatore `typeof`. Se produce "object" per entrambi i valori, dovete effettuare un confronto profondo. Esiste però una piccola eccezione che dovete tener presente: per un caso, anche `typeof null` dà come risultato "object".

# 5

## FUNZIONI DI ORDINE SUPERIORE

*Ci sono due modi per progettare del software: uno è di crearlo tanto semplice da non lasciar spazio ai difetti, l'altro di crearlo tanto complicato da non riuscire a trovarci difetti.*

Tony Hoare, Premio Turing 1980

Un programma grosso è un programma costoso e non soltanto per il tempo necessario per realizzarlo. Grandi dimensioni quasi sempre richiedono complessità e la complessità confonde i programmatore. I programmatore confusi, a loro volta, tendono a introdurre errori (*bachi*) nei programmi. E più grossi sono i programmi, più grande è lo spazio dove i bachi si possono nascondere, il che li rende più difficili da trovare.

Torniamo brevemente agli ultimi due programmi di esempio dell'Introduzione. Il primo è fine a se stesso e lungo sei righe:

---

```
var total = 0, count = 1;
while (count <= 10) {
    total += count;
    count += 1;
}
console.log(total);
```

---

Il secondo fa uso di due funzioni esterne ed è di una sola riga:

---

```
console.log(sum(range(1, 10)));
```

---

Quale dei due ha più probabilità di nascondere un baco?

Tenendo conto del codice per definire `sum` e `range`, anche il secondo programma è relativamente grande (in effetti è più grande del primo). Ciononostante, il secondo programma è probabilmente il più corretto.

Questo perché la soluzione è espressa in un vocabolario che corrisponde al problema in questione. Sommare dei numeri non richiede cicli e contatori: la questione si risolve in termini di intervalli e somme.

Le definizioni di questo vocabolario (le funzioni `sum` e `range`) possono contenere cicli,

contatori e altri particolari incidentali. Poiché, però, esprimono concetti più semplici del programma nel suo insieme, sono più facili da utilizzare correttamente.

## Astrazione

Nel contesto della programmazione, questi tipi di vocabolari si chiamano *astrazioni*. Le astrazioni nascondono i dettagli dell'esecuzione e offrono la possibilità di parlare dei problemi a un livello superiore (o più astratto).

Per fare un esempio, prendete le due ricette seguenti per una zuppa di piselli. Ecco la prima:

*Mettere una tazza di piselli per persona in una ciotola. Aggiungere acqua fino a coprire completamente i piselli. Lasciare nell'acqua per almeno 12 ore. Togliere i piselli dall'acqua e metterli in una pentola. Aggiungere 4 tazze d'acqua per persona. Coprire la pentola e far cuocere a fuoco basso per due ore. Prendere mezza cipolla per persona. Tagliarla a pezzi con un coltello. Aggiungerla ai piselli. Prendere un gambo di sedano per persona. Tagliarlo a pezzi con un coltello. Aggiungerlo ai piselli. Prendere una carota per persona. Tagliarla a pezzi. Con un coltello! Aggiungerla ai piselli. Lasciar cuocere per 10 minuti.*

E la seconda:

*Per persona: 1 tazza di piselli secchi spezzati, mezza cipolla tritata, un gambo di sedano e una carota.*

*Ammollare i piselli per 12 ore. Sobbollire per due ore in 4 tazze d'acqua (per persona). Aggiungere le verdure tritate. Far cuocere per altri 10 minuti.*

La seconda ricetta è più breve e più facile da interpretare. Bisogna, però, conoscere il significato di alcuni termini specifici: *ammollare*, *sobbollire*, *tritare* e forse anche *verdure*.

Quando si programma, non potete usare tutte le parole che avete a disposizione nel dizionario. Pertanto, rischiate di scegliere la strada della prima ricetta: quella di definire esattamente tutti i passi che il computer deve compiere, uno per uno, ignorando completamente se e quali possano essere espressi da concetti di livello superiore.

Come programmatore, dovete pertanto prendere l'abitudine di riconoscere quando un certo concetto va traslato in astrazione e definito come nuova "parola".

## Astrazioni per passare al setaccio gli array

Le funzioni più semplici, come quelle viste finora, sono un buon modo per costruire delle astrazioni. Non sempre, però, soddisfano tutte le aspettative.

Nel capitolo precedente, avete incontrato più volte il seguente ciclo for :

---

```
var array = [1, 2, 3];
for (var i = 0; i < array.length; i++) {
    var current = array[i];
    console.log(current);
}
```

---

Il concetto da esprimere è “Registra nella console ogni elemento che il ciclo trova nell’array”. L’espressione, piuttosto tortuosa, tira in ballo una variabile contatore `i`, una verifica della lunghezza dell’array e un’ulteriore dichiarazione di variabile per individuare l’elemento corrente. Al di là della sua scarsa eleganza, questa soluzione lascia spazio a parecchi errori potenziali: per esempio, potreste riutilizzare il nome della variabile `i`, scrivere per errore `lenght` invece di `length`, scambiare le variabili `i` e `current`, e così via.

Meglio pertanto provare a impostare una funzione come astrazione.

Intanto, non è difficile scrivere una funzione che esamini un array e richiami `console.log` per ogni suo elemento.

---

```
function logEach(array) {
    for (var i = 0; i < array.length; i++)
        console.log(array[i]);
}
```

---

Ammettiamo, però, di voler fare qualcosa di più che registrare gli elementi: poiché “fare qualcosa” può essere rappresentato come funzione e dato che le funzioni non sono altro che valori, allora potete passare l’azione voluta come valore di funzione.

---

```
function forEach(array, action) {
    for (var i = 0; i < array.length; i++)
        action(array[i]);
}
forEach(["Wampeter", "Foma", "Granfalloon"], console.log);
// → Wampeter
// → Foma
// → Granfalloon
```

---

Spesso, invece di passare al ciclo `forEach` una funzione predefinita, potete creare un valore di funzione al volo.

---

```
var numbers = [1, 2, 3, 4, 5], sum = 0;
forEach(numbers, function(number) {
    sum += number;
```

```
});  
console.log(sum);  
// → 15
```

---

Questo codice assomiglia molto ai cicli `for` classici, col corpo scritto come blocco sotto la riga con `forEach`. Tuttavia, il corpo si trova ora all'interno del valore di funzione, oltre che inserito tra le parentesi della chiamata a `forEach`. Ecco perché il codice va chiuso con una parentesi graffa e una tonda.

In questo modo, potete specificare un nome di variabile per l'elemento corrente (`number`) invece di doverlo recuperare manualmente dall'array.

È anche vero che non bisogna scrivere `forEach`, in quanto è disponibile come metodo standard per gli array. Siccome l'array è il “qualsiasi” su cui agisce il metodo, `forEach` richiede un solo argomento obbligatorio: la funzione da eseguire per ciascun elemento.

Per illustrare l'utilità di questa soluzione, riprendiamo una delle funzioni del capitolo precedente, che contiene due cicli che attraversano array:

---

```
function gatherCorrelations(journal) {  
  var phis = {};  
  for (var entry = 0; entry < journal.length; entry++) {  
    var events = journal[entry].events;  
    for (var i = 0; i < events.length; i++) {  
      var event = events[i];  
      if (!(event in phis))  
        phis[event] = phi(tableFor(event, journal));  
    }  
  }  
  return phis;  
}
```

---

Applicare `forEach` a questa funzione la abbrevia e la rende più chiara e meglio definita:

---

```
function gatherCorrelations(journal) {  
  var phis = {};  
  journal.forEach(function(entry) {  
    entry.events.forEach(function(event) {  
      if (!(event in phis))  
        phis[event] = phi(tableFor(event, journal));  
    });  
  });
```

```
    return phis;  
}
```

---

## Funzioni di ordine superiore

Le funzioni che operano su altre funzioni, accettandole come argomenti o restituendole come risultati, si chiamano *funzioni di ordine superiore*. Visto che le funzioni sono valori normali, l'esistenza di questo tipo di funzioni non dovrebbe stupire. Il termine viene dalla matematica, dove la distinzione tra funzioni e altri valori viene presa più seriamente.

Le funzioni di ordine superiore consentono l'astrazione di *azioni* e non solo di valori. Ne esistono di diversi tipi. Per esempio, si possono avere funzioni che creano nuove funzioni.

---

```
function greaterThan(n) {  
  return function(m) { return m > n; };  
}  
  
var greaterThan10 = greaterThan(10);  
console.log(greaterThan10(11));  
// → true
```

---

E si possono avere funzioni che ne modificano altre:

```
function noisy(f) {  
  return function(arg) {  
    console.log("sto chiamando con", arg);  
    var val = f(arg);  
    console.log("chiamato con", arg, "- ottenuto", val);  
    return val;  
  };  
}  
  
noisy(Boolean)(0);  
// → sto chiamando con 0  
// → chiamato con 0 - ottenuto false
```

---

È anche possibile scrivere funzioni che offrono nuovi tipi di flussi di controllo:

```
function unless(test, then) {  
  if (!test) then();  
}  
  
function repeat(times, body) {
```

```
for (var i = 0; i < times; i++) body(i);
}
repeat(3, function(n) {
  unless(n % 2, function() {
    console.log(n, "numero pari");
  });
});
// → 0 numero pari
// → 2 numero pari
```

---

Le regole lessicali sugli ambiti di visibilità discusse nel [Capitolo 3](#) risultano molto utili quando si usano le funzioni in questo modo. Nell'esempio precedente, la variabile `n` è un parametro della funzione `outer`. Poiché la funzione interna (`inner`) vive all'interno dell'ambiente di quella esterna (`outer`), si può usare `n`. I corpi delle funzioni interne hanno accesso alle variabili che stanno loro intorno e hanno un ruolo simile ai blocchi tra parentesi graffe utilizzati dai cicli regolari e dalle dichiarazioni condizionali. Un'importante differenza è che le variabili dichiarate all'interno di funzioni interne non hanno effetto nell'ambiente della funzione esterna, il che di solito è un vantaggio.

## Passare gli argomenti

La funzione `noisy` definita in precedenza, dove l'argomento è avvolto in un'altra funzione, presenta un grosso problema.

---

```
function noisy(f) {
  return function(arg) {
    console.log("calling with", arg);
    var val = f(arg);
    console.log("called with", arg, "- got", val);
    return val;
  };
}
```

---

Se `f` accettasse più di un parametro, la funzione troverebbe solo il primo. Si potrebbero aggiungere altri argomenti alla funzione interna (`arg1`, `arg2` e così via) e passarli tutti a `f`, ma non si può sapere a priori quanti ne serviranno. Questa soluzione negherebbe inoltre a `f` l'accesso alle informazioni contenute in `arguments.length`. Poiché verrebbe passato sempre lo stesso numero di argomenti, non possiamo sapere quanti argomenti avevamo passato inizialmente.

Per queste situazioni, le funzioni JavaScript offrono un metodo `apply`. Passando a questo metodo un array di argomenti (o un oggetto simile), il metodo richiamerà la

funzione con quegli argomenti.

---

```
function transparentWrapping(f) {  
    return function() {  
        return f.apply(null, arguments);  
    };  
}
```

---

Questa funzione non serve a molto, ma dimostra la struttura che ci interessa: la funzione restituita passa a `f` tutti gli argomenti dati e solo quelli. Lo fa passando il proprio oggetto `arguments` ad `apply`. Il primo argomento per `apply`, che nell'esempio è `null`, può simulare una chiamata a un metodo: ci torneremo sopra nel prossimo capitolo.

## JSON

Le funzioni di ordine superiore che, in un modo o nell'altro, applicano una funzione agli elementi di un array sono di uso comune in JavaScript. Il metodo `forEach` è la più primitiva di tali funzioni, ma ne esistono parecchie altre, disponibili come metodi per gli array. Facciamo qualche esperimento con un altro insieme di dati per prendere un po' di familiarità con queste funzioni.

Anni fa, qualcuno fece un sacco di ricerche in vari archivi e mise insieme un libro sulla storia del mio cognome (Haverbeke, che significa “Ruscello di avena”). Quando lo aprii, speravo di trovarci storie di cavalieri, pirati e alchimisti... mentre il libro è solo pieno di contadini delle Fiandre. Tanto per divertirmi un po', estrassi le informazioni sui miei antenati diretti e le convertii in un formato leggibile dal computer.

Ecco un esempio del risultato:

---

```
[  
  {"name": "Emma de Milliano", "sex": "f",  
   "born": 1876, "died": 1956,  
   "father": "Petrus de Milliano",  
   "mother": "Sophia van Damme"},  
  {"name": "Carolus Haverbeke", "sex": "m",  
   "born": 1832, "died": 1905,  
   "father": "Carel Haverbeke",  
   "mother": "Maria van Brussel"},  
  ... eccetera  
]
```

---

Questo formato si chiama JSON, che si pronuncia come il nome proprio Jason e sta per JavaScript Object Notation. Il formato JSON viene ampiamente utilizzato per la

raccolta e lo scambio di dati sul Web.

JSON è simile a JavaScript come stile di impostazione di array e oggetti, con alcune limitazioni. Tutti i nomi delle proprietà devono essere racchiusi tra virgolette doppie; sono permesse solo semplici espressioni di dati, ma nessuna chiamata di funzioni, né variabili, né altro che richieda delle computazioni. In JSON non sono ammessi neppure i commenti.

JavaScript offre due funzioni, `JSON.stringify` e `JSON.parse`, per convertire dati da e in questo formato. Il primo accetta un valore JavaScript e restituisce una stringa convertita in JSON. Il secondo prende stringhe di questo tipo e le converte nel valore corrispondente.

---

```
var string = JSON.stringify({name: "X", born: 1980});
console.log(string);
// → {"name": "X", "born": 1980}
console.log(JSON.parse(string).born);
// → 1980
```

---

La variabile `ANCESTRY_FILE`, che si trova sul sito <http://eloquentjavascript.net/code/> nell'area dedicata a questo capitolo e in un file scaricabile, contiene i dati del mio file JSON come stringa. Ecco come convertirla per sapere quante persone contiene:

---

```
var ancestry = JSON.parse(ANCESTRY_FILE);
console.log(ancestry.length);
// → 39
```

---

## Filtrare gli array

La funzione che segue consente di trovare, tra i dati raccolti sugli antenati, le persone che erano giovani nel 1924. Questa funzione esclude gli elementi di un array che non superano la prova:

---

```
function filter(array, test) {
  var passed = [];
  for (var i = 0; i < array.length; i++) {
    if (test(array[i]))
      passed.push(array[i]);
  }
  return passed;
}

console.log(filter(ancestry, function(person) {
  return person.born > 1900 && person.born < 1925;
```

```
});  
// → [{name: "Philibert Haverbeke", ...}, ...]
```

---

Per riempire un “buco” nella computazione, si usa l’argomento `test`, che è un valore di funzione. La funzione di prova (`test`) viene richiamata per ciascun elemento; il suo valore di restituzione determina se un certo elemento va inserito nell’array da restituire.

Tre persone erano vive e giovani nel 1924: mio nonno, mia nonna e la mia prozia.

Si noti che la funzione `filter` non elimina elementi dall’array, ma costruisce un nuovo array che contiene solo gli elementi che superano la prova. Questa è una funzione pura, in quanto non modifica l’array che le viene passato.

Come `forEach`, anche `filter` è un metodo standard per gli array. Nell’esempio, si è definita la funzione solo per dimostrarne il funzionamento interno. D’ora in poi, utilizzeremo il formato seguente:

```
console.log(ancestry.filter(function(person) {  
    return person.father == "Carel Haverbeke";  
}));  
// → [{name: "Carolus Haverbeke", ...}]
```

---

## Trasformazioni con `map`

Ammettiamo di avere un array di oggetti, che rappresentano persone, ottenuto filtrando in qualche modo l’array degli antenati, e di volere invece un array di nomi, più facili da leggere.

Il metodo `map` trasforma un array applicando a tutti i suoi elementi una funzione e costruendo un nuovo array dai valori restituiti. Il nuovo array avrà la stessa lunghezza di quello passato alla funzione, ma il suo contenuto sarà “mappato” a una nuova forma.

```
function map(array, transform) {  
    var mapped = [];  
    for (var i = 0; i < array.length;  
        mapped.push(transform(array[i]));  
    return mapped;  
}  
  
var overNinety = ancestry.filter(function(person) {  
    return person.died - person.born > 90;  
});  
console.log(map(overNinety, function(person) {  
    return person.name;
```

```
});  
// → ["Clara Aernoudts", "Emile Haverbeke",  
//      "Maria Haverbeke"]
```

---

Per combinazione, i componenti della famiglia che hanno vissuto almeno fino a 90 anni sono gli stessi tre che erano giovani negli anni Venti, ossia la generazione più recente nell'insieme di dati che avevo raccolto. Evidentemente la medicina ha fatto grandi progressi!

Come `forEach` e `filter`, anche `map` è un metodo standard degli array.

## Le somme con reduce

Un'altra situazione molto comune dove si devono svolgere calcoli su un array è quando si vuole ottenere un solo valore per l'array. L'esempio già presentato, dove si calcolava la somma di un gruppo di numeri, ben illustra questa situazione. Un altro esempio potrebbe essere quello di trovare la persona nata nell'anno meno recente tra quelli inseriti nella raccolta di dati.

L'operazione di ordine superiore che rappresenta queste situazioni si chiama *riduzione* (in inglese, `reduce` o `fold`). Potete immaginarla come un modo per “ripiegare” l'array un elemento alla volta: quando si sommano dei numeri, si parte dal valore in posizione zero e si aggiunge il valore di ciascun elemento alla somma degli elementi precedenti, ossia al totale corrente.

I parametri della funzione `reduce` sono l'array, una funzione di somma e un valore iniziale. Poiché è meno intuitiva delle funzioni `filter` e `map`, dovete esaminarla con attenzione:

```
function reduce(array, combine, start) {  
  var current = start;  
  for (var i = 0; i < array.length; i++)  
    current = combine(current, array[i]);  
  return current;  
}  
  
console.log(reduce([1, 2, 3, 4], function(a, b) {  
  return a + b;  
}, 0));  
// → 10
```

---

Il metodo standard `reduce`, che corrisponde a questa funzione, offre un vantaggio aggiunto: se l'array contiene almeno un elemento, si può evitare di specificare l'argomento `start`. Il metodo prenderà il primo elemento dell'array come valore iniziale e calcolerà la somma a partire dal secondo elemento.

Per trovare l'antenato più vecchio, nei dati raccolti, reduce si può utilizzare come segue:

---

```
console.log(ancestry.reduce(function(min, cur) {
  if (cur.born < min.born) return cur;
  else return min;
}));
// → {name: "Pauwels van Haverbeke", born: 1535, ...}
```

---

## Componibilità

Proviamo a pensare a come avremmo impostato l'esempio precedente (trovare l'antenato con l'anno di nascita più lontano) senza funzioni di ordine superiore. Il codice non è poi tanto peggiore:

---

```
var min = ancestry[0];
for (var i = 1; i < ancestry.length; {
  var cur = ancestry[i];
  if (cur.born < min.born)
    min = cur;
}
console.log(min);
// → {name: "Pauwels van Haverbeke", born: 1535, ...}
```

---

C'è qualche variabile in più e il programma è due righe più lungo, ma pur sempre abbastanza facile da capire.

Le funzioni di ordine superiore cominciano, in effetti, a dimostrare tutta la loro efficacia quando bisogna *combinare* delle funzioni. Per esempio, proviamo a scrivere del codice che trovi l'età media per gli uomini e per le donne nell'insieme di dati:

---

```
function average(array) {
  function plus(a, b) { return a + b; }
  return array.reduce(plus) / array.length;
}

function age(p) { return p.died - p.born; }
function male(p) { return p.sex == "m"; }
function female(p) { return p.sex == "f"; }
console.log(average(ancestry.filter(male).map(age)));
// → 61.67
```

```
console.log(average(ancestry.filter(female).map(age)));
// → 54.56
```

---

Se sembra sciocco dover definire una funzione `plus`, va tenuto presente che in JavaScript gli operatori, diversamente dalle funzioni, non sono valori e pertanto non possono essere passati come argomenti.

Invece di invischiare la logica in un gran ciclo continuo, la componiamo nei singoli concetti che interessano: stabilire il sesso, calcolare l'età e fare la media dei numeri. Applicando queste funzioni una per una otterremo i risultati desiderati.

Questa soluzione è fantastica per scrivere del codice facile da leggere. Purtroppo, la chiarezza ha un costo.

## Il costo

Nel paese felice del codice elegante e degli arcobaleni, vive un mostro guastafeste che si chiama *inefficienza*.

Un programma che elabora un array si esprime, al massimo dell'eleganza, in una sequenza di passi distinti, ciascuno dei quali svolge un'azione sull'array e produce un nuovo array. Questo procedimento, però, è piuttosto costoso.

Analogamente, passare una funzione a `forEach` e far svolgere il lavoro di trasformare l'array da quel metodo è una soluzione (per noi) comoda e facile da leggere. Le chiamate di funzione, in JavaScript, però, costano più care di semplici corpi di ciclo.

Lo stesso discorso vale per molte delle tecniche che possono migliorare la chiarezza dei programmi. Le astrazioni aggiungono dei livelli tra le operazioni grezze del computer e i concetti di programmazione, e costringono la macchina a lavorare di più. Questa non è tuttavia una legge ferrea: alcuni linguaggi di programmazione offrono un miglior supporto per le astrazioni, senza aggiungere pesantezza. Persino in JavaScript i programmatore più esperti riescono a scrivere codice astratto che viene comunque eseguito velocemente. Il problema, in ogni caso, sorge più spesso di quel che vorremmo.

Per fortuna, i computer sono in genere molto, molto veloci. Quando si elaborano insiemi di dati relativamente piccoli o si risponde a eventi che avvengono a velocità umana (per esempio, quando l'utente preme un pulsante), *non ha importanza* se la soluzione elegante richiede mezzo millisecondo, quando l'alternativa super-ottimizzata ce ne mette solo un decimillesimo.

Non guasta tuttavia abituarsi a prender nota di quanto tempo richiederà grossomodo l'esecuzione di una certa parte del programma. Se si ha un ciclo all'interno di un ciclo (sia direttamente, sia attraverso la chiamata, da parte del ciclo esterno, a una funzione che attiva il ciclo interno), il codice del ciclo interno sarà eseguito  $N \times M$  volte, dove  $N$  è il numero di ripetizioni del ciclo esterno e  $M$  il numero di ripetizioni del ciclo interno per ciascuna delle iterazioni esterne. Se il ciclo interno, a sua volta, avvia un altro ciclo che richiede  $P$  iterazioni, il corpo del ciclo viene eseguito  $M \times N \times P$  volte e così via. Maggiore è

il risultato di questa operazione, più lenta sarà l'esecuzione del programma: in casi come questo, il problema sta tutto nella piccola parte di codice che sta nel ciclo interno.

## Bis-bis-bis-bis...

Mio nonno, Philibert Haverbeke, è riportato nel file con i dati. Partendo da lui, posso seguire l'albero dei miei antenati per scoprire se la persona più antica, Pauwels van Haverbeke, è un mio ascendente diretto. E se lo è, posso calcolare quanto del mio DNA condivido con lui (in teoria).

Per poter passare dal nome di un genitore all'oggetto che rappresenta quella persona, per prima cosa impostiamo un oggetto che associa i nomi alle persone:

---

```
var byName = {};
ancestry.forEach(function(person) {
  byName[person.name] = person;
});
console.log(byName["Philibert Haverbeke"]);
// → {name: "Philibert Haverbeke", ...}
```

---

Ora, la questione non è proprio semplice come seguire le proprietà del padre e contare quante ne servono per arrivare a Pauwels. In famiglia, ci sono diversi casi di matrimonio tra secondi cugini (in fondo si viveva in piccoli villaggi e non ci si spostava molto!). Per questo, i rami dell'albero genealogico si intrecciano in alcuni punti; il che significa che io condivido più di  $1/2^G$  di geni con questa persona, dove  $G$  è il numero di generazioni tra Pauwels e me. Questa formula esprime il concetto che, in ogni generazione, il pool genetico si divide a metà.

Un modo sensato per esprimere la questione è di immaginarla analoga alla funzione `reduce`, che condensa un array in un singolo valore combinando i valori progressivamente, da sinistra a destra. Anche qui si vuole condensare la struttura di dati in un valore singolo, ma in questo caso seguendo la discendenza familiare. La *forma* dei dati, quindi, è un albero genealogico, non una lista piatta.

Per ridurre questa forma, calcoliamo il valore per una data persona combinando i valori dei suoi antenati. Ciò si può fare ricorsivamente: per la persona A, calcoliamo i valori dei genitori di A, il che richiede di calcolare il valore dei nonni di A e così via. In teoria, dovremmo analizzare un numero infinito di persone, ma poiché l'insieme di dati in questione è finito, a un certo punto ci fermiamo. In pratica, assegniamo alla funzione di riduzione un valore predefinito, che esprima i valori di chi non fa parte dei dati. Nel nostro caso, quel valore sarà zero, in quanto si presume che le persone che non fanno parte della lista non condividano DNA con l'antenato in questione.

Dati una persona, una funzione per combinare i valori dei genitori della persona data e un valore predefinito, `reduceAncestors` condensa un valore a partire dall'albero

genealogico.

---

```
function reduceAncestors(person, f, defaultValue) {
    function valueFor(person) {
        if (person == null)
            return defaultValue;
        else
            return f(person, valueFor(byName[person.mother]),
                     valueFor(byName[person.father]));
    }
    return valueFor(person);
}
```

---

La funzione interna (`valueFor`) esamina una sola persona. Grazie al potere della ricorsività, può semplicemente richiamarsi più volte per esaminare padre e madre di quella persona. I risultati, insieme all'oggetto `person`, vengono passati a `f`, che restituisce il valore per la persona data.

Con questo risultato, si calcola la quantità di DNA che mio nonno condivideva con Pauwels van Haverbeke e si divide il risultato per quattro.

---

```
function sharedDNA(person, fromMother, fromFather) {
    if (person.name == "Pauwels van Haverbeke")
        return 1;
    else
        return (fromMother + fromFather) / 2;
}
var ph = byName["Philibert Haverbeke"];
console.log(reduceAncestors(ph, sharedDNA, 0) / 4);
// → 0.00049
```

---

La persona di nome Pauwels van Haverbeke, chiaramente, condivideva il 100% del DNA con Pauwels van Haverbeke, in quanto non ci sono casi di omonimia nell'insieme di dati. Pertanto, in questo caso la funzione restituisce 1. Tutte le altre persone condividono la media dei valori condivisi dai rispettivi genitori.

Statisticamente parlando, si può dire a questo punto che io condivido circa lo 0,05% del mio DNA con quel mio antenato del XVI secolo. Va detto che si tratta di un'approssimazione statistica e non di un valore effettivo. È un numero piuttosto piccolo, ma data la quantità di materiale genetico che ciascuno di noi si porta dietro (un po' più di 3 miliardi di paia di basi), si può presumere che alcuni aspetti della mia macchina biologica trovino origine in Pauwels.

Avremmo potuto calcolare questo numero anche senza ricorrere a `reduceAncestors`. Distinguendo il problema generale (condensare un albero genealogico) dal caso specifico (calcolare il DNA condiviso), però, migliora la chiarezza del codice e si può riutilizzare la parte astratta del programma per altre situazioni. Per esempio, il codice seguente consente di trovare la percentuale di antenati noti, per una certa persona, che hanno vissuto più di 70 anni:

---

```
function countAncestors(person, test) {
  function combine(person, fromMother, fromFather) {
    var thisOneCounts = test(person);
    return fromMother + fromFather + (thisOneCounts ? 1 : 0);
  }
  return reduceAncestors(person, combine, 0);
}

function longLivingPercentage(person) {
  var all = countAncestors(person, function(person) {
    return true;
  });
  var longLiving = countAncestors(person, function(person) {
    return (person.died - person.born) >= 70;
  });
  return longLiving / all;
}
console.log(longLivingPercentage(byName["Emile Haverbeke"]));
// → 0.145
```

---

Questi numeri non vanno tuttavia presi troppo sul serio, visto che l'insieme di dati di partenza contiene un gruppo di persone abbastanza arbitrario. Il codice, però, illustra come `reduceAncestors` può fornire un valido elemento di vocabolario per lavorare con una struttura dati di tipo albero genealogico.

## Il metodo bind

Tutte le funzioni hanno un metodo `bind`, che crea una nuova funzione che richiamerà la funzione originaria, ma con alcuni argomenti prestabiliti.

Il listato seguente mostra un esempio d'uso di `bind`. Qui, si definisce una funzione `isInSet` che indica se una certa persona si trova in una serie data di stringhe. Per richiamare `filter` e raccogliere gli oggetti persone i cui nomi fanno parte dell'insieme di dati, si può impostare un'espressione che chiama `isInSet` con l'insieme di dati come primo argomento oppure *applicare in parte* la funzione `isInSet`.

---

```

var theSet = ["Carel Haverbeke", "Maria van Brussel",
              "Donald Duck"];
function isInSet(set, person) {
    return set.indexOf(person.name) > -1;
}
console.log(ancestry.filter(function(person) {
    return isInSet(theSet, person);
}));
```

// → [{name: "Maria van Brussel", ...},  
// {name: "Carel Haverbeke", ...}]

```

console.log(ancestry.filter(isInSet.bind(null, theSet)));
// → ... stesso risultato
```

---

La chiamata a `bind` restituisce una funzione che richiama `isInSet` con `theSet` come primo argomento, seguito dagli altri argomenti eventualmente passati alla funzione collegata.

Il primo argomento, dove nell'esempio si passa `null`, viene utilizzato nelle chiamate al metodo, in modo simile al primo argomento di `apply`. Di questo parlerò più in dettaglio nel prossimo capitolo.

## Riepilogo

La possibilità di passare valori di funzione ad altre funzioni è un aspetto di JavaScript molto utile. Esso, infatti, consente di impostare dei calcoli con “buchi” come funzioni e fare in modo che il codice che richiama quelle funzioni riempia i buchi fornendo valori di funzione che descrivono i calcoli mancanti.

Gli array dispongono di un buon numero di metodi di ordine superiore: `forEach` per svolgere qualche operazione con ciascuno degli elementi, `filter` per costruire un nuovo array senza alcuni degli elementi, `map` per costruire un nuovo array dove su ogni elemento si è fatta passare una funzione e `reduce` per combinare tutti gli elementi dell'array in un solo valore.

Le funzioni hanno un metodo `apply`, che consente di richiamarle con un array che ne specifica gli argomenti. Offrono inoltre un metodo `bind`, che serve per creare versioni parziali della funzione di partenza.

## Esercizi

### *Appiattire degli array*

Applicate il metodo `reduce` insieme al metodo `concat` per “appiattire” un array di array e

ridurlo a un solo array che contenga tutti gli elementi degli array dati.

## Differenza di età tra madri e figli

Prendendo l'insieme di dati di esempio di questo capitolo, calcolate la differenza media di età tra madri e figli, prendendo come base l'età della madre al momento della nascita dei figli. Per questo, potete utilizzare la funzione `average` descritta in precedenza nel capitolo.

Noterete che non tutte le madri inserite nei dati sono presenti nell'array. In questi casi potrebbe essere utile l'oggetto `byName`, che aiuta a trovare l'oggetto di una persona partendo dal nome.

## Speranza di vita storica

Quando esaminammo le persone dell'insieme di dati che vissero più di 90 anni, il risultato riportava solo persone dell'ultima generazione. Esaminiamo più attentamente quel fenomeno.

Calcolate, per ciascun secolo, l'età media delle persone inserite nella raccolta dati genealogici. Per assegnare le persone ai secoli, prendete l'anno della morte, dividetelo per 100 e arrotondate lo per eccesso, come per esempio in `Math.ceil(person.died / 100)`.

Impostate poi una funzione `groupBy` che esprima l'astrazione dell'operazione di raggruppamento per secolo. La funzione dovrebbe accettare come argomenti un array e una funzione che calcola il gruppo per un elemento dell'array e restituisce un oggetto che mappa i nomi dei gruppi ad array dei membri di ciascun gruppo.

## Tutti e qualcuno

Gli array offrono i metodi standard `every` e `some`. Entrambi accettano una funzione predicativa che, se richiamata con un elemento di un array come argomento, restituisce `true` o `false`. Così come `&&` restituisce un valore `true` solo quando sono `true` tutte e due le espressioni, `every` restituisce `true` solo quando la funzione predicativa restituisce `true` per tutti gli elementi dell'array. In maniera analoga, `some` restituisce `true` quando la funzione predicativa restituisce `true` per uno qualunque degli elementi (il primo incontrato). Entrambi i metodi non elaborano più elementi dello stretto necessario: per esempio, se `some` trova che la funzione predicativa è valida per il primo elemento dell'array, non andrà a esaminare i valori che seguono.

Impostate due funzioni, `every` e `some`, che si comportano come i metodi di quel nome, salvo che, invece di essere metodi, accettano l'array come primo argomento.

## LA VITA SEGRETA DEGLI OGGETTI

*Il problema coi linguaggi orientati agli oggetti è che si portano dietro tutto quest'ambiente隐含的. Vuoi una banana e invece ti trovi con un gorilla che tiene in mano la banana e tutta la giungla.*

Joe Armstrong, intervistato su *Coders at Work*

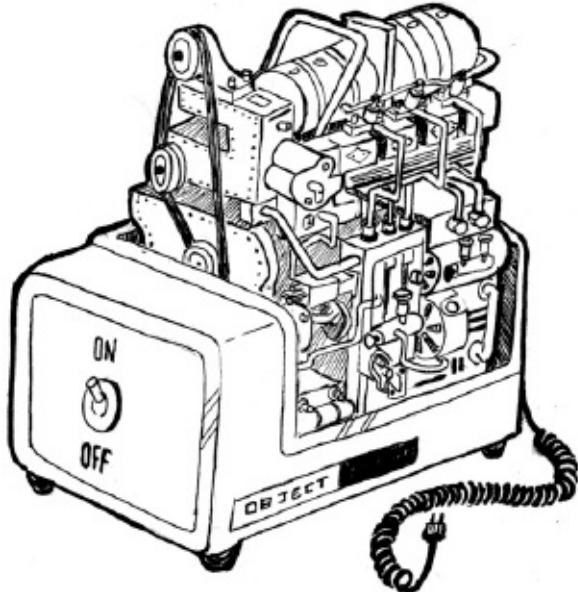
Nel linguaggio dei programmatore, “oggetto” è una parola carica di significati. Nella mia professione, gli oggetti sono uno stile di vita, il pomo della discordia per guerre sante e i testimoni di una moda un tempo molto amata, che continua ad avere un suo fascino.

Per chi non è dell’ambiente, questo può creare un po’ di confusione. Cominciamo con una breve storia degli oggetti come costrutto di programmazione.

### Storia

Come quasi tutte le storie sulla programmazione, quella degli oggetti nasce dal problema della complessità. Una scuola filosofica dice che, per gestirla meglio, basta dividerla in piccoli scompartimenti, isolati l’uno dall’altro. Questi scompartimenti han finito con l’essere chiamati *oggetti*.

Un oggetto è un guscio rigido, che nasconde al suo interno della complessità fluida e informe, offrendo al suo posto manopole e connettori (come i metodi) che presentano un’*interfaccia* attraverso la quale si può usare l’oggetto. L’idea è che l’interfaccia sia relativamente semplice e tutte le complicazioni che avvengono *all’interno* dell’oggetto possano essere ignorate quando si interviene su di esso.



Per esempio, pensate a un oggetto che offre un’interfaccia a una parte dello schermo. Consente di disegnare o di scrivere in una certa zona dello schermo, ma nasconde i particolari tecnici di come forme e simboli vengano convertiti in pixel. Le sole cose che dovete sapere per far uso di quell’oggetto sono i metodi disponibili; per esempio, `drawCircle` per tracciare un cerchio.

I concetti di fondo vennero sviluppati inizialmente negli anni Settanta e Ottanta. Negli anni Novanta, il progresso di quei concetti fu accompagnato da una colossale campagna promozionale: la rivoluzione della programmazione orientata agli oggetti. Improvvisamente, eserciti di programmatori dichiararono che gli oggetti erano l’unico modo per programmare e tutto quel che non aveva a che fare con gli oggetti erano assurdità fuori moda.

Peraltro, quel tipo di fanatismo produce sempre grandi quantità di schiocchezze senza senso; rispetto a quei tempi, si è, in effetti, verificata una specie di contro-rivoluzione. Al giorno d’oggi, in alcuni ambiti gli oggetti godono persino di cattiva reputazione.

Io preferisco guardare alla questione da un punto di vista pratico piuttosto che ideologico. Parecchi sono i concetti utili, venuti alla ribalta grazie alla cultura orientata agli oggetti: in particolare, quello dell’*incapsulazione*, ossia della distinzione tra complessità interna e interfaccia esterna. E questi aspetti vanno sicuramente studiati.

Questo capitolo descrive l’interpretazione, abbastanza eccentrica, che JavaScript dà degli oggetti e il modo in cui si pone rispetto alle tecniche orientate agli oggetti più classiche.

## Metodi

I metodi non sono altro che proprietà che mantengono dei valori di funzione. Ecco un metodo semplice:

---

```
var rabbit = {};
```

```
rabbit.speak = function(line) {
  console.log("The rabbit says '" + line + "'");
};

rabbit.speak("I'm alive.");
// → The rabbit says 'I'm alive.' [Il coniglio dice 'Sono vivo']
```

---

Di solito, i metodi svolgono una qualche operazione con l'oggetto su cui sono chiamati a intervenire. Quando si chiama una funzione come metodo, ossia avendola chiamata subito dopo averla esaminata come proprietà, come in `object.method()`, la variabile speciale `this` nel suo corpo punta all'oggetto in questione.

---

```
function speak(line) {
  console.log("The " + this.type + " rabbit says '" +
    line + "'");
}

var whiteRabbit = {type: "white", speak: speak};
var fatRabbit = {type: "fat", speak: speak};
whiteRabbit.speak("Oh my ears and whiskers, " +
  "how late it's getting!");
// → The white rabbit says 'Oh my ears and whiskers, how
//     late it's getting!' [Il coniglio bianco dice 'Oh cielo,
//     com'e' tardi!']

fatRabbit.speak("I could sure use a carrot right now.");
// → The fat rabbit says 'I could sure use a carrot
//     right now.' [Il coniglio grasso dice 'Ci vorrebbe proprio
//     una carota']
```

---

In questo listato, si utilizza la parola chiave `this` per produrre il tipo di oggetto (in questo caso `rabbit`, un coniglio) che parla. Si ricordi che i metodi `apply` e `bind` accettano un primo argomento che può essere utilizzato per simulare chiamate di metodo. Questo primo argomento viene qui usato per assegnare un valore a `this`.

Esiste un metodo simile ad `apply`, chiamato `call`. Anche `call` richiama la funzione di cui è un metodo, ma accetta argomenti normalmente e non come array. Come con `apply` e `bind`, a `call` si può passare un valore specifico per `this`.

---

```
speak.apply(fatRabbit, ["Burp!"]);
// → The fat rabbit says 'Burp!' [Il coniglio grasso dice 'Burp!']

speak.call({type: "old"}, "Oh my.");
// → The old rabbit says 'Oh my.' [Il coniglio vecchio dice 'Oh my!']
```

---

# Prototipi

Si esamini il codice seguente:

---

```
var empty = {};
console.log(empty.toString());
// → function toString(){...}
console.log(empty.toString());
// → [object Object]
```

---

Con questo, ho tirato fuori una proprietà da un oggetto vuoto. Magico!

In realtà non è proprio così. È solo perché non ho ancora rivelato come funzionano gli oggetti in JavaScript. Oltre alle loro proprietà, quasi tutti gli oggetti hanno un *prototipo*. Un prototipo è un altro oggetto, che viene utilizzato come “sorgente” di proprietà. Quando un oggetto riceve una richiesta per una proprietà che non ha, la proprietà viene cercata nel suo prototipo; se non la si trova nemmeno lì, la ricerca continua nel prototipo del prototipo e così via.

Ma chi è il prototipo dell’oggetto vuoto? È il prototipo ancestrale, l’entità che sta dietro quasi tutti gli oggetti, `Object.prototype`.

---

```
console.log(Object.getPrototypeOf({}) ==
            Object.prototype);
// → true
console.log(Object.getPrototypeOf(Object.prototype));
// → null
```

---

Come potete immaginare, la funzione `Object.getPrototypeOf` restituisce il prototipo di un oggetto.

La relazione tra prototipi e oggetti in JavaScript forma una struttura ad albero, dove alla radice si trova `Object.prototype`. Quest’oggetto offre alcuni metodi disponibili per tutti gli oggetti, come `toString`, che converte l’oggetto nella sua rappresentazione in formato stringa.

Alcuni oggetti non dispongono di un prototipo definito direttamente in `Object.prototype`, ma ottengono le proprietà predefinite da un altro oggetto. Tutte le funzioni derivano da `Function.prototype` e tutti gli array da `Array.prototype`.

---

```
console.log(Object.getPrototypeOf(isNaN) ==
            Function.prototype);
// → true
console.log(Object.getPrototypeOf([]) ==
            Array.prototype);
```

```
// → true
```

---

Questo tipo di oggetti prototipo ha a sua volta un prototipo, spesso `Object.prototype`, e può pertanto offrire indirettamente metodi come `toString`.

La funzione `Object.getPrototypeOf` restituisce il prototipo di un oggetto. Per creare un oggetto con un prototipo specifico, si usa `Object.create`.

---

```
var protoRabbit = {
  speak: function(line) {
    console.log("The " + this.type + " rabbit says '" +
      line + "'");
  }
};

var killerRabbit = Object.create(protoRabbit);
killerRabbit.type = "killer";
killerRabbit.speak("SKREEEE!");
// → The killer rabbit says 'SKREEEE!' [Il coniglio assassino dice
//   'SKREEEE!']
```

---

Il “coniglio prototipico” dell’esempio funge da contenitore per le proprietà condivise da tutti gli oggetti coniglio. I singoli oggetti coniglio, come il coniglio assassino (`killerRabbit`) dell’esempio, dispongono di proprietà specifiche valide solo per se stessi (in questo caso il tipo, `killer`) e derivano dal prototipo le proprietà condivise.

## Costruttori

I *costruttori* offrono un modo più semplice per creare oggetti derivati da un prototipo condiviso. In JavaScript, quando si chiama una funzione con la parola chiave `new` la si tratta come costruttore. Nel costruttore, la variabile `this` è legata a un nuovo oggetto, che sarà restituito dalla chiamata se non è previsto esplicitamente un diverso oggetto come valore di restituzione.

Un oggetto creato con la parola chiave `new` si chiama *istanza* del proprio costruttore.

L’esempio che segue dimostra un semplice costruttore per oggetti `rabbit`. Per convenzione, i nomi dei costruttori si scrivono con l’iniziale maiuscola per distinguerli dalle altre funzioni.

---

```
function Rabbit(type) {
  this.type = type;
}

var killerRabbit = new Rabbit("killer");
var blackRabbit = new Rabbit("black");
```

```
console.log(blackRabbit.type);
// → black
```

---

I costruttori, così come tutte le funzioni, dispongono automaticamente di una proprietà, chiamata `prototype`, che contiene un oggetto predefinito, vuoto, derivato da `Object.prototype`. Tutte le istanze create con questo costruttore avranno come prototipo questo oggetto. Pertanto, per aggiungere un metodo `speak` agli oggetti coniglio creati col costruttore `Rabbit`, sarà sufficiente scrivere:

```
Rabbit.prototype.speak = function(line) {
  console.log("The " + this.type + " rabbit says '" +
    line + "'");
}
blackRabbit.speak("Doom...");
// → The black rabbit says 'Doom...'
```

---

È importante capire la differenza tra come un prototipo sia associato a un costruttore (attraverso la sua proprietà `prototype`) e come gli oggetti abbiano un loro prototipo (che si recupera con `Object.getPrototypeOf`). In effetti, poiché i suoi costruttori sono funzioni, il prototipo di un costruttore è `Function.prototype`. La sua *proprietà* `prototype` sarà il prototipo delle istanze create attraverso il costruttore, ma non il suo stesso prototipo.

## Prevalenza sulle proprietà derivate

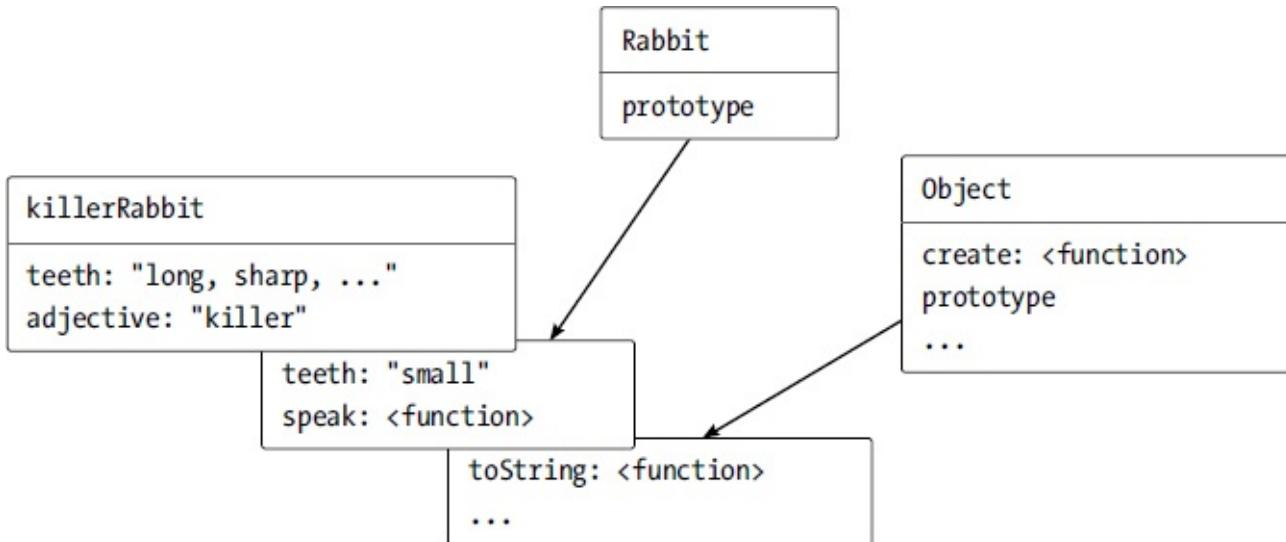
Quando a un oggetto si aggiunge una proprietà, non importa se presente nel prototipo o meno, la proprietà viene aggiunta all'oggetto stesso, il quale a sua volta la otterrà come proprietà propria. Se il prototipo dispone di una proprietà con lo stesso nome, questa non avrà effetto sull'oggetto. Il prototipo rimane invariato.

```
Rabbit.prototype.teeth = "small";
console.log(killerRabbit.teeth);
// → small
killerRabbit.teeth = "long, sharp, and bloody";
console.log(killerRabbit.teeth);
// → long, sharp, and bloody
console.log(blackRabbit.teeth);
// → small
console.log(Rabbit.prototype.teeth);
// → small
```

---

Lo schema qui sotto mostra la situazione dopo l'esecuzione del codice. I prototipi

Rabbit e Object fanno da scenario per killerRabbit, e si possono esaminare nei prototipi le proprietà che non si trovano nell'oggetto stesso.



Prevalere sulle proprietà esistenti in un prototipo può essere molto utile. Come dimostra l'esempio precedente, dove si definiscono i denti del coniglio, la prevalenza consente di esprimere delle proprietà eccezionali per alcune istanze di una classe di oggetti più generica, mentre gli oggetti “non eccezionali” recuperano il valore normale dal loro prototipo.

La prevalenza viene utilizzata anche per assegnare ai prototipi delle funzioni e degli array un metodo `toString` diverso da quello del prototipo di base.

---

```
console.log(Array.prototype.toString ==  
          Object.prototype.toString);  
// → false  
console.log([1, 2].toString());  
// → 1, 2
```

---

Nell'esempio, chiamare `toString` per un array dà un risultato simile a quello di `.join(",")`: inserisce delle virgole tra i valori dell'array. Richiamare direttamente `Object.prototype.toString` con un array dà come risultato una stringa diversa: poiché questa funzione non sa nulla dell'array, si limita a riportare tra parentesi quadre la parola “object” e il nome del tipo.

---

```
console.log(Object.prototype.toString.call([1, 2]));  
// → [object Array]
```

---

## Interferenza dei prototipi

In qualunque momento, si può usare un prototipo per aggiungere nuove proprietà e nuovi metodi a tutti gli oggetti basati su di esso. Per esempio, immaginiamo che i conigli

debbano danzare:

---

```
Rabbit.prototype.dance = function() {  
    console.log("The " + this.type + " rabbit dances a jig.");  
};  
killerRabbit.dance();  
// → The killer rabbit dances a jig.
```

---

Comodo! Ci sono, però, situazioni dove ciò può dar problemi. Nei capitoli precedenti, usavamo un oggetto per associare dei valori a dei nomi, creando delle proprietà per i nomi e assegnando loro come valore i valori corrispondenti. Ecco un esempio tratto dal [Capitolo 4](#):

---

```
var map = {};  
function storePhi(event, phi) {  
    map[event] = phi;  
}  
storePhi("pizza", 0.069);  
storePhi("touched tree", -0.081);
```

---

Per iterare sui valori phi dell'oggetto, si usa un ciclo `for/in` e si verifica la presenza di un nome con l'operatore `in` regolare. Purtroppo, in questo caso il prototipo dell'oggetto si mette di mezzo:

---

```
Object.prototype.nonsense = "hi";  
for (var name in map)  
    console.log(name);  
// → pizza  
// → touched tree  
// → nonsense  
console.log("nonsense" in map);  
// → true  
console.log("toString" in map);  
// → true  
// Elimina la proprietà che dà problemi  
delete Object.prototype.nonsense;
```

---

Il che è sbagliato. Nell'insieme di dati non c'è un evento chiamato “nonsense”, né tantomeno uno che si chiami “toString”.

Stranamente, `toString` non compare nel ciclo `for/in`, eppure l'operatore `in` ha restituito `true`. Questo perché JavaScript distingue tra proprietà *enumerabili* e proprietà

*non enumerabili.*

Tutte le proprietà create con delle semplici assegnazioni sono enumerabili. Le proprietà standard di `Object.prototype`, invece, sono tutte non enumerabili, il che spiega perché non compaiano nel ciclo `for/in` dell'esempio.

Possiamo definire le proprietà non enumerabili che ci interessano con la funzione `Object.defineProperty`, che consente di controllare il tipo della proprietà che vogliamo creare.

---

```
Object.defineProperty(Object.prototype, "hiddenNonsense",
    {enumerable: false, value: "hi"});

for (var name in map)
    console.log(name);
// → pizza
// → touched tree
console.log(map.hiddenNonsense);
// → hi
```

---

Ecco che la proprietà esiste, ma non compare nei cicli. E va bene. Abbiamo, però, ancora un problema con l'operatore `in` regolare, che ritiene che le proprietà di `Object.prototype` esistano nel nostro oggetto. La soluzione sta nel metodo `hasOwnProperty` dell'oggetto.

```
console.log(map.hasOwnProperty("toString"));
// → false
```

---

Il metodo indica se l'oggetto stesso ha la proprietà richiesta, senza andare a guardare i suoi prototipi. Il che è un'informazione più utile di quella che avevamo ottenuto con l'operatore `in`. Se temete che qualche brano del codice abbia interferito col prototipo-base di un oggetto, potete impostare come segue i cicli `for/in`:

```
for (var name in map) {
    if (map.hasOwnProperty(name)) {
        // ... questa è una proprietà propria
    }
}
```

---

## Oggetti senza prototipo

C'è di più. Ammettiamo che qualcuno abbia registrato il nome `hasOwnProperty` nel nostro oggetto `map` e ne abbia impostato il valore su 42. Ora, la chiamata a `map.hasOwnProperty`

cercherà di richiamare la proprietà locale, che mantiene un numero e non una funzione.

In casi come questo, i prototipi impiccano; meglio avere oggetti senza prototipo. Abbiamo già visto la funzione `Object.create`, che consente di creare un oggetto con un prototipo specifico. Passando `null` come prototipo, si crea un nuovo oggetto senza prototipo. Per gli oggetti come `map`, dove le proprietà possono essere qualunque cosa (e di qualunque tipo), questa è la soluzione migliore.

---

```
var map = Object.create(null);
map["pizza"] = 0.069;
console.log("toString" in map);
// → false
console.log("pizza" in map);
// → true
```

---

Molto meglio! A questo punto, non serve più la soluzione sporca con `hasOwnProperty`, perché tutte le proprietà dell'oggetto sono solo sue. Possiamo ora usare tranquillamente cicli `for/in`, senza più doverci preoccupare di `Object.prototype`.

## Polimorfismo

Quando richiamate su un oggetto la funzione `String`, che converte un valore in tipo stringa, essa richiama il metodo `toString` su quell'oggetto per tentare di produrre una stringa sensata. Ho già detto che alcuni dei prototipi standard definiscono la propria versione di `toString`, così da creare stringhe che offrono informazioni più significative di "[object Object]".

Questo è un semplice esempio di un'idea fenomenale. Quando si imposta un brano di codice in modo che abbia effetto su oggetti che offrono una certa interfaccia, in questo caso, un metodo `toString`, il codice vale e funziona per qualunque oggetto offra supporto alla stessa interfaccia.

Questa tecnica si chiama *polimorfismo*, sebbene non abbia a che vedere direttamente con le mutazioni di forma. Il codice polimorfico è in grado di funzionare con valori di forma diversa, a condizione che offrano supporto per l'interfaccia prevista.

## Impostare una tabella

Esaminiamo ora un esempio un pochino più complesso, che illustra gli aspetti più importanti del polimorfismo e della programmazione orientata agli oggetti.

Il progetto da sviluppare è un programma che, dato un array di celle, costruisce una stringa che contiene una tabella ben strutturata, ossia con i dati ben allineati in righe e colonne, come la seguente:

---

nome	altezza	paese
Kilimanjaro	5895	Tanzania
Everest	8848	Nepal
Mount Fuji	3776	Japan
Mont Blanc	4808	Italy/France
Vaalserberg	323	Netherlands
Denali	6168	United States
Popocatepetl	5465	Mexico

---

Il sistema per realizzare la tabella prevede che la funzione costruttrice chieda a ciascuna cella quanto debba essere larga e alta, e userà questi dati per determinare la larghezza delle colonne e l'altezza delle righe. La funzione costruttrice chiederà quindi alle celle di disegnarsi della corretta misura e raccoglierà i risultati in una stringa unica.

Il programma di layout comunicherà con gli oggetti cella attraverso un'interfaccia ben definita. Per questo, il tipo di cella da elaborare non viene stabilito in partenza. Si potranno così aggiungere nuovi stili di cella in un secondo tempo, per esempio delle celle sottolineate per le testatina della tabella, che funzioneranno senza bisogno di cambiare il programma, alla sola condizione di offrire supporto per la stessa interfaccia.

Ecco le definizioni dell'interfaccia:

- `minHeight()` restituisce un numero che indica l'altezza minima (in righe) richiesta per la cella.
- `minWidth()` restituisce un numero che indica la larghezza minima della cella (in caratteri).
- `draw(width, height)` restituisce un array di lunghezza `height`, che contiene una serie di stringhe, ciascuna di `width` caratteri di larghezza. Ciò rappresenta il contenuto della cella.

In quest'esempio farò grande uso di metodi di array di ordine superiore, in quanto il problema da risolvere si presta particolarmente bene a quel tipo di impostazione.

La prima parte del programma calcola array di larghezze di colonna minime e altezze di riga minime, per una griglia di celle. La variabile `rows` contiene un array di array, dove ciascuno degli array interni rappresenta una riga di celle.

---

```
function rowHeights(rows) {
  return rows.map(function(row) {
    return row.reduce(function(max, cell) {
      return Math.max(max, cell.minHeight());
    }, 0);
  });
}
```

```

}

function colWidths(rows) {
  return rows[0].map(function(_, i) {
    return rows.reduce(function(max, row) {
      return Math.max(max, row[i].minWidth());
    }, 0);
  });
}

```

---

Premettere il segno di sottolineatura (\_) al nome della variabile, o usarlo come nome di una variabile senza altri caratteri, serve a indicare (a chi legge) che quest'argomento non sarà usato.

La funzione `rowHeights` dovrebbe essere abbastanza intuitiva. Fa uso di `reduce` per calcolare l'altezza massima di un array di celle e lo avvolge in `map` per ripetere i calcoli per tutte le righe dell'array `rows`.

La funzione `colWidths` è un po' più complessa, in quanto l'array esterno è un array di righe e non di colonne. In effetti, non ho spiegato che `map` (così come `forEach`, `filter` e altri simili metodi di array) passa un secondo argomento alla sua funzione: l'indice dell'elemento corrente. Inserendo nella mappa gli elementi della prima riga e usando solo il secondo argomento della funzione `map`, `colWidths` costruisce un array con un solo elemento per ciascun indice di colonna. La chiamata a `reduce` esegue la funzione per ciascuno degli indici dell'array `rows` esterno e cattura la larghezza della colonna più larga.

Ecco il codice per tracciare una tabella:

---

```

function drawTable(rows) {
  var heights = rowHeights(rows);
  var widths = colWidths(rows);
  function drawLine(blocks, lineNo) {
    return blocks.map(function(block) {
      return block[lineNo];
    }).join(" ");
  }
  function drawRow(row, rowNum) {
    var blocks = row.map(function(cell, colNum) {
      return cell.draw(widths[colNum], heights[rowNum]);
    });
    return blocks[0].map(function(_, lineNo) {
      return drawLine(blocks, lineNo);
    }).join("\n");
  }
}

```

```

    }
    return rows.map(drawRow).join("\n");
}

```

---

La funzione `drawTable` utilizza la funzione ausiliaria interna `drawRow` per tracciare tutte le righe e le raccoglie tutte insieme, inserendo un a capo tra l'una e l'altra.

La funzione `drawRow` prima converte in *blocchi* gli oggetti cella della riga; i blocchi sono array di stringhe che rappresentano il contenuto di ogni cella, divise per riga. Una sola cella che contenga solo il numero 3776 potrebbe essere rappresentata da un array di un solo elemento come `["3776"]`, mentre una cella sottolineata può occupare due righe ed essere rappresentata dall'array `["name", "—"]`.

I blocchi di una riga, che hanno tutti la stessa altezza, devono risultare affiancati nel risultato finale. La seconda chiamata a `map`, in `drawRow`, costruisce l'output riga per riga, applicando la funzione `map` sulle righe del blocco più a sinistra e, per ciascuna di esse, raccogliendo una riga larga quanto la larghezza totale della tabella. Queste righe vengono poi unite con caratteri di a capo per realizzare la riga della tabella come valore di restituzione di `drawRow`.

La funzione `drawLine` estrae da un array di blocchi le righe che devono risultare affiancate e le unisce con uno spazio, per creare un intervallo di un carattere tra le colonne della tabella.

Scriviamo ora un costruttore per le celle che contengono testo, che implementi l'interfaccia per le celle della tabella. Il costruttore spezza una stringa in un array di righe col metodo `split` di `string`, che interrompe una stringa ogni volta che incontra l'argomento dato e restituisce un array dei singoli pezzi. Il metodo `minWidth` trova la larghezza massima delle righe di questo array.

```

function repeat(string, times) {
  var result = "";
  for (var i = 0; i < times; i++)
    result += string;
  return result;
}

function TextCell(text) {
  this.text = text.split("\n");
}

TextCell.prototype.minLength = function() {
  return this.text.reduce(function(width, line) {
    return Math.max(width, line.length);
  }, 0);
};

```

```

TextCell.prototype.minLength = function() {
    return this.text.length;
};

TextCell.prototype.draw = function(width, height) {
    var result = [];
    for (var i = 0; i < height; i++) {
        var line = this.text[i] || "";
        result.push(line + repeat(" ", width - line.length));
    }
    return result;
};

```

---

Nel codice si usa la funzione ausiliaria `repeat` per costruire una stringa il cui valore è l'argomento di `string`, ripetuto `times` volte. Il metodo `draw` la usa per “imbottire” le righe, in modo che abbiano tutte la stessa lunghezza.

Proviamo ora il codice per costruire una scacchiera 5x5:

---

```

var rows = [];
for (var i = 0; i < 5; i++) {
    var row = [];
    for (var j = 0; j < 5; j++) {
        if ((j + i) % 2 == 0)
            row.push(new TextCell("##"));
        else
            row.push(new TextCell(" "));
    }
    rows.push(row);
}
console.log(drawTable(rows));
// → ##      ##      ##
//      ##      ##
//  ##      ##      ##
//      ##      ##
//  ##      ##      ##

```

---

Funziona! Poiché, però, tutte le celle hanno le stesse dimensioni, il codice che imposta la tabella non svolge nessun lavoro interessante.

I dati per la tabella delle montagne, oggetto di quest'esempio, si trovano nella variabile `MOUNTAINS` e nei file scaricabili dal sito, <http://eloquentjavascript.net/code/>.

Per mettere in risalto la prima riga, che contiene i nomi delle colonne, vogliamo sottolineare le celle con una serie di trattini. Non c'è problema, basta impostare un tipo di cella per le sottolineature.

---

```
function UnderlinedCell(inner) {
  this.inner = inner;
}
UnderlinedCell.prototype.minLength = function() {
  return this.inner.minLength();
};
UnderlinedCell.prototype.minLength = function() {
  return this.inner.minLength() + 1;
};
UnderlinedCell.prototype.draw = function(width, height) {
  return this.inner.draw(width, height - 1)
    .concat([repeat("-", width)]);
};
```

---

Una cella sottolineata contiene un'altra cella. Ne riporta le dimensioni minime come uguali a quelle della sua cella interna (attraverso la chiamata ai metodi `minWidth` e `minHeight` di quella cella), ma aggiunge uno all'altezza per tener conto dello spazio occupato dal carattere usato per la sottolineatura.

Per tracciare questa cella, basta prendere il contenuto della cella interna e concatenarvi una riga piena di trattini.

Adesso che abbiamo il meccanismo per la sottolineatura, possiamo scrivere una funzione che costruisca una griglia di celle con i dati raccolti.

---

```
function dataTable(data) {
  var keys = Object.keys(data[0]);
  var headers = keys.map(function(name) {
    return new UnderlinedCell(new TextCell(name));
  });
  var body = data.map(function(row) {
    return keys.map(function(name) {
      return new TextCell(String(row[name]));
    });
  });
  return [headers].concat(body);
}
```

```
console.log(drawTable(dataTable(MOUNTAINS)));
// → name           height country
// → -----
//   Kilimanjaro  5895    Tanzania
//   ... eccetera
```

---

La funzione standard `Object.keys` restituisce un array di nomi di proprietà per un oggetto. La prima riga della tabella deve contenere delle celle sottolineate che danno il nome alle colonne. Sotto di essa, i valori degli oggetti dell'insieme di dati sono visualizzati come celle normali: le estraiamo passando la funzione `map` sull'array `keys`, così da esser certi che l'ordine delle celle sia lo stesso per tutte le righe.

La tabella che si ottiene è simile a quella riportata in precedenza, con la differenza che non allinea a destra i numeri della colonna `height`. A quello arriveremo tra un momento.

## Recuperare e impostare

Quando specifichiamo un'interfaccia, possiamo includere delle proprietà che non sono metodi. Per esempio, potevamo definire `minHeight` e `minwidth` semplicemente per mantenere dei numeri, ma ciò avrebbe richiesto di calcolarli nel costruttore, il che in questo caso aggiunge del codice che non è strettamente necessario per *costruire* l'oggetto. Per esempio, potrebbe creare problemi se cambiasse la cella interna di una cella sottolineata, in quanto in quel caso dovrebbe cambiare anche la cella sottolineata.

Queste considerazioni hanno indotto alcuni ad adottare il principio di non includere mai, nelle interfacce, delle proprietà che non siano metodi. Invece di accedere direttamente a una semplice proprietà valore, questi programmati usano metodi `getSomething` e `setSomething` rispettivamente per leggere e scrivere la proprietà. Questo approccio ha lo svantaggio di far scrivere, e leggere, più metodi del solito.

Per fortuna, JavaScript offre una tecnica che dà il meglio dei due mondi: possiamo, infatti, specificare proprietà che, dall'esterno, sembrano normali, ma hanno dei metodi nascosti associati a esse.

```
var pile = {
  elements: ["eggshell", "orange peel", "worm"],
  get height() {
    return this.elements.length;
  },
  set height(value) {
    console.log("Ignoring attempt to set height to", value);
  }
};
```

```
console.log(pile.height);
// → 3
pile.height = 100;
// → Ignoring attempt to set height to 100
```

---

Nel linguaggio degli oggetti, la notazione get o set per le proprietà consente di specificare se la funzione va eseguita quando la proprietà viene letta o quando viene scritta. Possiamo aggiungere questa proprietà anche a oggetti esistenti, per esempio a un prototipo, attraverso la funzione `Object.defineProperty` (che avevamo usato per creare delle proprietà non enumerabili).

```
Object.defineProperty(TextCell.prototype, "heightProp", {
  get: function() { return this.text.length; }
});
var cell = new TextCell("no\nway");
console.log(cell.heightProp);
// → 2
cell.heightProp = 100;
console.log(cell.heightProp);
// → 2
```

---

Nell'oggetto passato a `defineProperty`, si può usare una simile proprietà set per specificare un metodo di impostazione. Se definiamo un metodo get senza un corrispondente metodo set, il tentativo di scrivere la proprietà viene ignorato.

## Ereditarietà

L'esercizio di impostazione della tabella non è ancora finito. Allineare a destra le colonne contenenti numeri le rende più facili da leggere. Dobbiamo pertanto impostare un nuovo tipo di cella, simile a `TextCell`, ma che preveda di riempire le celle a sinistra, invece che a destra, in modo che il contenuto sia allineato a destra.

Potremmo semplicemente scrivere un nuovo costruttore con tutti e tre i metodi nel suo prototipo. Siccome i prototipi possono avere a loro volta dei prototipi, possiamo trovare una soluzione ancora più intelligente.

```
function RTextCell(text) {
  TextCell.call(this, text);
}
RTextCell.prototype = Object.create(TextCell.prototype);
RTextCell.prototype.draw = function(width, height) {
  var result = [];
```

```

for (var i = 0; i < height; i++) {
  var line = this.text[i] || "";
  result.push(repeat(" ", width - line.length) + line);
}
return result;
};

```

---

Riutilizziamo il costruttore e i metodi `minHeight` e `minWidth` dell'oggetto `TextCell` normale. Un oggetto `RTextCell` è praticamente equivalente a `TextCell`, salvo che il suo metodo `draw` contiene una funzione diversa.

Questa struttura si chiama *ereditarietà*. L'ereditarietà consente di costruire, con relativamente poco lavoro, dei tipi di dati leggermente diversi, a partire da tipi di dati esistenti. Di solito, il nuovo costruttore chiamerà il vecchio costruttore (attraverso il metodo `call`, in modo da poter assegnare il nuovo oggetto come valore per `this`). Una volta chiamato il nuovo costruttore, possiamo supporre che siano stati aggiunti tutti i campi che conteneva il vecchio tipo di oggetto. Facciamo poi in modo che il prototipo del costruttore derivi dal vecchio prototipo, in modo che le istanze di questo tipo abbiano tutte accesso alle proprietà di quel prototipo. Infine, prevaliamo su alcune delle proprietà aggiungendole al nuovo prototipo.

Modificando leggermente la funzione `dataTable` per usare `RTextCells` quando il valore delle celle è un numero, otteniamo la tabella che volevamo:

```

function dataTable(data) {
  var keys = Object.keys(data[0]);
  var headers = keys.map(function(name) {
    return new UnderlinedCell(new TextCell(name));
  });
  var body = data.map(function(row) {
    return keys.map(function(name) {
      var value = row[name];
      // This was changed:
      if (typeof value == "number")
        return new RTextCell(String(value));
      else
        return new TextCell(String(value));
    });
  });
  return [headers].concat(body);
}

```

```
console.log(drawTable(dataTable(MOUNTAINS)));
// → ... tabella perfettamente allineata
```

---

Insieme all'incapsulazione e al polimorfismo, l'ereditarietà è una componente fondamentale della tradizione orientata agli oggetti. Mentre, però, i primi due concetti vengono generalmente considerati molto utili, l'ereditarietà è talvolta oggetto di critiche.

La ragione principale è che viene spesso confusa col polimorfismo: viene descritta come uno strumento più potente di quanto non sia in realtà ed è spesso utilizzata a sproposito. Mentre encapsulazione e polimorfismo vengono usati per separare tra loro brani di codice, riducendo così la complessità del programma nel suo insieme, l'ereditarietà in sostanza lega insieme i tipi creando più confusione.

Come abbiamo visto, si può avere polimorfismo senza ereditarietà. Non voglio dire che bisogna assolutamente evitarla: anzi, io la uso regolarmente nei miei programmi. Va vista, però, come un trucchetto un po' sporco, che può aiutare a definire nuovi tipi scrivendo poco codice, e non come un principio capitale di organizzazione del codice. Per estendere i tipi, può essere meglio optare per la composizione, così come abbiamo visto in `UnderlinedCell`, che estende un altro oggetto cella semplicemente memorizzandovi una proprietà e ritrasmettendovi le chiamate ai metodi nei propri metodi.

## L'operatore instanceof

A volte può essere utile sapere se un oggetto deriva da un costruttore specifico. Per questo, JavaScript offre l'operatore binario `instanceof`.

```
console.log(new RTextCell("A") instanceof RTextCell);
// → true

console.log(new RTextCell("A") instanceof TextCell);
// → true

console.log(new TextCell("A") instanceof RTextCell);
// → false

console.log([1] instanceof Array);
// → true
```

---

L'operatore analizza i tipi ereditati. Una cella `RTextCell` è un'istanza di `TextCell` perché `RTextCell.prototype` deriva da `TextCell.prototype`. Quest'operatore si applica ai costruttori standard, come `Array`. Quasi tutti gli oggetti sono istanze di `Object`.

## Riepilogo

Gli oggetti sono, in effetti, più complessi di come avevo detto all'inizio. Hanno prototipi, che sono altri oggetti, e si comportano come se avessero proprietà che non hanno, se il

loro prototipo ha quelle proprietà. Gli oggetti semplici hanno come prototipo `Object.prototype`.

I costruttori, che sono funzioni i cui nomi di solito iniziano con una lettera maiuscola, si usano con l'operatore `new` per creare nuovi oggetti. Un buon modo per applicare questo concetto è di aggiungere al loro prototipo tutte le proprietà condivise dai valori di un certo tipo. Dato un oggetto e un costruttore, l'operatore `instanceof` indica se quell'oggetto è un'istanza del costruttore dato.

Un buon modo per usare gli oggetti è di specificare un'interfaccia specifica e dare chiare disposizioni che a un certo oggetto si parla solo attraverso l'interfaccia specificata. In questo modo, tutti gli altri aspetti dell'oggetto sono *incapsulati*, ossia nascosti dietro l'interfaccia.

A proposito di interfacce, chi dice che un solo tipo di oggetto può implementare una certa interfaccia? Quando più oggetti espongono la stessa interfaccia e si ha del codice che agisce su qualunque oggetto che abbia quell'interfaccia, si ha il polimorfismo. Un concetto molto utile.

Per implementare più tipi che differiscono solo in particolari minimi, si può far derivare il prototipo del nuovo tipo dal prototipo di un tipo preesistente e far chiamare il nuovo tipo dal costruttore del nuovo tipo. Con questo si ottiene un tipo di oggetto simile all'oggetto preesistente, ma dove si possono aggiungere proprietà, o prevalere su di esse, come necessario.

## Esercizi

### **Un tipo vettore**

Scrivete un costruttore `Vector` che rappresenti un vettore in uno spazio bidimensionale. Il vettore accetta parametri `x` e `y` (numeri) e li salva in proprietà dello stesso nome.

Assegnate al prototipo `Vector` due metodi, `plus` e `minus`, che accettino un altro vettore come parametro e restituiscano un nuovo vettore con la somma o la differenza dei valori `x` e `y` dei due vettori (quello in `this` e il parametro).

Aggiungete al prototipo una proprietà che recuperi `length` per calcolare la lunghezza del vettore; ossia, la distanza del punto  $(x, y)$  dall'origine  $(0, 0)$ .

### **Un'altra cella**

Definite un tipo di cella chiamato `StretchCell(inner, width, height)` conforme all'interfaccia per le celle della tabella descritta in questo capitolo. La cella deve avvolgere un'altra cella (come fa `UnderlinedCell`) e fare in modo che la cella risultante abbia almeno i valori dati per `width` e `height`, anche se la cella interna fosse per sua natura più piccola.

### **Interfaccia per sequenze**

Progettate *un'interfaccia* per l'astrazione di un'iterazione su un insieme di valori. Un oggetto che offre quest'interfaccia rappresenta una sequenza e l'interfaccia deve consentire, al codice che utilizza l'oggetto, di iterare lungo la sequenza, esaminando i valori dell'elemento e riconoscendo quando si arriva al termine della sequenza.

Una volta specificata l'interfaccia, scrivete una funzione `logFive` che accetti un oggetto sequenza e chiami `console.log` sui suoi primi cinque elementi, o meno se la sequenza ne ha meno di cinque.

Scrivete quindi un tipo di oggetto `ArraySeq` che avvolga un array e consenta di iterare sull'array attraverso l'interfaccia ottenuta. Impostate infine un tipo di oggetto `RangeSeq` che iteri su un intervallo di interi, accettando nel costruttore argomenti `from` e `to`.

## PROGETTO: VITA ELETTRONICA

*[...] chiedersi se le macchine possano pensare [...] ha lo stesso senso di chiedersi se i sottomarini possano nuotare.*

Edsger Dijkstra, *The Threats to Computing Science*

Nei capitoli dedicati ai progetti, smetterò per un attimo di subissarvi di nuove teorie e lavorerò invece con voi su un programma. La teoria è indispensabile quando s’impara a programmare, ma dovrebbe essere accompagnata dal leggere e comprendere programmi di un certo significato.

Il nostro progetto di questo capitolo è la realizzazione di un ecosistema virtuale, un piccolo mondo popolato da creaturine che si spostano e lottano per sopravvivere.

### Definizione

Per poter affrontare il compito, dovremo semplificare grandemente il concetto di *mondo*. Per la precisione, il nostro mondo sarà una griglia bidimensionale dove ogni entità occupa un’intera casella. A ogni *turno*, tutte le creature avranno la possibilità di svolgere una qualche azione.

Pertanto, dividiamo tempo e spazio in unità di dimensioni fisse: riquadri per lo spazio e turni per il tempo. Naturalmente si tratta di un’approssimazione grossolana e inaccurata. La nostra simulazione, però, dev’essere divertente, non accurata: possiamo pertanto decidere liberamente.

Possiamo definire il mondo con un *piano* (*plan*), un array di stringhe che definisce la griglia del mondo usando un carattere per casella:

---

```
var plan =  
["#####",  
 "# # O ##",  
 "# #",  
 "# ##### #",  
 "## # # ## #",  
 "### ## # # #",  
 "# #### #",  
 "# ## #",  
 "# ## O ##",  
 "# O # O ### #",  
 "# #",  
 "#####"];
```

---

I caratteri # rappresentano muri e rocce, mentre i caratteri o rappresentano le creature. Gli spazi sono proprio spazio vuoto.

Per creare un oggetto `World`, si può usare un array `plan`. L'oggetto tiene nota delle dimensioni e del contenuto del mondo. Dispone di un metodo `toString`, che riconverte il mondo in stringa stampabile (simile al piano su cui si basa), così che possiamo vedere quel che succede al suo interno. L'oggetto `World` ha anche un metodo `turn`, che fa muovere le creature quando viene il momento e aggiorna il mondo in base a quel che esse fanno.

## Rappresentazione dello spazio

La griglia che modella il mondo ha larghezza e altezza fisse. Le caselle sono identificate dalle loro coordinate x e y. Usiamo un semplice tipo vettore, `Vector` (come abbiamo visto negli esercizi del capitolo precedente), per rappresentare le coppie di coordinate.

---

```
function Vector(x, y) {  
    this.x = x;  
    this.y = y;  
}  
Vector.prototype.plus = function(other) {  
    return new Vector(this.x + other.x, this.y + other.y);  
};
```

---

Ci serve poi un tipo di oggetto che formi la griglia stessa. La griglia `grid` fa parte di un mondo, ma la definiamo come oggetto distinto (che sarà una proprietà dell'oggetto `World`) per non complicare il mondo stesso. Il mondo deve preoccuparsi delle cose del mondo e la griglia di quelle che la riguardano.

Abbiamo alcune opzioni disponibili per memorizzare una griglia di valori. Possiamo

usare un array di array di righe per raggiungere una certa casella accedendo a due proprietà, come segue:

---

```
var grid = [["top left", "top middle", "top right"],  
            ["bottom left", "bottom middle", "bottom right"]];  
console.log(grid[1][2]);  
// → bottom right
```

---

Oppure possiamo usare un solo array, di dimensioni `width × height`, e stabilire che l'elemento in  $(x,y)$  si trova nella posizione  $x + (y \times width)$  dell'array.

---

```
var grid = ["top left", "top middle", "top right",  
            "bottom left", "bottom middle", "bottom right"];  
console.log(grid[2 + (1 * 3)]);  
// → bottom right
```

---

Poiché l'accesso all'array sarà avvolto nei metodi che intervengono sull'oggetto di tipo griglia, la soluzione che sceglieremo non avrà importanza per il codice esterno. Io preferisco la seconda rappresentazione, perché semplifica di molto la creazione dell'array. Quando si chiama il costruttore `Array` con un solo numero come argomento, si crea un nuovo array vuoto della lunghezza data.

Il codice che segue definisce l'oggetto `Grid` con alcuni metodi di base:

---

```
function Grid(width, height) {  
    this.space = new Array(width * height);  
    this.width = width;  
    this.height = height;  
}  
Grid.prototype.isInside = function(vector) {  
    return vector.x >= 0 && vector.x < this.width &&  
        vector.y >= 0 && vector.y < this.height;  
};  
Grid.prototype.get = function(vector) {  
    return this.space[vector.x + this.width * vector.y];  
};  
Grid.prototype.set = function(vector, value) {  
    this.space[vector.x + this.width * vector.y] = value;  
};
```

---

Ed ecco una piccola prova:

---

```
var grid = new Grid(5, 5);
console.log(grid.get(new Vector(1, 1)));
// → undefined
grid.set(new Vector(1, 1), "X");
console.log(grid.get(new Vector(1, 1)));
// → X
```

---

## L'interfaccia di programmazione delle creature

Prima di affrontare il costruttore di `World`, dobbiamo essere più specifici sugli oggetti creatura che vivranno al suo interno. Il mondo chiederà alle creature che azione vogliono svolgere in questo modo: ogni oggetto creatura (`Critter`) ha un metodo `act` che restituisce *un'azione* quando viene chiamato. Le azioni sono a loro volta oggetti con una proprietà `type`, che definisce il tipo di azione che la creatura intende svolgere, per esempio "`move`" (spostarsi). L'azione può contenere anche altre informazioni, come la direzione di spostamento della creatura.

Le creature sono molto miopi e vedono solo le caselle che confinano con quella dove stanno. Anche questa loro visione limitata, comunque, può essere utile nel decidere quale azione vogliono svolgere. In ogni chiamata al metodo `act` si passa un oggetto `view`, che consente alla creatura di ispezionare i suoi dintorni. Alle otto caselle che rappresentano i dintorni, diamo i nomi delle direzioni della bussola: "`n`" per nord, "`ne`" per nord est e così via. Ecco l'oggetto che useremo per mettere in mappatura i nomi delle direzioni con gli scostamenti delle coordinate:

---

```
var directions = {
  "n": new Vector( 0, -1),
  "ne": new Vector( 1, -1),
  "e": new Vector( 1, 0),
  "se": new Vector( 1, 1),
  "s": new Vector( 0, 1),
  "sw": new Vector(-1, 1),
  "w": new Vector(-1, 0),
  "nw": new Vector(-1, -1)
};
```

---

L'oggetto `view` ha un metodo `look`, che accetta una direzione e restituisce un carattere, per esempio "`#`" quando trova un muro, o " " (spazio) quando nella casella non c'è nulla. L'oggetto offre anche i metodi `find` e `findAll`, che accettano come argomento un carattere della mappa. Il primo metodo restituisce una direzione dove si trova il

carattere, oppure `null` se nelle caselle contigue non lo trova. Il secondo restituisce un array con tutte le direzioni contenenti il carattere dato. Per esempio, per una creatura che si trovi a sinistra (ovest) di un muro, restituirà `["ne", "e", "se"]` se si chiama `findAll` sull'oggetto `view` col carattere `"#"` come argomento.

Ed ecco una creatura un po' stupida che va avanti a naso, finché non incontra un ostacolo e quindi rimbalza via in una direzione a caso:

---

```
function randomElement(array) {
  return array[Math.floor(Math.random() * array.length)];
}

var directionNames = "n ne e se s sw w nw".split(" ");
function BouncingCritter() {
  this.direction = randomElement(directionNames);
};

BouncingCritter.prototype.act = function(view) {
  if (view.look(this.direction) != " ")
    this.direction = view.find(" ") || "s";
  return {type: "move", direction: this.direction};
};
```

---

La funzione ausiliaria `randomElement` sceglie un elemento a caso in un array, recuperando un indice casuale attraverso `Math.random` e qualche altra operazione aritmetica. Riprenderemo questa funzione più avanti, perché i valori casuali possono essere molto utili nelle simulazioni.

Per scegliere una direzione a caso, il costruttore `BouncingCritter` chiama `randomElement` su un array di nomi di direzione. Potevamo usare anche `Object.keys` per recuperare l'array dall'oggetto `directions` definito in precedenza, ma quello non offre nessuna garanzia sull'ordine nel quale le proprietà sono elencate. In molte situazioni, i motori JavaScript moderni restituiscono le proprietà nell'ordine in cui sono state definite, ma questo non è un requisito.

Nel metodo `act`, `|| "s"` ha la funzione di evitare che `this.direction` dia come risultato `null` se in qualche modo la creatura rimane intrappolata, senza via d'uscita (per esempio, quando viene spinta in un angolo da altre creature).

## L'oggetto World

Adesso possiamo cominciare con l'oggetto `World`. Il costruttore accetta come argomenti una variabile `plan` (l'array di stringhe che rappresentano la griglia del mondo, descritta in precedenza) e una `legenda`. L'oggetto `legend` descrive il significato dei caratteri sulla mappa. Contiene un costruttore per ogni carattere, escluso lo spazio, che fa sempre

riferimento a null, il valore che useremo per rappresentare lo spazio vuoto.

---

```
function elementFromChar(legend, ch) {
    if (ch == " ")
        return null;
    var element = new legend[ch]();
    element.originChar = ch;
    return element;
}

function World(map, legend) {
    var grid = new Grid(map[0].length, map.length);
    this.grid = grid;
    this.legend = legend;
    map.forEach(function(line, y) {
        for (var x = 0; x < line.length; x++)
            grid.set(new Vector(x, y),
                     elementFromChar(legend, line[x]));
    });
}
```

---

In `elementFromChar`, per prima cosa creiamo un'istanza del tipo desiderato esaminando il costruttore del carattere e applicandovi `new`. Aggiungiamo quindi una proprietà `originChar`, in modo che si capisca sempre da quale carattere si era partiti per creare l'elemento.

La proprietà `originChar` serve quando si implementa il metodo `toString` dell'oggetto `World`. Questo metodo costruisce una stringa simile alla mappa partendo dallo stato corrente del mondo e svolgendo un ciclo bidimensionale sulle caselle della griglia.

---

```
function charFromElement(element) {
    if (element == null)
        return " ";
    else
        return element.originChar;
}

World.prototype.toString = function() {
    var output = "";
    for (var y = 0; y < this.grid.height; y++) {
        for (var x = 0; x < this.grid.width; x++) {
            var element = this.grid.get(new Vector(x, y));
            output += charFromElement(element);
        }
        output += "\n";
    }
    return output;
}
```

```
        output += charFromElement(element);
    }
    output += "\n";
}
return output;
};
```

---

L'oggetto `wall` (muro) è molto semplice: viene usato solo per occupare dello spazio e non ha un suo metodo `act`.

```
function Wall() {}
```

---

Quando proviamo l'oggetto `World`, prima creando un'istanza basata sul piano descritto in precedenza, e poi richiamando `toString`, otteniamo come risultato una stringa molto simile al piano che avevamo impostato all'inizio.

```
var world = new World(plan,
                      {"#": Wall,
                       "o": BouncingCritter});
console.log(world.toString());
// → #####
// #   #   #   o   ##
// #
// #       #####   #
// ##      #   #   ##   #
// ###      ##   #   #
// #       ###   #   #
// #   #####
// #   ##   o   #
// #   #   #   o   #####
// #   #
// #####
```

---

## this e la sua visibilità

Il costruttore `World` contiene una chiamata a `forEach`. Una cosa interessante da notare è che, all'interno della funzione passata a `forEach`, non ci troviamo più direttamente nell'area di visibilità della funzione del costruttore. Ogni chiamata di funzione riceve la sua associazione a `this`; pertanto, il `this` nella funzione interna non fa riferimento all'oggetto appena costruito, al quale fa riferimento il `this` esterno. In effetti, quando una funzione non viene chiamata come metodo, `this` fa riferimento all'oggetto globale.

Ciò significa che non possiamo scrivere `this.grid` per accedere alla griglia

dall'interno del ciclo. Invece, la funzione esterna crea una variabile locale normale, `grid`, attraverso la quale la funzione interna ottiene l'accesso alla griglia.

Questo è un piccolo problema strutturale di JavaScript. Per fortuna, la prossima versione del linguaggio offre una soluzione migliore. Nel frattempo, ci sono altre vie. Una delle più comuni è di specificare `var self = this` e poi fare riferimento a `self`, che è una variabile normale e pertanto visibile per le funzioni interne.

Un'altra soluzione è di usare il metodo `bind`, che consente di specificare un oggetto `this` da associare.

---

```
var test = {
  prop: 10,
  addPropTo: function(array) {
    return array.map(function(elt) {
      return this.prop + elt;
    }).bind(this));
  }
};

console.log(test.addPropTo([5]));
// → [15]
```

---

La funzione passata a `map` è il risultato della chiamata a `bind` e pertanto ha il suo `this` associato al primo argomento passato a `bind`, ossia il valore `this` della funzione esterna, che contiene l'oggetto `test`.

Quasi tutti i metodi di ordine superiore per gli array, come `forEach` e `map`, accettano un secondo argomento facoltativo, che serve per assegnare un `this` alle chiamate alla funzione di iterazione. Pertanto, possiamo esprimere l'esempio precedente in maniera un pochino più semplice:

---

```
var test = {
  prop: 10,
  addPropTo: function(array) {
    return array.map(function(elt) {
      return this.prop + elt;
    }, this); // ← no bind
  }
};

console.log(test.addPropTo([5]));
// → [15]
```

---

Questo vale solo per le funzioni di ordine superiore che danno supporto a un

parametro di *context*. Per quelle che non lo fanno, dovrete scegliere una delle altre soluzioni.

Nelle nostre funzioni di ordine superiore, il supporto al parametro di contesto viene dalla chiamata al metodo `call` per la funzione passata come argomento. Per esempio, il seguente è un metodo `forEach` per il nostro tipo `Grid`, che richiama una determinata funzione per ciascun elemento della griglia che non sia `null` o `undefined`:

---

```
Grid.prototype.forEach = function(f, context) {
  for (var y = 0; y < this.height; y++) {
    for (var x = 0; x < this.width; x++) {
      var value = this.space[x + y * this.width];
      if (value != null)
        f.call(context, value, new Vector(x, y));
    }
  }
};
```

---

## Animare la vita

Il passo successivo è scrivere un metodo `turn` per l'oggetto `World`, che dia alle creature la possibilità di agire. Il metodo passa sulla griglia attraverso il metodo `forEach`, che abbiamo appena descritto, in cerca di oggetti che abbiano un metodo `act`. Quando trova un oggetto adatto, ne richiama il metodo `act` per ottenere un oggetto `action` e portare a termine l'azione, se valida. Per ora, sono valide solo le azioni di spostamento (`move`).

Questa impostazione presenta tuttavia un possibile problema. Riuscite a individuarlo? Se lasciamo che le creature si spostino man mano che le incontriamo, esse potrebbero finire in una casella che non abbiamo ancora esaminato e dalla quale potranno spostarsi *di nuovo*. Pertanto, dobbiamo mantenere un array di creature che hanno già avuto il loro turno e ignorarle se le incontriamo di nuovo.

---

```
World.prototype.turn = function() {
  var acted = [];
  this.grid.forEach(function(critter, vector) {
    if (critter.act && acted.indexOf(critter) == -1) {
      acted.push(critter);
      this.letAct(critter, vector);
    }
  }, this);
};
```

---

Usiamo il secondo parametro del metodo `forEach` della griglia per accedere al giusto `this` nella funzione interna. Il metodo `letAct` contiene la logica che consente alle creature di spostarsi.

---

```
World.prototype.letAct = function(critter, vector) {  
    var action = critter.act(new View(this, vector));  
    if (action && action.type == "move") {  
        var dest = this.checkDestination(action, vector);  
        if (dest && this.grid.get(dest) == null) {  
            this.grid.set(vector, null);  
            this.grid.set(dest, critter);  
        }  
    }  
};  
  
World.prototype.checkDestination = function(action, vector) {  
    if (directions.hasOwnProperty(action.direction)) {  
        var dest = vector.plus(directions[action.direction]);  
        if (this.grid.isInside(dest))  
            return dest;  
    }  
};
```

---

Per prima cosa, ci limitiamo a chiedere alla creatura di agire, passandole un oggetto `view` che conosce il mondo e la posizione della creatura al suo interno (definiremo `View` tra poco). Il metodo `act` restituisce un'azione di qualche tipo.

Se il tipo dell'azione non è "move", il valore restituito viene ignorato. Se è "move" e se ha una proprietà `direction` che fa riferimento a una direzione valida e se la casella di destinazione è vuota (`null`), allora si imposta su `null` la casella dove si trovava la creatura prima dello spostamento e si sposta la creatura nella casella di destinazione.

Notate che `letAct` ignora i dati che non hanno senso, in quanto non dà per scontato che la proprietà `direction` dell'azione sia necessariamente valida o che il suo tipo abbia senso. Questo tipo di programmazione *difensiva* è utile in alcune situazioni. La sua applicazione principale è nel convalidare i dati inseriti da fonti su cui non si ha controllo (per esempio, dati inseriti dagli utenti o file), ma può essere utile anche per isolare tra loro dei sottosistemi. In questo caso, la nostra scelta dipende dal fatto che non vogliamo porre troppa attenzione nel programmare le creature, che così non dovranno verificare se le azioni che intendono svolgere abbiano significato. Si limitano a richiedere un'azione e ci pensa il mondo a stabilire se consentirla o no.

Questi due metodi non fanno parte dell'interfaccia esterna dell'oggetto `World`. Sono un dettaglio interno. Alcuni linguaggi consentono di dichiarare esplicitamente come

*privati* (*private*) certi metodi e proprietà, e lanciano un errore quando si cerca di usarli dall'esterno. JavaScript non offre questa possibilità e bisogna pertanto fare affidamento a qualche altra forma di comunicazione per descrivere che cosa appartiene all'interfaccia di un determinato oggetto. A volte può essere utile seguire uno schema di denominazione che distingua tra proprietà interne ed esterne, per esempio premettendo al nome di quelle interne un carattere di sottolineatura (\_). In questo modo, sarà più facile identificare gli errori nell'uso di proprietà che non fanno parte dell'interfaccia di un oggetto.

L'unica parte mancante, il tipo *view*, è definito come segue:

---

```
function View(world, vector) {
    this.world = world;
    this.vector = vector;
}

View.prototype.look = function(dir) {
    var target = this.vector.plus(directions[dir]);
    if (this.world.grid.isInside(target))
        return charFromElement(this.world.grid.get(target));
    else
        return "#";
};

View.prototype.findAll = function(ch) {
    var found = [];
    for (var dir in directions)
        if (this.look(dir) == ch)
            found.push(dir);
    return found;
};

View.prototype.find = function(ch) {
    var found = this.findAll(ch);
    if (found.length == 0) return null;
    return randomElement(found);
};
```

---

Il metodo *look* calcola le coordinate che vogliamo trovare e, se sono all'interno della griglia, trova il carattere corrispondente all'elemento che vi trova. Se le coordinate sono fuori dalla griglia, *look* immagina che ci sia un muro; in questo modo, se definiamo un mondo che non è circondato da mura, le creature non potranno comunque scavalcarne i confini.

## Si muove

In precedenza, abbiamo istanziato un oggetto `World`.

Ora che vi abbiamo aggiunto tutti i metodi necessari, dovrebbe essere possibile farlo muovere.

---

```
for (var i = 0; i < 5; i++) {  
    world.turn();  
    console.log(world.toString());  
}  
// → ... cinque giri di creature in movimento
```

---

Le prime due mappe che vengono visualizzate avranno un aspetto simile alle seguenti (a seconda della direzione casuale scelta dalle creature):

---

```
#####
#   #   #
      0   #   #
    #####   #   #
  #   #   ##   #   #
  ##   #   #   ##   #   #
  ##   ##   #   #   ##   #   #
  #   ###   #   #   #
  #   #####
  #   ##
  #   #   0   ###   #   #
  #   #   0   #   #   #   #
#0   #   0   #   #   #   0   0   #
#####
```

---

Si muovono! Per un'esperienza più interattiva con le creature che vagano qua e là e rimbalzano sui muri, aprirete il capitolo nella versione online del libro, su <http://eloquentjavascript.net/>.

## Altre forme di vita

Il momento cruciale del nostro mondo, se lo osservate abbastanza a lungo, è quando due creature si scontrano. Vi viene in mente un'altra forma di comportamento altrettanto interessante?

Quella a cui sono arrivato io è una creatura che si sposta lungo i muri. Concettualmente, la creatura tiene la sua mano sinistra (zampa, tentacolo o quel che sia) appoggiata al muro e lo segue. Questo concetto non è poi così facile da esprimere come sembra.

Questo perché dobbiamo poter “computare” usando le direzioni della bussola. Poiché le direzioni sono definite da una serie di stringhe, dobbiamo definire un’operazione

(dirPlus) che calcoli le direzioni relative. Pertanto, dirPlus("n", 1) significa un giro di 45 gradi, in senso orario, partendo da nord, il che porta a "ne". Analogamente, dirPlus("s", -2) significa un giro di 90 gradi in senso antiorario partendo da sud, che si traduce in est.

---

```
function dirPlus(dir, n) {  
    var index = directionNames.indexOf(dir);  
    return directionNames[(index + n + 8) % 8];  
}  
  
function WallFollower() {  
    this.dir = "s";  
}  
  
WallFollower.prototype.act = function(view) {  
    var start = this.dir;  
    if (view.look(dirPlus(this.dir, -3)) != " ")  
        start = this.dir = dirPlus(this.dir, -2);  
    while (view.look(this.dir) != " ") {  
        this.dir = dirPlus(this.dir, 1);  
        if (this.dir == start) break;  
    }  
    return {type: "move", direction: this.dir};  
};
```

---

Il metodo act deve semplicemente “scrutare” le caselle intorno alla creatura, partendo dalla sua sinistra e proseguendo in senso orario finché non trova una casella vuota. Si sposta quindi nella direzione della casella vuota.

Quel che complica le cose è che una creatura può finire in mezzo a dello spazio vuoto, sia come posizione di partenza, sia come risultato di uno spostamento intorno a un’altra creatura. Se applichiamo la soluzione che ho appena descritto in uno spazio vuoto, la povera creatura continua a girare a sinistra, senza mai fermarsi.

Ecco perché c’è un’ulteriore verifica (la dichiarazione `if`) che dice di cominciare a esaminare lo spazio partendo da sinistra solo se si presume che la creatura abbia appena superato un ostacolo, ossia, se lo spazio dietro e a sinistra della creatura non è vuoto. Diversamente, la creatura comincia a ispezionare lo spazio guardando direttamente davanti a sé e si sposterà in quella casella se la trova vuota.

Alla fine, trovate una prova dove, a ogni passata del ciclo, si confrontano `this.dir` e `start` per accertarsi che il ciclo non continui all’infinito quando la creatura è circondata da mura o da altre creature e non riesce a trovare una casella vuota.

## Una simulazione più simile alla vita

Per rendere più interessante la vita nel nostro mondo, aggiungeremo i concetti di cibo e riproduzione. Ciascuna delle cose viventi del mondo ottiene una nuova proprietà, `energy`, che si riduce quando si svolgono delle azioni e aumenta quando si mangia. Quando la creatura ha abbastanza energie, può riprodursi, generando una nuova creatura del suo stesso tipo. Per semplificare le cose, le creature del nostro mondo si riproducono in maniera asessuata, facendo tutto da sole.

Se le creature si spostano e si mangiano tra loro, il mondo soccomberà ben presto alla legge dell'aumento di entropia, consumerà tutta l'energia e diverrà un deserto senza vita. Per evitare che ciò accada (perlomeno troppo in fretta), aggiungiamo delle piante. Le piante non si spostano: usano la fotosintesi per crescere (ossia, aumentare la propria energia) e riprodursi.

Perché questo funzioni, ci serve un mondo con un diverso metodo `letAct`. Potremmo semplicemente sostituire il metodo del prototipo `World`, ma io mi sono affezionato alla simulazione con le creature che seguono il muro e mi spiace distruggere quel vecchio mondo.

Una soluzione prevede l'ereditarietà. Impostiamo un nuovo costruttore, `Lifelikeworld`, il cui prototipo si basa su quello di `World`, ma prevale sul suo metodo `letAct`. Il nuovo metodo `letAct` delega il lavoro di svolgere un'azione a diverse funzioni, memorizzate nell'oggetto `actionTypes`.

---

```
function Lifelikeworld(map, legend) {
    World.call(this, map, legend);
}

Lifelikeworld.prototype = Object.create(World.prototype);
var actionTypes = Object.create(null);

Lifelikeworld.prototype.letAct = function(critter, vector) {
    var action = critter.act(new View(this, vector));
    var handled = action &&
        action.type in actionTypes &&
        actionTypes[action.type].call(this, critter,
            vector, action);

    if (!handled) {
        critter.energy -= 0.2;
        if (critter.energy <= 0)
            this.grid.set(vector, null);
    }
};
```

---

Il nuovo metodo `letAct` prima verifica se è stata restituita un'azione, poi, se esiste una funzione di gestione (*handler*) per questo tipo di azione e, alla fine, se la funzione di gestione ha restituito `true`, confermando così di aver gestito correttamente l'azione. Notate l'uso di `call` per dare alla funzione di gestione accesso al mondo attraverso l'associazione al suo `this`.

Se per qualche ragione l'azione non ha funzionato, il comportamento predefinito è che la creatura si limita ad aspettare. Perde un quinto di punto di energia e, se il suo livello di energia scende a zero o meno, la creatura muore e viene rimossa dalla griglia.

## Metodi di gestione delle azioni

L'azione più semplice che una creatura può svolgere è crescere (`grow`) e viene usata dalle piante. Quando viene restituito un oggetto azione come `{type: "grow"}`, sarà richiamato il seguente metodo di gestione:

---

```
actionTypes.grow = function(critter) {
    critter.energy += 0.5;
    return true;
};
```

---

L'azione `grow` (crescere) ha sempre successo e aggiunge mezzo punto al livello di energia della pianta.

L'azione `move` (spostarsi) è più complessa.

---

```
actionTypes.move = function(critter, vector, action) {
    var dest = this.checkDestination(action, vector);
    if (dest == null ||
        critter.energy <= 1 ||
        this.grid.get(dest) != null)
        return false;
    critter.energy -= 1;
    this.grid.set(vector, null);
    this.grid.set(dest, critter);
    return true;
};
```

---

Per prima cosa, questa azione verifica, attraverso il metodo `checkDestination` definito in precedenza, se il risultato è una destinazione valida. Se la destinazione non è valida o non è vuota, o se la creatura non ha sufficiente energia, `move` restituisce `false` per indicare che non è stata svolta alcuna azione. Diversamente, sposta la creatura e sottrae il costo energetico.

Oltre a muoversi, le creature possono mangiare (eat).

---

```
actionTypes.eat = function(critter, vector, action) {  
    var dest = this.checkDestination(action, vector);  
    var atDest = dest != null && this.grid.get(dest);  
    if (!atDest || atDest.energy == null)  
        return false;  
    critter.energy += atDest.energy;  
    this.grid.set(dest, null);  
    return true;  
};
```

---

Mangiare un'altra creatura comporta anche il recupero di una casella di destinazione valida. Questa volta, la destinazione non deve essere vuota e deve contenere qualcosa che abbia energia, come una creatura (ma non un muro, i muri non sono commestibili). In questo caso, l'energia della creatura mangiata viene trasferita a chi la mangia e la vittima viene eliminata dalla griglia.

Infine, lasciamo che le nostre creature si riproducano.

---

```
actionTypes.reproduce = function(critter, vector, action) {  
    var baby = elementFromChar(this.legend,  
                                critter.originChar);  
    var dest = this.checkDestination(action, vector);  
    if (dest == null ||  
        critter.energy <= 2 * baby.energy ||  
        this.grid.get(dest) != null)  
        return false;  
    critter.energy -= 2 * baby.energy;  
    this.grid.set(dest, baby);  
    return true;  
};
```

---

Riprodursi costa due volte il livello di energia della neonata creatura. Pertanto, per prima cosa creiamo un ipotetico piccolo usando `elementFromChar` sul carattere della creatura di origine. Una volta ottenuto il piccolo, troviamo il suo livello di energia e verifichiamo se il genitore ne ha a sufficienza per portarlo nel mondo. Ci serve anche una destinazione valida e vuota.

Se tutto funziona, il piccolo viene inserito nella griglia (non è più ipotetico) e l'energia si trasferisce da genitore a neonato.

## Popolare il nuovo mondo

Abbiamo ora una struttura per simulare queste creature più simili alla vita vera. Potremmo importare le creature del vecchio mondo, ma poiché quelle non dispongono di una proprietà per l'energia, arriverebbero già morte. Pertanto, ne creiamo di nuove. Per prima cosa impostiamo una pianta, che è una forma di vita abbastanza semplice.

---

```
function Plant() {  
    this.energy = 3 + Math.random() * 4;  
}  
  
Plant.prototype.act = function(context) {  
    if (this.energy > 15) {  
        var space = context.find(" ");  
        if (space)  
            return {type: "reproduce", direction: space};  
    }  
    if (this.energy < 20)  
        return {type: "grow"};  
};
```

---

Le piante partono con un livello di energia compreso tra 3 e 7, a caso, in modo che non si riproducano tutte insieme. Quando una pianta arriva a 15 punti di energia e ha dello spazio vuoto vicino, può riprodursi in quello spazio. Se non può riprodursi, una pianta cresce finché raggiunge un livello di energia di 20 punti.

Definiamo ora un mangiatore di piante (PlantEater).

---

```
function PlantEater() {  
    this.energy = 20;  
}  
  
PlantEater.prototype.act = function(context) {  
    var space = context.find(" ");  
    if (this.energy > 60 && space)  
        return {type: "reproduce", direction: space};  
    var plant = context.find("*");  
    if (plant)  
        return {type: "eat", direction: plant};  
    if (space)  
        return {type: "move", direction: space};  
};
```

---

Per le piante, useremo il carattere \*, ed è questo che la creatura andrà a cercare quando vuole del cibo.

## Far vivere il mondo

Con questo, abbiamo abbastanza elementi per provare il nostro nuovo mondo. Immaginate la mappa che segue come una verde vallata, dove ci sono un branco di erbivori, qualche masso sparso e vegetazione lussureggiante dappertutto.

```
var valley = new LifelikeWorld(  
    ["#####",  
     "####",  
     "## ***",  
     "# *###*",  
     "# *** 0",  
     "# ***",  
     "# 0",  
     "#*",  
     "#**",  
     "#***",  
     "#****",  
     "#*****",  
     "#*****"],  
    {"#": Wall,  
     "0": PlantEater,  
     "*": Plant}  
)
```

Vediamo che cosa succede quando eseguiamo il programma.

---

```
#####
# #####
## *** 0   *## #### # ** 0   ##
# *## *   ** *## # **##   ## #
# ** ### *## *# # ** 0   ##0   #
#      ### *# # *0   * * ##   #
#      ## 0   # #   *** ## 0   #
#      #* 0   # #** #***   #
#*      #** 0   # #** 0   #****   #
#* 0 0 #** *## #*** #*** #**** 0   #
##*      ###* ### ###* ###* 0   ###
#####
```

---

```
#####
# #####0 0   ##### #### 0   #####
##      ## ## ##   ## #
# ##0   ## # ##   0   ## #
# 0 0 0 0 *## # #   ## #
#      ***## 0 # #   ## #
#      ***## 0 # #   0 ## *   #
#      # *** * # #   # 0   #
#      # 0***** 0 # #   0 # 0   #
#      #***** # #   ## 0 0   #
##      #***** # #   ## 0   ###
#####
```

---

```
#####
# #####
##      ## ## ##   ## #
# ## # ## ## #   ** * ## #
# ## # ## ## #   ***** ## #
#      ## # #   ##### #
#      ## * #   ##### #
#      0 ## * # #   ##### #
#      # # #   # #   # ** ** ## #
#      # # #   # #   #   #
#      # # #   # #   # #   #
##      # # #   # #   # #   ###
#####
```

---

Il più delle volte, le piante si moltiplicano e si espandono con una certa velocità; ma a un certo punto l'abbondanza di cibo fa esplodere la popolazione degli erbivori, che spazzano via quasi tutte le piante e provocano una carestia di massa per tutte le creature. A volte, l'ecosistema si riprende e inizia un nuovo ciclo. Altre volte, una delle specie si estingue. Se si tratta degli erbivori, lo spazio si riempie di piante. Se si tratta delle piante, le creature che restano muoiono di fame e la valle diventa un deserto di desolazione. Ah, la crudeltà della natura!

# Esercizi

## *Stupidità artificiale*

Che gli abitanti del nostro mondo si estinguano dopo pochi minuti è abbastanza deprimente. Per evitarlo, potremmo provare a creare un mangiatore di piante più intelligente.

I nostri erbivori presentano diversi problemi ovvi. Primo, sono molto voraci: si rimpinzano di piante finché non hanno eliminato tutta la vita vegetale. Secondo, i loro spostamenti a caso (ricordate che il metodo `view. find` restituisce una direzione a caso quando trova più direzioni valide) li fanno andare di qua e di là senza meta, col risultato che muoiono di fame se non trovano qualche pianta. Infine, si riproducono a grande velocità, il che intensifica i cicli tra abbondanza e carestia.

Definite un nuovo tipo di creatura che risolva uno o più dei problemi esposti e sostituitelo al vecchio tipo `PlantEater` della vallata. Verificatene il funzionamento e modificate lo se necessario.

## *Predatori*

Tutti gli ecosistemi che si rispettano hanno una catena alimentare più lunga di un solo anello. Impostate un'altra creatura che si cibi di erbivori. Noterete che la stabilità è ancora più difficile da ottenere, adesso che ci sono cicli a più livelli. Cercate di elaborare una strategia che mantenga l'ecosistema in equilibrio almeno per un po' di tempo.

Per questo, può esservi utile ampliare il mondo. In questo modo, le oscillazioni delle popolazioni locali avranno meno possibilità di far scomparire completamente una specie e ci sarà spazio per una popolazione di prede abbastanza numerosa da poter mantenere una piccola popolazione di predatori.

## BACHI E GESTIONE DEGLI ERRORI

*Il debugging è due volte più difficile dello scrivere il codice. Pertanto, se scrivete codice sfruttando al massimo le vostre capacità, per definizione non siete abbastanza intelligenti per fare il debugging.*

Brian Kernighan e P.J. Plauger, *The Elements of Programming Style*

I programmi sono pensieri cristallizzati. A volte sono pensieri confusi. Altre volte, ci scappano degli errori quando convertiamo i pensieri in codice. Nell'uno e nell'altro caso, il risultato sono programmi difettosi.

I difetti dei programmi si chiamano bachi. I bachi possono essere errori di programmazione o problemi insiti nei sistemi con i quali il programma interagisce. Alcuni bachi sono immediatamente rilevabili, mentre altri sono più sottili e rimangono nascosti per anni nel sistema.

Spesso i problemi vengono a galla quando il programma incontra una situazione che il programmatore non aveva previsto. In alcuni casi, situazioni di questo tipo sono inevitabili: quando si chiede di inserire l'età e un utente digita *arancione*, il programma si trova in una posizione difficile. In qualche modo, bisogna prevedere e gestire queste situazioni.

### Errori di programmazione

Quando si tratta di errori di programmazione, la soluzione è semplice. Basta trovarli e sistemarli. Questi errori vanno da semplici errori di scrittura, ai quali il computer reagisce protestando appena li incontra, a sottigliezze nell'interpretare il comportamento del programma, che danno risultati sbagliati solo in determinate circostanze. Diagnosticare bachi di questo secondo tipo può richiedere settimane.

L'aiuto offerto dal linguaggio nella ricerca di errori di questo tipo varia da linguaggio a linguaggio. Come ci si può aspettare, JavaScript si colloca all'estremità meno collaborativa della scala. Alcuni linguaggi richiedono che il programmatore specifichi i tipi di tutte le variabili e di tutte le espressioni ancor prima di poter eseguire il programma, e segnalano immediatamente quando cerchiamo di usare un tipo in maniera non ammessa. JavaScript prende in considerazione i tipi solo in fase di esecuzione e, anche allora, ci permette di fare cose decisamente insensate, come `x = true * "scimmia"` senza protestare.

Ci sono però delle cose che JavaScript rifiuta di accettare. Scrivere un programma che

non sia sintatticamente valido fa scattare immediatamente un errore. Altre cose, come richiamare qualcosa che non sia una funzione o recuperare una proprietà per un valore indefinito (`undefined`), provocano errori durante l'esecuzione, quando il programma incontra l'azione sbagliata.

Spesso, però, i vostri calcoli senza senso si limitano a produrre valori `NAN` (non un numero) o `undefined`. E il programma continua senza lamentarsi, convinto di svolgere calcoli interessanti. L'errore si manifestera solo in un secondo tempo, dopo che il valore errato ha attraversato una serie di funzioni. Potrebbe persino non provocare errori, ma semplicemente far produrre al programma un risultato sbagliato. Trovare l'origine di questi errori può essere molto difficile.

Il processo di trovare gli errori, i bachi, nei programmi si chiama *debugging* o eliminazione degli errori.

## Modalità strict

Si può imporre un minimo di rigore a JavaScript attivando la modalità strict. Ciò si fa inserendo la stringa "use strict" all'inizio del file o del corpo di una funzione. Ecco un esempio:

---

```
function canYouSpotTheProblem() {  
    "use strict";  
    for (counter = 0; counter < 10; counter++)  
        console.log("Happy happy");  
}  
canYouSpotTheProblem();  
// → ReferenceError: counter is not defined
```

---

Di solito, se dimenticate di specificare `var` davanti a una variabile, come con `counter` in questo errore, JavaScript crea silenziosamente una variabile globale e usa quella. In modalità strict, viene invece segnalato un errore, cosa che è molto utile. Bisogna tuttavia notare che questa soluzione non funziona quando la variabile in questione esiste già come variabile globale, ma solo quando la si sarebbe creata assegnando un valore.

Un altro comportamento specifico della modalità strict è che l'associazione con `this` ha il valore `undefined` nelle funzioni che non vengono richiamate come metodi. Quando si effettuano queste chiamate al di fuori della modalità strict, `this` fa riferimento all'oggetto con visibilità globale. Pertanto, se per sbaglio richiamate un metodo o un costruttore in maniera non corretta in modalità strict, JavaScript lancia un errore appena cerca di leggere qualcosa da `this`, invece di continuare a lavorare con l'oggetto globale, creando e leggendo variabili globali.

Si consideri per esempio il codice seguente, che richiama un costruttore senza la parola chiave `new` e pertanto il suo `this` non farà riferimento a un oggetto appena costruito

con new:

---

```
function Person(name) { this.name = name; }
var ferdinand = Person("Ferdinand"); // oops
console.log(name);
// → Ferdinand
```

---

Qui, la chiamata (sbagliata) a Person ha avuto esito, ma ha restituito un valore undefined e creato la variabile globale name. In modalità strict, il risultato è diverso:

```
"use strict";
function Person(name) { this.name = name; }
// Oops, forgot 'new'
var ferdinand = Person("Ferdinand");
// → TypeError: Cannot set property 'name' of undefined
```

---

In questo caso, veniamo immediatamente avvertiti che qualcosa è sbagliato. Il che è molto utile.

La modalità strict offre alcuni altri servizi. Impedisce di assegnare a una funzione più parametri con lo stesso nome ed elimina completamente alcune funzionalità problematiche del linguaggio (come per esempio l'istruzione with, che è talmente male impostata che non ne riparla da nessun'altra parte del libro).

In breve, inserire la stringa "use strict" all'inizio del programma difficilmente guasta e può aiutarvi a scoprire dei problemi.

## Prove di esecuzione

Visto che il linguaggio non aiuta un granché a trovare gli errori, dovremo trovarli noi, eseguendo il programma e valutando se fa la cosa giusta o no.

Farlo a mano, più e più volte, è un ottimo modo per impazzire. Per fortuna, è spesso possibile scrivere un secondo programma che renda automatiche le prove.

Come esempio, riprendiamo il tipo Vector.

---

```
function Vector(x, y) {
  this.x = x;
  this.y = y;
}
Vector.prototype.plus = function(other) {
  return new Vector(this.x + other.x, this.y + other.y);
};
```

---

Scriveremo un programma che verifichi che l'impostazione di `vector` dia i risultati desiderati. Dopodiché, ogni volta che cambiamo l'impostazione, ripetiamo l'esecuzione del programma di prova, in modo da essere ragionevolmente sicuri di non aver combinato guai. Quando aggiungiamo nuove funzionalità (per esempio, un nuovo metodo) al tipo `vector`, inseriamo addirittura i test relativi:

---

```
function testVector() {  
    var p1 = new Vector(10, 20);  
    var p2 = new Vector(-10, 5);  
    var p3 = p1.plus(p2);  
    if (p1.x !== 10) return "fail: x property";  
    if (p1.y !== 20) return "fail: y property";  
    if (p2.x !== -10) return "fail: negative x property";  
    if (p3.x !== 0) return "fail: x from plus";  
    if (p3.y !== 25) return "fail: y from plus";  
    return "everything ok";  
}  
  
console.log(testVector());  
// → everything ok
```

---

Scrivere prove come la precedente tende però a produrre codice ripetitivo e poco elegante. Per fortuna, esistono strumenti di software che aiutano a realizzare ed eseguire serie di test attraverso un linguaggio (sotto forma di funzioni e metodi) adatto per esprimere le prove e fornire risultati ricchi di informazioni quando una prova non riesce. Questi strumenti si chiamano *testing frameworks* (strutture di prova).

## Debugging

Quando trovate qualcosa che non funziona nel programma, qualcosa che non si comporta come dovrebbe o produce errori, allora dovete scoprire dove sta il problema.

A volte la causa è ovvia: se il messaggio di errore punta a una riga del programma, leggendo la descrizione dell'errore e la riga incriminata dovrete già vedere di che cosa si tratta.

Non è sempre così, però. A volte, la riga che fa scattare il problema è il primo posto dove un valore fasullo, prodotto da qualche altra parte, viene usato in maniera non valida. E a volte non ci sono messaggi di errore: solo un risultato non valido. Se avete completato gli esercizi dei capitoli precedenti, probabilmente vi siete già trovati in situazioni come queste.

Nel programma di esempio che segue, cerchiamo di convertire un numero intero in una stringa, in qualunque base (decimale, binario e così via), recuperando ogni volta

l'ultima cifra e dividendo il numero per eliminarla. Il risultato pazzesco prodotto indica che ci dev'essere un baco.

---

```
function numberToString(n, base) {  
    var result = "", sign = "";  
    if (n < 0) {  
        sign = "-";  
        n = -n;  
    }  
    do {  
        result = String(n % base) + result;  
        n /= base;  
    } while (n > 0);  
    return sign + result;  
}  
  
console.log(numberToString(13, 10));  
// → 1.5e-3231.3e-3221.3e-3211.3e-3201.3e-3191.3e-3181.3...
```

---

Anche se avete già visto dove sta il problema, provate un attimo a far finta di non saperlo. Sappiamo che il programma non funziona e vogliamo scoprire il perché.

Intanto, dovete resistere alla tentazione di cominciare a cambiare cose a caso nel codice. Invece, pensate. Analizzate che cosa succede e trovate una teoria che ne spieghi il perché. Fate, quindi, altre osservazioni per mettere alla prova la teoria oppure, se non avete ancora messo insieme una teoria, fate altre osservazioni che vi aiutino a formularne una.

Inserire in punti strategici del programma qualche chiamata a `console.log` è un buon modo per ottenere informazioni su quel che il programma sta facendo. In questo caso, vogliamo che `n` prenda i valori 13, 1 e 0. Scriviamo il suo valore all'inizio del ciclo.

---

```
13  
1.3  
0.13  
0.013  
...  
1.5e-323
```

---

Bene. Dividere 13 per 10 non dà un numero intero. Invece di `n /= base`, quel che ci serve davvero è `n = Math.floor(n / base)`, in modo che il numero venga correttamente “spostato” a destra.

Un'alternativa a `console.log` è usare le funzionalità del *debugger* del browser. I

browser moderni offrono tutti la capacità di impostare un *punto di interruzione* (breakpoint) in una data riga del codice. In questo modo, l'esecuzione del programma va in pausa ogni volta che si arriva alla riga col punto di interruzione, cosa che vi permette di esaminare i valori delle variabili a quel punto. Non entro nei particolari qui, perché i debugger variano da browser a browser; vi basta comunque aprire gli strumenti di sviluppo del vostro browser e fare qualche ricerca sul Web. Un altro modo per impostare un punto di interruzione è di inserire una dichiarazione debugger (che consiste solo di quella parola chiave) nel programma. Se gli strumenti di sviluppo sono attivi, il programma va in pausa ogni volta che raggiunge quella dichiarazione, cosa che vi permette di verificarne lo stato.

## Propagazione degli errori

Purtroppo, chi programma non ha il potere di evitare tutti i problemi. Se il programma comunica col mondo esterno, non importa come, c'è la probabilità che riceva dati di input non validi o che altri sistemi coi quali tenta di comunicare siano difettosi o non raggiungibili.

I programmi semplici, e quelli eseguiti solo sotto la propria supervisione, possono semplicemente fermarsi in questi casi. Potrete andare a vedere dove sta il problema e riprovare. D'altro canto, le applicazioni "reali" devono essere fatte in modo da non bloccarsi. A volte la cosa giusta da fare è accettare i dati di input non validi e continuare comunque. In altri casi, è meglio far sapere all'utente che cosa non ha funzionato prima di interrompere il programma. In entrambi i casi, però, il programma deve fare qualcosa di attivo in risposta al problema.

Immaginate di avere una funzione `promptInteger` che chiede all'utente di inserire un numero intero e poi lo restituisce. Che cosa va restituito se l'utente digita *arancione*?

Una possibilità è di restituire un valore speciale. Le scelte più comuni per questi valori sono `null` e `undefined`.

---

```
function promptNumber(question) {  
  var result = Number(prompt(question, ""));  
  if (isNaN(result)) return null;  
  else return result;  
}  
console.log(promptNumber("How many trees do you see?"));
```

---

Questa è una strategia valida. Ora, il codice che chiama `promptNumber` deve per prima cosa verificare se si è rilevato un numero; diversamente, il programma deve continuare, magari ripetendo la domanda o inserendo un valore predefinito.

In molte situazioni, soprattutto quando gli errori sono comuni e la funzione di chiamata dovrebbe prevederli esplicitamente, restituire un valore speciale è un ottimo

modo per indicare un errore. Ci sono però degli svantaggi. Intanto, che cosa succede se la funzione può già restituire qualunque tipo di valore? Per queste funzioni, è difficile trovare un valore speciale che si distingua da un risultato valido.

Il secondo problema col restituire valori speciali è che il codice può diventare davvero complicato. Se in un brano di codice si richiama `prompt`-`Number` 10 volte, bisogna verificare 10 volte se è stato restituito un valore `null`. E se la risposta è a sua volta `null`, la funzione chiamante dovrà controllare anche quella e così via.

## Eccezioni

Quando una funzione non può continuare normalmente, quel che vorremmo è semplicemente interrompere quel che stiamo facendo e tornare subito indietro a una posizione che sa come risolvere il problema. Di questo si occupa la *gestione delle eccezioni*.

Le eccezioni sono un meccanismo che consente al codice che incontra un problema di *sollevarlo* (o *lanciare*) un'eccezione, che non è altro che un valore. Sollevare un'eccezione qualche volta ricorda il valore sovraccarico restituito da una funzione: esce non solo dalla funzione corrente, ma anche dalle altre funzioni chiamanti, fino alla prima chiamata che ha avviato l'esecuzione corrente. Questo modo di procedere si chiama *stack unwinding* (srotolamento della pila). Vi ricordate sicuramente la pila delle chiamate, discussa nel [Capitolo 3](#). Un'eccezione percorre tutto la pila, buttando via tutti i contesti di chiamata che incontra.

Se le eccezioni saltassero direttamente al fondo della pila non sarebbero di grande utilità. Sarebbero solo un bel modo per mandare all'aria il programma. La loro forza sta nel fatto che si possono impostare degli "ostacoli" lungo la pila, che *catturano* l'eccezione quando passa. A questo punto potete far qualcosa con l'eccezione, dopodiché il programma riprende nel punto dove avevate catturato l'eccezione.

Ecco un esempio:

---

```
function promptDirection(question) {
  var result = prompt(question, "");
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new Error("Invalid direction: " + result);
}

function look() {
  if (promptDirection("Which way?") == "L")
    return "a house";
  else
    return "two angry bears";
```

```
}

try {
    console.log("You see", look());
} catch (error) {
    console.log("Something went wrong: " + error);
}
```

---

Per sollevare un'eccezione, si usa la parola chiave `throw`. Per catturare l'eccezione, si avvolge un brano di codice in un blocco `try` seguito dalla parola chiave `catch`. Quando il codice nel blocco `try` fa sollevare un'eccezione, si calcola il blocco `catch`. Il nome della variabile (tra parentesi) dopo `catch` viene associato al valore dell'eccezione. Una volta che ha finito il blocco `catch`, o se il blocco `try` termina senza problemi, il controllo continua al di sotto dell'intera dichiarazione `try/catch`.

In questo caso, abbiamo usato il costruttore `Error` per creare il valore d'eccezione. Si tratta di un costruttore standard di JavaScript che crea un oggetto con una proprietà `message`. Negli ambienti JavaScript moderni, le istanze di questo costruttore raccolgono informazioni anche sulla pila delle chiamate che esisteva quando si è creata l'eccezione, la cosiddetta *traccia dello stack*. Queste informazioni sono memorizzate nella proprietà `stack` e possono venire in aiuto quando si effettua il debugging: indicano, infatti, la funzione dove si è verificato l'errore e tutte le altre funzioni che conducevano alla chiamata che non ha avuto esito.

Si noti che la funzione `look` ignora completamente la possibilità che `promptDirection` possa fallire. Questo è il grande vantaggio delle eccezioni: il codice di gestione degli errori è necessario solo nei punti dove si verifica l'errore e dove lo si gestisce. Le funzioni tra questi due punti possono ignorare tutto.

O perlomeno, quasi...

## Ripulire dopo le eccezioni

Considerate questa situazione: una funzione, `withContext`, deve accertarsi che, durante la sua esecuzione, la variabile di livello superiore `context` mantenga un valore di contesto specifico. Al termine dell'esecuzione, riporta la variabile al suo valore precedente.

```
var context = null;

function withContext(newContext, body) {
    var oldContext = context;
    context = newContext;
    var result = body();
    context = oldContext;
    return result;
```

```
}
```

---

Che cosa succede se body solleva un'eccezione? In quel caso, la chiamata a `withContext` viene eliminata dallo stack dall'eccezione e `context` non potrà mai essere riportata al suo vecchio valore.

Ecco un'altra caratteristica delle istruzioni `try`: possono essere seguite da un blocco `finally`, invece o in aggiunta a un blocco `catch`. Un blocco `finally` significa “Qualunque cosa accada, esegui questo codice dopo aver cercato di eseguire il codice del blocco `try`”. Se una funzione ha il compito di far pulizia, il suo codice va di solito inserito in un blocco `finally`.

---

```
function withContext(newContext, body) {  
  var oldContext = context;  
  context = newContext;  
  try {  
    return body();  
  } finally {  
    context = oldContext;  
  }  
}
```

---

Si noti che non dobbiamo più memorizzare il risultato di `body` (che è il valore da restituire) in una variabile. Il blocco `finally` viene eseguito anche se il valore di restituzione arriva direttamente dal blocco `try`. Adesso possiamo continuare senza paura:

```
try {  
  withContext(5, function() {  
    if (context < 10)  
      throw new Error("Not enough context!");  
  });  
} catch (e) {  
  console.log("Ignoring: " + e);  
}  
// → Ignoring: Error: Not enough context!  
console.log(context);  
// → null
```

---

Anche se fallisse la funzione che ha richiamato, `withContex` avrà comunque ripulito la variabile `context`.

## Catch selettivo

Quando un’eccezione arriva in fondo allo stack senza essere catturata, viene gestita dall’ambiente. Che cosa ciò significhi dipende dall’ambiente. Nei browser, di solito viene scritta una descrizione dell’errore sulla console di JavaScript (che si apre dal menu Strumenti o Sviluppo del browser).

Quando si tratta di errori di programmazione o di problemi che il programma non è in grado di gestire, spesso la cosa migliore è lasciar correre. Un’eccezione non gestita è un modo accettabile di segnalare che il programma non funziona e la console di JavaScript, nei browser moderni, offrirà qualche informazione sulle chiamate che si trovavano nello stack quando si è verificato il problema.

Per problemi che ci si aspetta possano succedere di norma, che il programma si interrompa con un’eccezione non gestita non è tuttavia una bella soluzione.

Anche gli usi non validi del linguaggio, come i riferimenti a variabili che non esistono, la lettura di proprietà `null` o chiamate a cose diverse dalle funzioni, fanno sollevare eccezioni. Queste eccezioni possono essere catturate come tutte le altre.

Quando si entra nel corpo di un blocco `catch`, sappiamo solo che qualcosa nel corpo di `try` ha fatto scattare un’eccezione. Non sappiamo, però, che cosa l’ha provocata e nemmeno che eccezione sia.

JavaScript (abbastanza clamorosamente) non offre supporto diretto per catturare selettivamente le eccezioni: o si catturano tutte o non se ne cattura nessuna. Il che ci fa presupporre che l’eccezione sollevata sia proprio quella a cui pensavamo quando abbiamo scritto il blocco `catch`. Ma non è detto! Potrebbe trattarsi di qualcos’altro o possiamo aver introdotto un baco da qualche altra parte. Ecco un esempio, dove proviamo a richiamare ripetutamente `promptDirection` finché non otteniamo una risposta valida:

---

```
for (;;) {
  try {
    var dir = promptDirection("Where?"); // ← typo!
    console.log("You chose ", dir);
    break;
  } catch (e) {
    console.log("Not a valid direction. Try again.");
  }
}
```

---

Il costrutto `for (;;)` offre un modo per creare deliberatamente un ciclo che non termina da solo. Usciamo dal ciclo solo quando viene fornita una direzione valida. Poiché abbiamo sbagliato a scrivere `promptDirection`, otterremo un errore di “`undefined variable`” (variabile non definita). E poiché il blocco `catch` ignora completamente il valore dell’eccezione (`e`), anche ammesso che sappia quale sia il problema, tratta l’errore come

se fosse sbagliato l'input (il che non è vero!). Oltre a provocare un ciclo infinito, ciò “seppellisce” il messaggio di errore che ci indicherebbe la variabile scritta male.

Come regola generale, non cercate di catturare tutte le eccezioni, se non per reindirizzarle da qualche parte; per esempio sulla rete, per indicare a un altro sistema che il vostro programma si è bloccato. Anche in quei casi, pensate attentamente alla possibilità di nascondere informazioni utili.

Tornando al problema iniziale, vogliamo catturare un tipo specifico di eccezione. Per questo, possiamo andare a vedere nel blocco `catch` se l'eccezione che abbiamo ricevuto è quella che ci interessa e, in caso contrario, la rilanciamo. Ma come possiamo riconoscere un'eccezione?

Naturalmente, possiamo far corrispondere la sua proprietà `message` al messaggio di errore che presumiamo possa verificarsi. Questo, però, non è un bel modo di scrivere codice, in quanto useremmo delle informazioni utili per gli umani (il messaggio) per prendere una decisione in sede programmatica. Non appena qualcuno modifica o traduce il messaggio, il codice smette di funzionare.

Meglio pertanto definire un nuovo tipo di errore e usare `instanceof` per identificarlo.

---

```
function InputError(message) {
  this.message = message;
  this.stack = (new Error()).stack;
}
InputError.prototype = Object.create(Error.prototype);
InputError.prototype.name = "InputError";
```

---

Facciamo derivare il prototipo da `Error.prototype` in modo che `instanceof Error` restituisca `true` anche per gli oggetti `InputError`. Gli assegniamo anche una proprietà `name`, visto che i tipi di errore standard (`Error`, `SyntaxError`, `ReferenceError` e così via) hanno tutti questa proprietà.

L'assegnazione alla proprietà `stack` tenta di dare all'oggetto una traccia di stack utile, sulle piattaforme che lo consentono, creando un normale oggetto errore e quindi usando la proprietà `stack` di quell'oggetto come se fosse la sua.

A questo punto, `promptDirection` può lanciare l'errore:

---

```
function promptDirection(question) {
  var result = prompt(question, "");
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new InputError("Invalid direction: " + result);
}
```

---

E il ciclo lo può catturare con più precisione:

---

```
for (;;) {
  try {
    var dir = promptDirection("Where?");
    console.log("You chose ", dir);
    break;
  } catch (e) {
    if (e instanceof InputError)
      console.log("Not a valid direction. Try again.");
    else
      throw e;
  }
}
```

---

Con questo, catturiamo solo le istanze di `InputError` e lasciamo passare le altre eccezioni. Se dovete ripetere l'errore di digitazione, sarà segnalato correttamente l'errore di variabile non definita.

## Asserzioni

Le *asserzioni* sono uno strumento per svolgere dei controlli elementari sugli errori di programmazione. Esaminate questa funzione ausiliaria, `assert`:

---

```
function AssertionFailed(message) {
  this.message = message;
}

AssertionFailed.prototype = Object.create(Error.prototype);

function assert(test, message) {
  if (!test)
    throw new AssertionFailed(message);
}

function lastElement(array) {
  assert(array.length > 0, "empty array in lastElement");
  return array[array.length - 1];
}
```

---

È questo un modo compatto per forzare le aspettative, sperando di far saltare il programma se la condizione definita non tiene. Per esempio, la funzione `lastElement`, che recupera l'ultimo elemento di un array, restituirebbe `undefined` sugli array vuoti se

non ci fosse l'asserzione. Recuperare l'ultimo elemento da un array vuoto non ha senso e indica pertanto quasi sicuramente un errore di programmazione.

Le asserzioni sono un modo per accertarsi che le svolte di programmazione provochino degli errori proprio nel punto dove si verificano, invece di produrre silenziosamente valori senza senso che possono dare problemi in altre parti del sistema.

## Riepilogo

Gli errori e i dati di input non validi sono inevitabili. I bachi dei programmi vanno trovati e sistemati. I bachi sono in genere più facili da notare se si hanno strumenti automatizzati di prova e se ai programmi aggiungiamo delle asserzioni.

I problemi causati da fattori esterni al controllo del programma andrebbero gestiti elegantemente. A volte, quando il problema può essere risolto localmente, dei valori di restituzione speciali possono essere un buon modo per identificarli. Altrimenti, sono meglio le eccezioni.

Lanciare un'eccezione fa svolgere lo stack delle chiamate fino al blocco try/catch successivo e fino al fondo dello stack. Il valore dell'eccezione sarà passato al blocco catch che la cattura, nel quale si verifica se si tratta proprio dell'eccezione che si cercava e quindi la si usa per fare qualcosa. Per gestire l'imprevedibilità del controllo di flusso causato dalle eccezioni, si possono usare blocchi finally per assicurarsi che almeno un brano di codice sia eseguito sempre quando termina il blocco.

## Esercizi

### Riprova

Diciamo di avere una funzione primitiveMultiply che, nel 50 percento dei casi, solleva un'eccezione di tipo MultiplicatorUnitFailure. Scrivete una funzione che la avvolga e continui a provare finché una chiamata ha successo, dopodiché ne restituisce il risultato.

Accertatevi di gestire solo le eccezioni desiderate.

### La scatola chiusa

Considerate l'oggetto seguente (piuttosto innaturale):

---

```
var box = {
    locked: true,
    unlock: function() { this.locked = false; },
    lock: function() { this.locked = true; },
    _content: [],
    get content() {
```

```
if (this.locked) throw new Error("Locked!");
return this._content;
}
};
```

---

È una scatola con un lucchetto. Al suo interno c'è un array, ma ci potete arrivare solo quando la scatola è aperta. Non potete accedere direttamente alla proprietà `_content`.

Scrivete una funzione, chiamata `withBoxUnlocked`, che accetta un valore di funzione come argomento, apre la scatola, esegue la funzione e quindi verifica che la scatola sia richiusa prima di restituire. E questo indipendentemente dal fatto che la funzione argomento abbia restituito un valore normale o abbia lanciato un'eccezione.

## ESPRESSIONI REGOLARI

*C'è gente che, di fronte a un problema, pensa 'Ma certo, userò le espressioni regolari'. E si trova di colpo con due problemi.*

Jamie Zawinski

Gli strumenti e le tecniche di programmazione sopravvivono e si diffondono in modo caotico, evoluzionario. Non sono i più eleganti o i più intelligenti che vincono, ma quelli che funzionano abbastanza bene nella nicchia giusta; per esempio perché sono integrati con altre tecnologie di successo.

In questo capitolo, parlerò proprio di uno di questi strumenti: le *espressioni regolari*. Le espressioni regolari sono un modo per descrivere le sequenze nei dati stringa. Costituiscono un piccolo linguaggio a sé, che fa parte di JavaScript e di altri linguaggi e strumenti.

Le espressioni regolari sono allo stesso tempo intricate e utilissime. La loro sintassi è misteriosa e l'interfaccia che offre loro JavaScript poco elegante. Sono, comunque, uno strumento molto potente per esaminare ed elaborare le stringhe. Capire come funzionano le espressioni regolari vi farà diventare programmatore migliori.

### Impostare un'espressione regolare

Un'espressione regolare è un tipo di oggetto, che può essere costruito col costruttore `RegExp` o espresso come valore letterale racchiudendone la struttura tra barre (/).

---

```
var re1 = new RegExp("abc");
var re2 = /abc/;
```

---

Entrambi questi oggetti espressione regolare rappresentano la stessa sequenza di caratteri: una *a* seguita da una *b* seguita da una *c*.

Quando si usa il costruttore `RegExp`, la sequenza va indicata come stringa normale; si applicano pertanto le solite regole per le barre rovesciate.

Nella seconda notazione, dove la sequenza è racchiusa tra barre, le barre rovesciate sono trattate diversamente. Intanto, visto che la sequenza si chiude con una barra, dobbiamo inserire una barra rovesciata prima di eventuali barre che facciano *parte* della sequenza. Inoltre, le barre rovesciate che non fanno parte di codici speciali, come `\n`, vengono *mantenute* e non ignorate come succede con le stringhe, e pertanto cambiano il

significato della sequenza. Alcuni caratteri, come i punti di domanda e i segni più, hanno significati speciali nelle espressioni regolari; vanno pertanto preceduti da una barra rovesciata se vogliamo che rappresentino il rispettivo carattere.

---

```
var eighteenPlus = /eighteen\+/;
```

---

Per sapere con esattezza quali caratteri vanno preceduti dal carattere di escape (la barra rovesciata), dovete imparare tutti i caratteri che hanno un significato speciale. Ci vorrà del tempo per farlo; pertanto, per il momento, ricordate di mettere una barra rovesciata davanti a tutti i caratteri che non siano lettere, numeri o spazi vuoti.

## Prove di corrispondenza

Le espressioni regolari dispongono di diversi metodi. Il più semplice è `test`. Passandolo a una stringa, restituisce un valore booleano che indica se la stringa contiene la sequenza indicata nell'espressione.

---

```
console.log(/abc/.test("abcde"));
// → true
console.log(/abc/.test("abxde"));
// → false
```

---

Un'espressione regolare che consiste solo di caratteri non speciali rappresenta semplicemente la sequenza di caratteri in essa contenuta. Se in qualunque punto della stringa che esaminiamo (e non solo all'inizio) si trova *abc*, `test` restituisce `true`.

## Trovare serie di caratteri

Per scoprire se una stringa contiene *abc*, si potrebbe anche richiamare `indexof`. Le espressioni regolari ci permettono però di espandere il campo ed esprimere sequenze molto più complicate.

Diciamo, per esempio, che vogliamo trovare qualunque numero. In un'espressione regolare, basta inserire una sequenza di caratteri tra parentesi quadre per trovare uno qualunque dei caratteri tra parentesi.

Le due espressioni seguenti trovano tutte le stringhe che contengono un numero:

---

```
console.log(/[0123456789]/.test("in 1992"));
// → true
console.log(/[0-9]/.test("in 1992"));
// → true
```

---

Tra parentesi quadre, un trattino tra due caratteri indica una serie di caratteri

nell'ordine determinato dal loro numero Unicode. I caratteri da 0 a 9 si trovano in sequenza nell'ordine Unicode (coi codici da 48 a 57), pertanto [0-9] li comprende tutti e trova qualunque cifra.

Ci sono diversi gruppi di caratteri che si possono esprimere con scorciatoie integrate nella sintassi delle espressioni regolari. I numeri sono uno di essi: \d ha lo stesso significato di [0-9].

\d	Qualunque cifra
\w	Un carattere alfanumerico (“carattere word”)
\s	Qualunque spazio vuoto (spazio, tabulatore, a capo e simili)
\D	Un carattere che <i>non sia</i> un numero
\W	Un carattere non alfanumerico
\S	Un carattere che <i>non sia</i> spazio vuoto
.	Qualunque carattere escluso l'a capo

Pertanto, per trovare un formato data/ora come 30-01-2003 15:20, si può usare l'espressione:

---

```
var dateTime = /\d\d-\d\d-\d\d\d\d\d\d \d\d:\d\d/;
console.log(dateTime.test("30-01-2003 15:20"));
// → true
console.log(dateTime.test("30-jan-2003 15:20"));
// → false
```

---

Certo che non è bello da vedere. Ci sono troppe barre rovesciate, che rendono praticamente impossibile leggere la sequenza. Vedremo più avanti una versione più elegante dell'espressione.

Le barre rovesciate si possono usare anche all'interno delle parentesi quadre. Per esempio [/d.] indica qualunque cifra o il punto. Si noti però che il punto da solo, quando è inserito tra parentesi quadre, perde il suo significato speciale e lo stesso vale per altri caratteri speciali, come il +.

Per *invertire* una serie di caratteri, ossia per dire che si vogliono trovare tutti i caratteri *tranne* quelli indicati, potete inserire un circonflesso ^ dopo la parentesi sinistra:

---

```
var notBinary = /[^\d]/;
console.log(notBinary.test("1100100010100110"));
// → false
console.log(notBinary.test("1100100010200110"));
// → true
```

---

## Ripetere parte di una sequenza

Ora sappiamo come trovare una cifra singola. Ma come facciamo a trovare un numero, nel senso di sequenza di una o più cifre?

Aggiungendo un segno + dopo un carattere, in un'espressione regolare, indichiamo che l'elemento può essere ripetuto. Pertanto, /\d+/ trova uno o più caratteri cifra.

---

```
console.log(/\d/.test("123"));
// → true

console.log(/\d/.test(""));
// → false

console.log(/\d*/.test("123"));
// → true

console.log(/\d*/.test(""));
// → true
```

---

L'asterisco (\*) ha un significato simile, con la differenza che la sequenza si può trovare anche zero volte. In pratica, un carattere seguito da un asterisco non nega di trovare la sequenza: se non trova una corrispondenza di testo, si limita a trovarne zero istanze.

Il punto di domanda rende parte della sequenza “facoltativa”, nel senso che la si può trovare zero o una volta. Nell'esempio che segue, può trovare la u, ma trova anche la sequenza dove manca la u.

---

```
var neighbor = /neighbou?r/;
console.log(neighbor.test("neighbour"));
// → true

console.log(neighbor.test("neighbor"));
// → true
```

---

Per indicare che una sequenza deve essere presente esattamente il numero di volte dato, usate le parentesi graffe. Specificare {4} dopo un elemento, per esempio, indica che la sequenza deve presentarsi esattamente quattro volte. In questo modo potete specificare anche un intervallo: {2, 4} indica che l'elemento si deve presentare almeno due e al massimo quattro volte.

Ecco un'altra versione della sequenza data/ora che ammette giorni, mesi e ore in cifra singola e doppia. È anche più chiara da leggere:

---

```
var dateTime = /\d{1,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/;
console.log(dateTime.test("30-1-2003 8:45"));
// → true
```

---

Per indicare intervalli aperti tra parentesi graffe, basta omettere il numero a destra o a

sinistra della virgola. Così per esempio, `{,5}` significa da zero a cinque volte e `{5,}` cinque o più volte.

## Raggruppare le sottoespressioni

Per usare un operatore come `*` o `+` su più di un elemento per volta, potete usare le parentesi. Una parte di espressione regolare racchiusa tra parentesi conta come elemento singolo per gli operatori che la seguono.

```
var cartoonCrying = /boo+(hoo+)+/i;  
console.log(cartoonCrying.test("Boohoooohoooo"));  
// → true
```

Il primo e il secondo segno `+` si applicano solo alla seconda *o* di *boo* e di *hoo*. Il terzo segno `+` si applica invece a tutto il gruppo `(hoo+)` e ne trova una o più sequenze.

La `i` alla fine dell'espressione nell'esempio precedente indica che l'espressione regolare non distingue tra maiuscole e minuscole. Pertanto, trova anche la *B* maiuscola nella stringa di input, anche se la sequenza stessa è in lettere minuscole.

## Corrispondenze e gruppi

Il metodo `test` è il sistema più semplice per trovare corrispondenza con un'espressione regolare. Indica solo se la corrispondenza è stata trovata e null'altro. Le espressioni regolari hanno anche un metodo `exec` (`execute`) che restituisce `null` se non si trovano corrispondenze; diversamente, restituisce un oggetto con informazioni sulla corrispondenza.

```
var match = /\d+/.exec("one two 100");  
console.log(match);  
// → ["100"]  
console.log(match.index);  
// → 8
```

Gli oggetti restituiti da `exec` hanno una proprietà `index` che indica la *posizione* dove inizia la corrispondenza nella stringa. A parte questo, l'oggetto sembra (e in effetti è) un array di stringhe il cui primo elemento è la stringa trovata; nell'esempio precedente, la sequenza di numeri che cercavamo.

I valori stringa hanno un metodo `match` che si comporta in modo simile.

```
console.log("one two 100".match(/\d+/));  
// → ["100"]
```

Quando l'espressione regolare contiene sottoespressioni chiuse tra parentesi, l'array riporta anche il testo corrispondente a quei gruppi. La corrispondenza intera è sempre il primo elemento. Segue poi la parte trovata dal primo gruppo (quello la cui parentesi sinistra viene per prima nell'espressione), poi dal secondo e così via.

---

```
var quotedText = '/([^\"]*)"/;
console.log(quotedText.exec("she said 'hello'"));
// → ["'hello'", "hello"]
```

---

Quando un gruppo non trova corrispondenza (per esempio, quando è seguito da un punto di domanda), la sua posizione nell'array ha come risultato `undefined`. Analogamente, quando un gruppo trova più corrispondenze, l'array riporta solo l'ultima.

---

```
console.log(/bad(ly)?/.exec("bad"));
// → ["bad", undefined]
console.log(/(\d)+/.exec("123"));
// → ["123", "3"]
```

---

I gruppi possono essere utili per estrarre parti di una stringa. Se, oltre a verificare se una stringa contiene una data, volessimo estrarla e costruire un oggetto che la rappresenta, potremmo mettere tra parentesi le sequenze di numeri e recuperare la data dal risultato di `exec`.

Prima, però, dobbiamo fare una piccola divagazione per parlare del modo migliore per registrare valori data e ora in JavaScript.

## Il tipo data

JavaScript ha un tipo di oggetto standard per rappresentare le date, nel senso di punti nel tempo: il tipo `Date`. Se create un oggetto data con `new`, otterrete la data e l'ora correnti.

---

```
console.log(new Date());
// → Wed Dec 04 2013 14:24:57 GMT+0100 (CET)
```

---

Potete anche creare un oggetto per un momento specifico:

---

```
console.log(new Date(2009, 11, 9));
// → Wed Dec 09 2009 00:00:00 GMT+0100 (CET)
console.log(new Date(2009, 11, 9, 12, 59, 59, 999));
// → Wed Dec 09 2009 12:59:59 GMT+0100 (CET)
```

---

JavaScript segue una convenzione dove la numerazione dei mesi parte da zero (e dicembre corrisponde a 11), mentre quella dei giorni parte da uno. Per nulla chiaro e poco intelligente. Fate attenzione!

Gli ultimi quattro argomenti (ore, minuti, secondi e millisecondi) sono facoltativi e prendono valore zero se non altrimenti definiti.

La marcatura oraria è memorizzata come numero di millisecondi a partire dall'inizio del 1970 e usa numeri negativi per le date precedenti. Questa è la convenzione stabilita dal "tempo Unix", che fu inventato più o meno allora. Richiamando il metodo `getTime` su un oggetto `Date` restituisce questo numero che, come potete immaginare, è molto alto.

---

```
console.log(new Date(2013, 11, 19).getTime());
// → 1387407600000
console.log(new Date(1387407600000));
// → Thu Dec 19 2013 00:00:00 GMT+0100 (CET)
```

---

Se passate al costruttore `Date` un solo argomento, esso viene trattato come conteggio di millisecondi. Per recuperare il conteggio di millisecondi corrente, potete creare un nuovo oggetto `Date` e richiamare su di esso `getTime`; oppure, potete richiamare la funzione `Date.now`.

Gli oggetti `Date` offrono altri metodi, come `getFullYear`, `getMonth`, `getDate`, `getHours`, `getMinutes` e `getSeconds` per estrarre i singoli componenti. Esiste anche `getYear`, che restituisce un valore anno a due cifre (come 93 o 14), peraltro abbastanza inutile.

Aggiungendo le parentesi per raggruppare le parti dell'espressione che ci interessano, possiamo ora creare un oggetto `Date` da una stringa.

---

```
function findDate(string) {
  var dateTime = /(\d{1,2})-(\d{1,2})-(\d{4})/;
  var match = dateTime.exec(string);
  return new Date(Number(match[3]),
    Number(match[2]) - 1,
    Number(match[1]));
}
console.log(findDate("30-1-2003"));
// → Thu Jan 30 2003 00:00:00 GMT+0100 (CET)
```

---

## Confini di parola e di stringa

Purtroppo, `findDate` può estrarre anche una data assurda, come 00-1-3000 dalla stringa "100-1-30000". Poiché la corrispondenza può avvenire in qualunque punto della stringa, in questo caso parte dal secondo carattere e si ferma al penultimo.

Se vogliamo forzare la corrispondenza a tutta la stringa, possiamo aggiungere i marcatori `^` e `$`. Il segno `^` indica l'inizio e il segno `$` la fine della stringa di input.

Pertanto, `/^\d+$/` trova una stringa intera di una o più cifre, `/^!/` trova tutte le stringhe che iniziano con un punto esclamativo e `/x^/` non trova nessuna stringa perché non ci può essere un carattere (sia esso `x` o qualunque altro) prima dell'inizio della stringa.

Se però vogliamo solo accertarci che la data inizi e finisce al confine di una parola, possiamo usare il marcatore `\b`. Il confine di parola può essere l'inizio o la fine della stringa o qualunque punto della stringa che abbia un carattere word (come `\w`) da una parte e un carattere non-word dall'altra.

---

```
console.log(/cat/.test("concatenate"));
// → true
console.log(/\bcat\b/.test("concatenate"));
// → false
```

---

Si noti che un marcatore di confine non rappresenta un carattere: forza semplicemente la ricerca di corrispondenze con l'espressione regolare solo quando una certa condizione ha valore nella posizione che occupa nella sequenza.

## Sequenze alternative

Diciamo di voler sapere se un brano di testo contiene non solo un numero, ma un numero seguito da una delle parole *pig*, *cow* o *chicken*, al singolare o al plurale.

Potremmo scrivere tre espressioni regolari e provarle una per volta, ma c'è un modo migliore. Il carattere `|` indica che le sequenze vanno cercate in alternativa tra l'espressione di destra e quella di sinistra. Così possiamo scrivere:

---

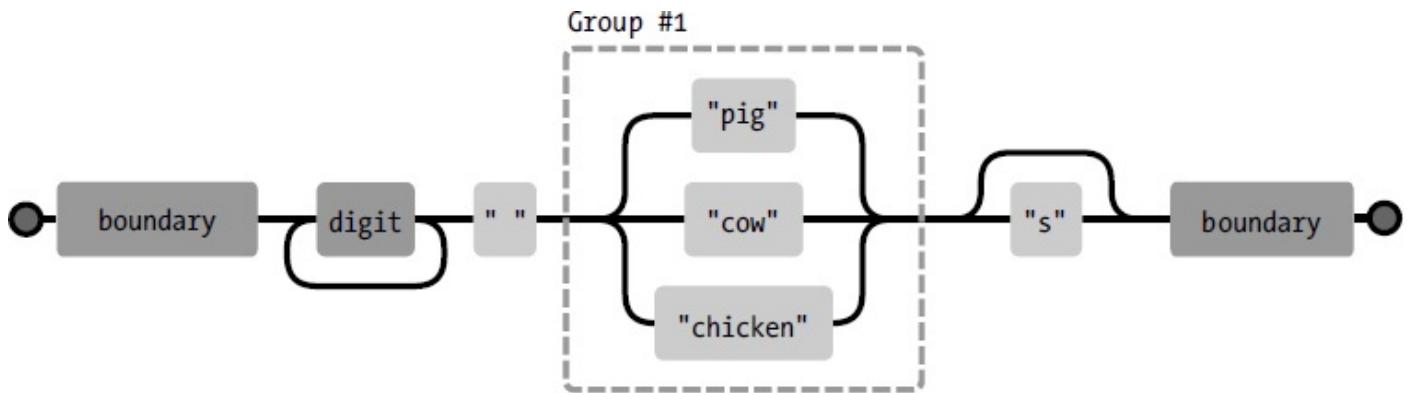
```
var animalCount = /\b\d+ (pig|cow|chicken)s?\b/;
console.log(animalCount.test("15 pigs"));
// → true
console.log(animalCount.test("15 pigchickens"));
// → false
```

---

Possiamo usare le parentesi per delimitare la parte della sequenza a cui si applica l'operatore `|` e inserire più operatori per esprimere l'alternativa tra più di due sequenze.

## La meccanica delle corrispondenze

Possiamo pensare alle espressioni regolari come a diagrammi di flusso. Questo è il diagramma per l'espressione dell'esempio precedente:



La nostra espressione cerca la sequenza seguendo il percorso da sinistra a destra del diagramma. Nella stringa manteniamo la posizione corrente e, ogni volta che ci spostiamo in un blocco, verifichiamo se la parte della stringa dopo la posizione corrente corrisponde al blocco.

Pertanto, se cerchiamo la corrispondenza per "the 3 pigs" con quell'espressione regolare, il flusso nel diagramma si svolgerà come segue:

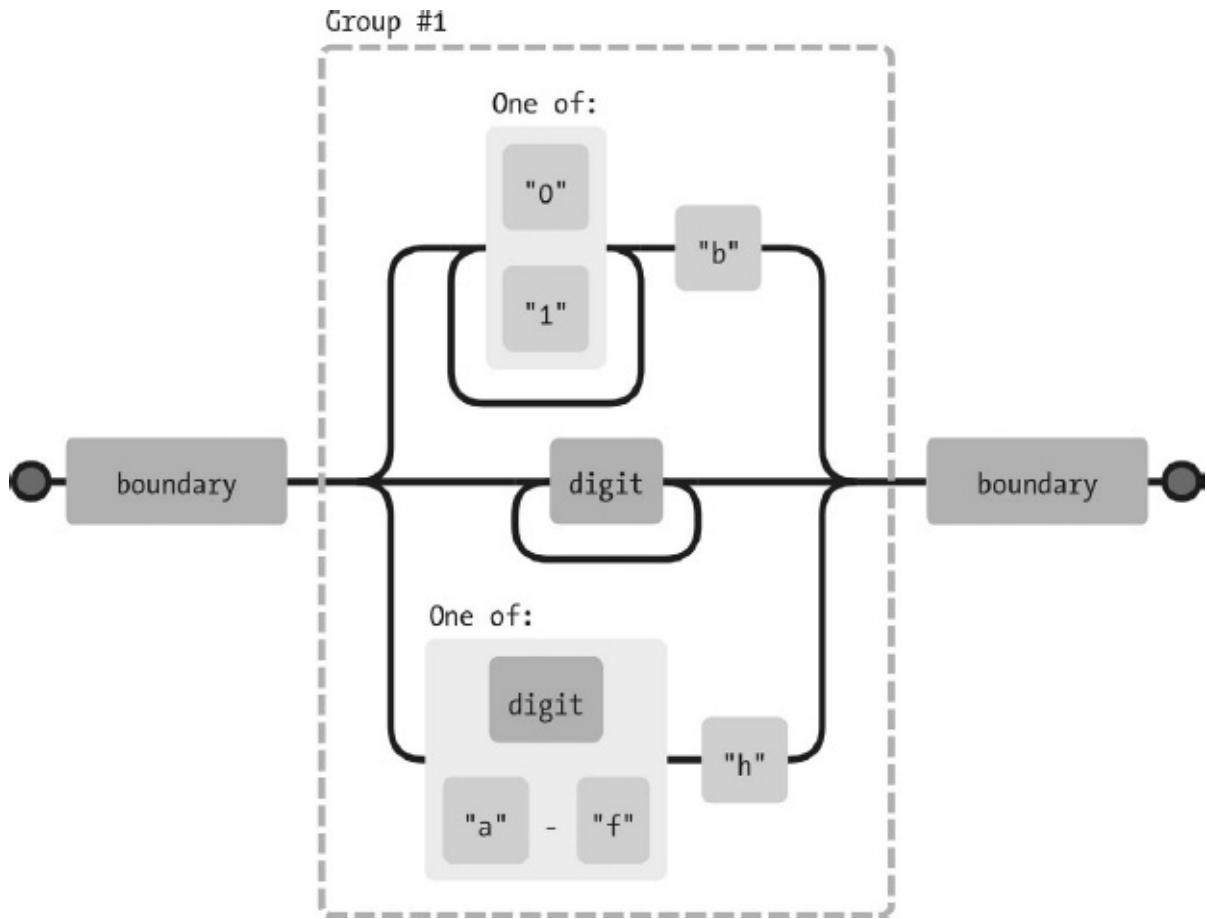
- In posizione 4, c'è un confine di parola e possiamo pertanto superare il primo blocco.
- Sempre in posizione 4, troviamo un numero; possiamo pertanto superare il secondo blocco.
- In posizione 5, una parte del percorso torna in ciclo a prima del secondo blocco (numero), mentre l'altra va avanti attraverso il blocco che contiene lo spazio. Qui c'è uno spazio e non un numero, e pertanto dobbiamo seguire il secondo percorso.
- Arriviamo adesso alla posizione 6 (l'inizio di "pigs") e al trivio del diagramma. Qui non troviamo né "cow", né "chicken", ma troviamo "pig" e seguiamo pertanto quel percorso.
- In posizione 9, dopo il trivio, un percorso salta il blocco *s* e va diretto all'ultimo confine di parola, mentre l'altro trova una *s*. Qui c'è un carattere *s* e non un confine di parola: oltrepassiamo pertanto il blocco *s*.
- Arriviamo in posizione 10 (la fine della stringa) e troviamo solo un confine di parola. Poiché la fine di una stringa vale come confine di parola, attraversiamo l'ultimo blocco per completare la corrispondenza.

Concettualmente, il motore per le espressioni regolari cerca la corrispondenza così: parte dall'inizio della stringa e cerca la corrispondenza in quella posizione. In questo caso, c'è un confine di parola e può pertanto andare al di là del primo blocco; ma poiché lì non c'è un numero, non trova corrispondenza per il secondo blocco. Passa quindi al secondo carattere della stringa e ripete il processo, e continua così, finché trova una corrispondenza o raggiunge la fine della stringa e decide che non c'è corrispondenza.

## Backtracking

L'espressione regolare `/\b([01]+b|\d+|[\da-f]h)\b/` trova o un numero binario seguito da una *b*, o un numero decimale normale senza suffisso, o un numero esadecimale (ossia,

in base 16, dove le lettere da *a* a *f* rappresentano le cifre da 10 a 15) seguito da una *h*. Ecco il diagramma corrispondente:



Con quest'espressione, può succedere che si prenda il percorso in alto (binario) anche se l'input non contiene un numero binario. Per esempio, se si prova con la stringa "103", diventa evidente solo quando si raggiunge il 3 che ci troviamo nel percorso sbagliato. La stringa si trova nell'espressione, ma non nel percorso dove ci troviamo.

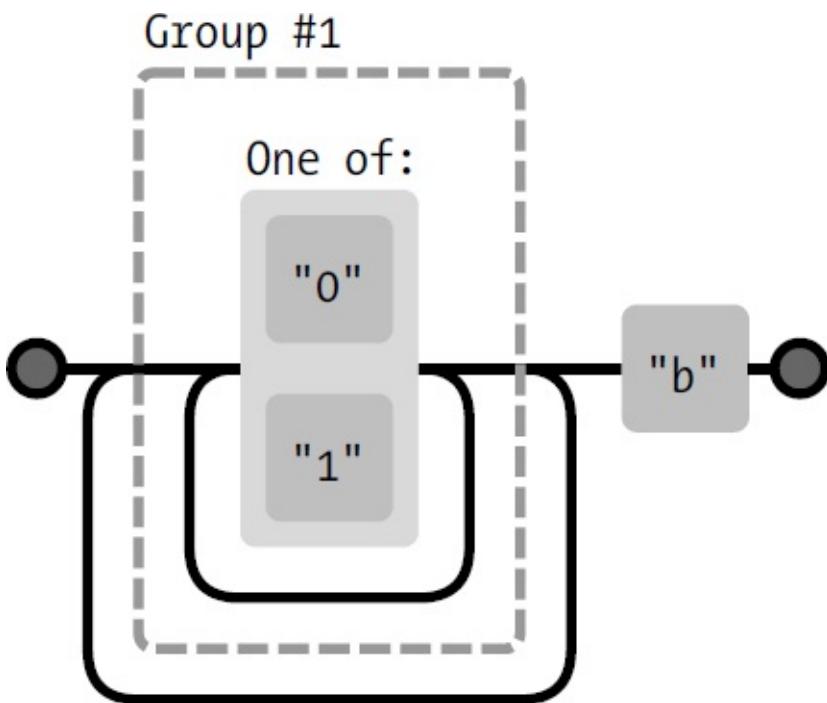
Pertanto, si torna indietro col *backtracking*. Quando entra in un percorso, la funzione ricorda la posizione corrente (in questo caso, all'inizio della stringa, subito dopo il primo blocco del diagramma) e può pertanto tornare indietro e provare un percorso diverso se quello scelto non dà il risultato cercato. Per la stringa "103", una volta arrivata al 3 prova il percorso per i numeri decimali, dove trova corrispondenza.

La funzione si interrompe appena trova una corrispondenza piena. Ciò significa che se ci sono più percorsi dove si può trovare corrispondenza, viene usato solo il primo, nell'ordine dato ai percorsi nell'espressione regolare.

Il backtracking ha luogo anche per gli operatori di ripetizione, come *+* e *\**. Se si cerca la corrispondenza con */^.x\*/* in "abcxe", la parte *.\** verifica prima tutta la stringa. Il motore rileva che ci vuole una *x* per trovare corrispondenza nella sequenza. Poiché non ci sono *x* dopo la fine della stringa, l'operatore *\** cerca corrispondenza scartando l'ultimo carattere. Siccome non c'è una *x* dopo *abcx*, torna indietro di nuovo e verifica solo *abc*. Trova ora una *x* dove ci deve essere e restituisce pertanto una corrispondenza nelle posizioni da 0 a 4.

Si possono scrivere espressioni regolari che svolgono un sacco di backtracking: il

problema si verifica quando una sequenza può trovare diversi tipi di corrispondenza nei dati di input. Per esempio, se facciamo un po' di confusione nell'impostare un'espressione regolare per un numero binario, potremmo scrivere per sbaglio `/([01]+)+b/`.



Per trovare corrispondenza in lunghe serie di zero e uno senza che ci sia una *b* alla fine, la funzione prima attraversa il ciclo interno, finché finisce i numeri. Poi rileva che manca la *b* e torna indietro di una posizione; ripete una volta il ciclo esterno e rinuncia, cercando ancora una volta di tornare indietro dal ciclo interno. E continuerà a provare tutti i percorsi possibili che passano attraverso i due cicli. Ciò significa che il volume di lavoro *raddoppia* per ogni carattere in più. Anche solo con una dozzina di caratteri, ci vuole un'eternità per trovare la corrispondenza.

## Il metodo replace

I valori stringa offrono un metodo `replace`, che può essere utilizzato per sostituire parte della stringa con un'altra stringa:

```
console.log("papa".replace("p", "m"));
// → mama
```

Il primo argomento può essere anche un'espressione regolare. In questo caso, viene sostituito il primo elemento dove l'espressione regolare trova corrispondenza. Aggiungendo un'opzione `g (globale)` all'espressione regolare, vengono sostituiti tutti gli elementi corrispondenti e non solo il primo.

```
console.log("Borobudur".replace(/[ou]/, "a"));
// → Barobudur
console.log("Borobudur".replace(/[ou]/g, "a"));
// → Barabadar
```

---

Sarebbe stato bello avere un argomento in più, o un metodo `replaceAll`, per stabilire se sostituire una sola o tutte le corrispondenze. Purtroppo, invece, la scelta va affidata a una proprietà dell'espressione regolare.

La forza delle espressioni regolari col metodo `replace` viene dalla possibilità di fare riferimento all'indietro ai gruppi dove si è trovata corrispondenza nella stringa da sostituire. Per esempio, immaginiamo di avere una lunga stringa che contiene nomi di persone, un nome per riga, con prima il cognome e poi il nome nel formato `Lastname, Firstname`. Se vogliamo scambiare la posizione dei nomi ed eliminare la virgola per arrivare al formato `Firstname Lastname`, possiamo usare il codice seguente:

---

```
console.log(  
  "Hopper, Grace\nMcCarthy, John\nRitchie, Dennis"  
    .replace(/([\w ]+), ([\w ]+)/g, "$2 $1"));  
// → Grace Hopper  
//   John McCarthy  
//   Dennis Ritchie
```

---

I codici `$1` e `$2` nella stringa di sostituzione si riferiscono ai gruppi tra parentesi della sequenza. `$1` viene sostituito dal testo trovato nel primo gruppo, `$2` da quello nel secondo e così via fino a `$9`. Alla corrispondenza nel suo insieme si fa riferimento con `$.&`.

È inoltre possibile passare una funzione, invece di una stringa, come secondo argomento per `replace`. Per ciascuna sostituzione, viene richiamata la funzione coi gruppi corrispondenti (e con tutta la corrispondenza) come argomenti; il suo valore di restituzione viene inserito nella nuova stringa.

Ecco un semplice esempio:

---

```
var s = "the cia and fbi";  
console.log(s.replace(/\b(fbi|cia)\b/g, function(str) {  
  return str.toUpperCase(); }));  
// → the CIA and FBI
```

---

Ed eccone uno più interessante:

---

```
var stock = "1 lemon, 2 cabbages, and 101 eggs";  
function minusOne(match, amount, unit) {  
  amount = Number(amount) - 1;  
  if (amount == 1) // only one left, remove the 's'  
    unit = unit.slice(0, unit.length - 1);  
  else if (amount == 0)  
    amount = "no";
```

```
    return amount + " " + unit;
}
console.log(stock.replace(/(\d+) (\w+)/g, minusOne));
// → no lemon, 1 cabbage, and 100 eggs
```

---

In questo secondo esempio, il metodo cerca in una stringa tutte le occorrenze di un numero seguito da una parola alfanumerica e restituisce una stringa dove i valori trovati sono diminuiti di uno.

Il gruppo `(\d+)` diventa l'argomento `amount` della funzione e il gruppo `(\w+)` viene associato a `unit`. La funzione converte `amount` in un numero, cosa che funziona sempre, visto che ha trovato `\d+`, e calcola delle correzioni se è rimasto solo uno o zero.

## Operatori greedy

Non ci vuole molto a usare `replace` per scrivere una funzione che elimina tutti i commenti da un brano di JavaScript. Ecco un primo esperimento:

```
function stripComments(code) {
  return code.replace(/\//.*|\/*[^]*\//g, "");
}

console.log(stripComments("1 /* 2 */3"));
// → 1 + 3

console.log(stripComments("x = 10;// ten!"));
// → x = 10;

console.log(stripComments("1 /* a *//* b */ 1"));
// → 1 1
```

---

La parte prima dell'operatore `or` si limita a cercare due caratteri barra seguiti da qualunque carattere che non sia un ritorno a capo. La parte per i commenti su più righe è più complessa. Usiamo `[^]` (qualunque carattere che non sia nella serie di caratteri vuota) per trovare qualunque carattere. Qui non possiamo usare un punto perché i commenti su più righe possono continuare su una nuova riga e i punti non corrispondono al carattere di ritorno a capo.

L'output dell'esempio, però, non è quello che cercavamo. Come mai?

Come dicevo nel paragrafo sul backtracking, la parte dell'espressione `[^]*` cerca di trovare tutto quel che può. Se questo fa fallire la parte successiva della sequenza, la funzione si sposta all'indietro di un carattere e riprova da lì. Nell'esempio, prima cerca corrispondenza per tutto quel che è rimasto della stringa e poi torna indietro da lì. Troverà un'occorrenza di `/*` dopo essere tornata indietro di quattro posizioni. Quello, però, non è ciò che vogliamo: volevamo trovare un solo commento e non andare fino alla fine del codice per trovare la fine dell'ultimo blocco di commenti.

A causa di questo comportamento, diciamo che gli operatori di ripetizione (+, \*, ? e {}) sono *greedy*, ingordi, nel senso che trovano corrispondenze ovunque possono e tornano indietro da lì. Se aggiungete un punto di domanda dopo questi operatori (+?, \*?, ??, {}?), essi non sono più greedy e cercano il minimo possibile, trovando più corrispondenze solo quando il resto della sequenza non corrisponde alla corrispondenza più piccola.

Questo è esattamente quel che vogliamo in questo caso. Con l'asterisco che trova la sequenza di caratteri più piccola che finisce con /\*, troviamo la corrispondenza con un solo blocco di commenti e null'altro.

---

```
function stripComments(code) {  
    return code.replace(/\/\/.*|\/\*[^\]*?\*\//g, "");  
}  
console.log(stripComments("1 /* a *//* b */ 1"));  
// → 1 + 1
```

---

In molti casi, i bachi nei programmi che fanno uso di espressioni regolari si trovano proprio nell'uso accidentale di operatori greedy, laddove sarebbe stato preferibile uno non-greedy. Quando usate un operatore di ripetizione, ricordatevi di valutare se la versione non-greedy può bastare.

## Creare dinamicamente oggetti RegExp

Ci sono situazioni dove non sappiamo esattamente quale sequenza di caratteri dobbiamo trovare. Immaginiamo per esempio di voler trovare un nome utente in un brano di testo, per poi racchiuderlo tra segni di sottolineatura per evidenziarlo. Dal momento che saprete il nome solo una volta che il programma è in esecuzione, non potete usare la notazione a barre.

Potete, però, creare una stringa e applicarvi il costruttore RegExp. Ecco un esempio:

---

```
var name = "harry";  
var text = "Harry is a suspicious character.";  
var regexp = new RegExp("\b(" + name + ")\\b", "gi");  
console.log(text.replace(regexp, "_$1_"))  
// → _Harry_ is a suspicious character.
```

---

Quando creiamo i marcatori di confine \b, dobbiamo usare due barre rovesciate perché le stiamo inserendo in una stringa e non in un'espressione regolare chiusa tra barre. Il secondo argomento del costruttore RegExp contiene l'opzione per l'espressione regolare, in questo caso "gi" per indicare *global* (globale) e indifferentemente maiuscole o minuscole.

Che succederebbe, invece, se il nome fosse "dea+h1[]rd", magari perché è quello

scelto da un giovane nerd? In quel caso, l'espressione regolare non avrebbe senso e non troverebbe il nome dell'utente.

Come soluzione, possiamo aggiungere delle barre rovesciate davanti ai caratteri di cui non ci fidiamo. Aggiungere barre rovesciate davanti ai caratteri alfabetici è una pessima idea, perché cose come \b e \n hanno significati speciali. Possiamo, però, tranquillamente usare il carattere di escape davanti a tutti i caratteri che non sono alfanumerici né spazi.

---

```
var name = "dea+h1[]rd";
var text = "This dea+h1[]rd guy is super annoying.";
var escaped = name.replace(/[\w\s]/g, "\\$&");
var regexp = new RegExp("\\b(" + escaped + ")\\b", "gi");
console.log(text.replace(regexp, "_$1_"));
// → This _dea+h1[]rd_ guy is super annoying.
```

---

## Il metodo search

Sulle stringhe non si può richiamare il metodo `indexof` con un'espressione regolare. Esiste, però, un altro metodo, `search`, che accetta come argomento un'espressione regolare. Come `indexof`, restituisce l'indice della prima posizione dove si trova corrispondenza o -1 se non se ne trova.

---

```
console.log(" word".search(/\S/));
// → 2
console.log(" ".search(/\S/));
// → -1
```

---

Purtroppo, non c'è modo di indicare che la corrispondenza deve partire da una posizione data, come si può fare invece col secondo argomento di `indexof`.

## La proprietà lastIndex

Nemmeno il metodo `exec` offre un modo comodo per iniziare la ricerca da una posizione data nella stringa. Ne offre però uno piuttosto scomodo.

Gli oggetti espressioni regolari hanno proprietà. Una di esse è `source`, che contiene la stringa dalla quale si è realizzata l'espressione. Un'altra proprietà è `lastIndex`, che determina, ma solo in alcune circostanze, da dove inizierà la prossima corrispondenza.

Le circostanze sono che l'espressione regolare abbia l'opzione globale (g) attiva e che la corrispondenza si trovi attraverso il metodo `exec`. Anche in questo caso, una soluzione più sensata sarebbe stata di consentire il passaggio di un argomento aggiuntivo al metodo `exec`, ma del resto il buon senso non è una caratteristica tipica dell'interfaccia per le

espressioni regolari di JavaScript.

---

```
var pattern = /y/g;  
pattern.lastIndex = 3;  
var match = pattern.exec("xyzzy");  
console.log(match.index);  
// → 4  
console.log(pattern.lastIndex);  
// → 5
```

---

Se si trova corrispondenza, la chiamata a `exec` automaticamente aggiorna la proprietà `lastIndex` in modo che punti alla posizione immediatamente successiva. Se non si trova corrispondenza, `lastIndex` viene reimpostato su zero, che è il valore degli oggetti espressione regolare al momento della costruzione con `new`.

Quando si usa il valore di un'espressione regolare globale per più chiamate a `exec`, gli aggiornamenti automatici della proprietà `lastIndex` possono dare problemi, in quanto l'espressione regolare potrebbe iniziare (per caso) nella posizione di indice "avanzata" da una chiamata precedente.

---

```
var digit = /\d/g;  
console.log(digit.exec("here it is: 1"));  
// → ["1"]  
console.log(digit.exec("and now: 1"));  
// → null
```

---

Un altro effetto interessante dell'opzione globale è che cambia il comportamento del metodo `match` sulle stringhe. Quando viene richiamato con un'espressione regolare, invece di restituire un array simile a quello restituito da `exec`, `match` trova tutte le corrispondenze nella stringa e restituisce un array contenente le stringhe trovate.

---

```
console.log("Banana".match(/an/g));  
// → ["an", "an"]
```

---

Fate dunque attenzione con le espressioni regolari globali. I casi dove sono necessarie (le chiamate a `replace` e le posizioni dove volete usare esplicitamente `lastIndex`) sono praticamente le sole situazioni dove vanno usate.

## ***Passare in ciclo i risultati***

Succede abbastanza spesso di voler esaminare tutte le occorrenze di una certa sequenza in una stringa, in modo da accedere all'oggetto `match` nel corpo del ciclo attraverso `lastIndex` ed `exec`.

---

```
var input = "A string with 3 numbers in it... 42 and 88.";
```

```
var number = /\b(\d+)\b/g;
var match;
while (match = number.exec(input))
    console.log("Found", match[1], "at", match.index);
// → Found 3 at 14
//   Found 42 at 33
//   Found 88 at 40
```

---

Questo è possibile perché il valore di un'espressione di assegnazione è il valore assegnato. Pertanto, usando `match = number.exec(input)` come condizione nella dichiarazione `while`, cerchiamo la corrispondenza all'inizio di ogni iterazione, salviamo il risultato in una variabile e concludiamo il ciclo quando non troviamo più corrispondenze.

## Analizzare un file INI

Per concludere il capitolo, esaminiamo un problema che richiede le espressioni regolari. Immaginiamo di dover scrivere un programma che raccolga automaticamente da Internet informazioni sui nostri nemici (e qui non riportiamo tutto il programma, ma solo la parte che legge il file di configurazione: scusate se vi deludo!). Il file di configurazione ha l'aspetto seguente:

---

```
searchengine=http://www.google.com/search?q=$1
spitefulness=9.7
; comments are preceded by a semicolon... (i commenti sono preceduti da un punto e virgola)
; each section concerns an individual enemy (ogni sezione riguarda un singolo nemico)

[larry]
fullname=Larry Doe
type=kindergarten bully
website=http://www.geocities.com/CapeCanaveral/11451

[gargamel]
fullname=Gargamel
type=evil sorcerer
outputdir=/home/marijn/enemies/gargamel
```

---

Le regole esatte per questo formato (che è in realtà ampiamente utilizzato e si chiama di solito file *INI*) sono le seguenti:

- Le righe vuote e le righe che iniziano con punto e virgola vengono ignorate.
- Le righe racchiuse tra [ e ] iniziano una nuova sezione.

- Le righe che contengono un identificatore alfanumerico seguito da un carattere = aggiungono un'impostazione alla sezione corrente.
- Qualunque altra cosa non ha valore.

Dobbiamo ora convertire una stringa come questa in un array di oggetti, ciascuno con una sua proprietà name e un array di impostazioni. Avremo bisogno di un oggetto per ciascuna delle sezioni e di uno per le impostazioni globali, nella parte in alto.

Poiché il formato va elaborato riga per riga, possiamo cominciare col dividere il file in righe separate. Nel [Capitolo 6](#) avevamo usato `string.split("\n")`. Alcuni sistemi operativi, però, non usano solo il ritorno a capo, ma lo fanno precedere da un carattere invio ("\\r\\n"). Poiché il metodo `split` accetta come argomento anche un'espressione regolare, possiamo applicarlo a un'espressione regolare come `/\\r?\\n/` per dividere le righe in entrambe le situazioni.

---

```
function parseINI(string) {
  // Comincia con un oggetto che contiene i campi di livello superiore
  var currentSection = {name: null, fields: []};
  var categories = [currentSection];
  string.split(/\r?\n/).forEach(function(line) {
    var match;
    if (/^\s*(;.*)?$/ .test(line)) {
      return;
    } else if (match = line.match(/^\[(.*)\]$/)) {
      currentSection = {name: match[1], fields: []};
      categories.push(currentSection);
    } else if (match = line.match(/^(\w+)=(.*)$/)) {
      currentSection.fields.push({name: match[1],
        value: match[2]});
    } else {
      throw new Error("Line '" + line + "' is invalid.");
    }
  });
  return categories;
}
```

---

Il codice passa il file riga per riga, aggiornando man mano l'oggetto per la sezione corrente, `currentSection`. Prima, verifica se si può ignorare la riga con l'espressione `/^\s*(;.*)?$/`. Riuscite a vedere come funziona? La parte tra parentesi cerca i commenti e il segno ? fa in modo che trovi anche le righe che contengono solo spazio vuoto.

Se la riga non è un commento, il codice verifica se apre una nuova sezione. In questo

caso, crea un nuovo oggetto `currentSection` al quale aggiungere le impostazioni che seguono.

L'ultima possibilità sensata è che la riga sia un'impostazione normale, che il codice aggiunge alla sezione corrente.

Se una riga non corrisponde a nessuna di queste forme, la funzione lancia un errore.

Si noti l'uso ricorrente di `^` e `$` per assicurare che l'espressione trovi tutta la riga e non solo parte di essa. Se si lasciano fuori questi modificatori, il codice per lo più funziona, ma in alcuni casi dà risultati imprevedibili, un baco difficile da rilevare.

La sequenza `if (match = string.match(...))` è simile allo stratagemma dove si usa un'assegnazione come condizione per una dichiarazione `while`. Non potete essere sempre sicuri che la chiamata a `match` avrà successo: pertanto, potete accedere all'oggetto solo all'interno di una dichiarazione `if` che lo verifichi. Per non interrompere la catena di `if`, assegniamo il risultato (la corrispondenza) a una variabile e usiamo quell'assegnazione come verifica per la dichiarazione `if`.

## Caratteri internazionali

A causa del suo sviluppo iniziale piuttosto semplicistico, e del fatto che questo avvio semplicistico venne successivamente formalizzato come comportamento normale, le espressioni regolari in JavaScript sono poco evolute e non riconoscono i caratteri che non appaiono in lingua inglese. Per esempio, nelle espressioni regolari di JavaScript il “carattere word” può soltanto essere uno dei 26 caratteri dell’alfabeto latino (maiuscolo e minuscolo) e, per qualche strana ragione, il segno di sottolineatura. Caratteri come  $\epsilon$  o  $\beta$ , che pure sono caratteri word, non vengono trovati con `\w` e troveranno anzi corrispondenza con `\W` maiuscolo, ossia la categoria dei caratteri non-word.

Per qualche accidente storico, `\s` (spazio vuoto) non ha questo problema e trova tutti i caratteri che lo standard Unicode considera come spazio vuoto, comprese cose come il segno di spazio unificatore e il separatore di vocali della lingua mongola.

In altri linguaggi, la sintassi per le espressioni regolari offre regole per trovare categorie specifiche di caratteri Unicode, tra cui “tutte le maiuscole”, “tutta la punteggiatura” o “tutti i caratteri di controllo”. È prevista l’aggiunta di strumenti per queste categorie in JavaScript, ma purtroppo non se ne parla come di una questione che verrà attuata nel prossimo futuro.

## Riepilogo

Le espressioni regolari sono oggetti che rappresentano sequenze di caratteri in una stringa. Dispongono di una loro sintassi per esprimere queste sequenze.

/[abc]/	Qualunque carattere di una serie di caratteri
/[^abc]/	Qualunque carattere non in una serie di caratteri
/[0-9]/	Qualunque carattere compreso in un intervallo
/x+ /	Una o più occorrenze del carattere x
/x+? /	Una o più occorrenze, non greedy
/x*/	Zero o più occorrenze
/x? /	Zero o una occorrenza
/x{2,4}/	Tra due e quattro occorrenze
/(abc)/	Un gruppo
/a b c/	Una di una serie di sequenze
/\d/	Qualunque carattere cifra
/\w/	Un carattere alfanumerico (carattere word)
/\s/	Qualunque carattere spazio vuoto
/./	Qualunque carattere tranne l'a capo
/\b/	Un confine di parola
/^ /	Inizio dell'input
/\$/	Fine dell'input

Le espressioni regolari hanno un metodo `test` per verificare se si trova corrispondenza in una stringa data. Offrono inoltre un metodo `exec` che, una volta trovata la corrispondenza, restituisce un array con tutti i gruppi trovati. Tale array ha una proprietà `index` che indica dove è iniziata la corrispondenza.

Le stringhe hanno un metodo `match` per confrontarle con un'espressione regolare e un metodo `search` per trovare una corrispondenza, che restituisce solo la posizione iniziale della corrispondenza. Il metodo `replace` consente di sostituire delle sequenze, del tutto o in parte, con una stringa di sostituzione. In alternativa, potete passare una funzione a `replace`, che la utilizza per costruire una stringa di sostituzione basata sul testo trovato e sui gruppi corrispondenti.

Le espressioni regolari possono avere opzioni, che vengono indicate dopo la barra di chiusura. L'opzione `i` indica che la ricerca non distingue tra maiuscole e minuscole, mentre l'opzione `g` rende l'espressione *globale*; cosa che, tra l'altro, fa sì che il metodo `replace` sostituisca tutte le istanze invece della prima soltanto.

Si può usare il costruttore `RegExp` per creare un valore espressione regolare da una stringa.

Le espressioni regolari sono uno strumento molto affilato, ma con un manico difficile da maneggiare. Semplificano grandemente alcuni compiti, ma diventano facilmente impossibili da gestire con problemi complicati. Parte dell'abilità richiesta per la loro applicazione sta nella capacità di resistere alla tentazione di usarle in situazioni dove non servono.

## Esercizi

È quasi inevitabile che questi esercizi vi rendano confusi e frustrati dal comportamento inesplicabile di qualche espressione regolare. Potete provare l'espressione con uno strumento online, come quello che si trova su [debuggex.com](http://debuggex.com), per vedere se la sua virtualizzazione corrisponde a quel che avevate in mente e fare esperimenti col modo in cui risponde alle stringhe di output.

### Il golf

Nel gergo dei programmatori, il termine *code golf* indica un gioco dove si cerca di esprimere un determinato programma col minor numero di caratteri possibile. Analogamente, *regexp golf* indica l'abilità di scrivere l'espressione regolare più piccola per trovare corrispondenze di una sequenza particolare, e solo di quella.

Scrivete un'espressione regolare per ciascuno dei termini che seguono, che verifichi se le sottostinghe date sono presenti in una stringa.

L'espressione regolare deve trovare solo le stringhe che contengono una (e una sola) delle sottostinghe riportate. Non preoccupatevi dei confini delle parole se non sono citati esplicitamente. Quando l'espressione funziona, provate a renderla ancora più breve.

1. *car* e *cat*
2. *pop* e *prop*
3. *ferret*, *ferry* e *ferrari*
4. Tutte le parole che finiscono con *ious*
5. Un carattere spazio vuoto seguito da un punto, una virgola, un punto e virgola o un due punti
6. Una parola di più di sei lettere
7. Una parola senza la lettera *e*

Fate riferimento alla tabella riportata nel paragrafo Riepilogo per la sintassi. Provate tutte le soluzioni con qualche stringa.

### Stile delle citazioni

Immaginate di aver scritto una storia usando le virgolette semplici per contrassegnare i brani di dialogo. Ora volete sostituire tutte le virgolette semplici con virgolette doppie, mentre le virgolette semplici usate in altre parti della storia (per esempio, gli apostrofi) vanno lasciate come sono.

Provate a descrivere la sequenza che distingue tra questi due tipi di virgolette e impostate una chiamata al metodo `replace` che effettui le sostituzioni correttamente.

### Ancora numeri

Una serie di cifre si trova con la semplice espressione regolare `/\d+/`.

Scrivete un'espressione che trovi solo i numeri espressi in stile JavaScript. Dovete prevedere la possibilità di avere un segno più o meno davanti al numero, il punto decimale e la notazione dell'esponente, `5e-3` o `1E10`, anche in questo caso con o senza un segno davanti all'esponente. Notate che non è obbligatorio che ci siano cifre davanti o dopo il punto, ma il punto non può essere da solo. In altre parole, `.5` e `5.` sono numeri validi in JavaScript, ma un punto solitario non lo è.

# 10

## MODULI

*Un programmatore principiante scrive programmi come una formica costruisce il formicaio, un pezzo per volta, senza pensare alla struttura più grande. I suoi programmi saranno come castelli di sabbia. Per un po' staranno in piedi, ma se crescono ancora finiranno col crollare.*

*Quando si rende conto di questo problema, il programmatore passerà un sacco di tempo a pensare alla struttura. I suoi programmi avranno una struttura rigida, come sculture nella roccia. Sono solidi, ma quando bisogna modificarli, sarà necessario usare la violenza.*

*Il programmatore esperto sa quando applicare strutture e quando lasciare le cose nella loro forma più semplice. I suoi programmi sono come creta, solidi ma malleabili.*

Jamie Zawinski

Tutti i programmi hanno una forma. Nel suo aspetto più semplice, questa forma è determinata dalla divisione in funzioni e dai blocchi all'interno di esse. I programmatori hanno grande libertà nel dare una struttura ai loro programmi. La loro forma dipende più dal gusto del programmatore che dalla funzionalità e dallo scopo dei programmi.

Prendendo i programmi più grossi nel loro insieme, le singole funzioni finiscono col fondersi nello sfondo. In questi casi, i programmi possono diventare più facili da leggere se si introducono unità di organizzazione più grandi.

I *moduli* dividono i programmi in blocchi di codice, raggruppati in base a determinati criteri. In questo capitolo esploriamo alcuni dei benefici offerti da questa divisione e illustriamo le tecniche per costruire moduli in JavaScript.

### Perché i moduli aiutano

Gli autori dividono i libri in capitoli e paragrafi per molte ragioni. Queste suddivisioni aiutano chi legge a vedere com'è strutturato il libro e a trovare le parti che più gli interessano, e aiutano l'autore mettendo in evidenza l'argomento di ciascuna sezione.

I vantaggi di organizzare un programma in file o moduli distinti sono simili. Un'impostazione strutturata aiuta chi non è ancora a suo agio col codice a trovare quel che cerca e aiuta il programmatore a raggruppare gli aspetti collegati tra loro.

Alcuni programmi sono organizzati proprio come documenti scritti, con una struttura ben definita, che incoraggia il lettore a esaminare ordinatamente il programma, e con abbondante prosa (i commenti), che offre una descrizione coerente del codice. Leggere il programma fa così meno paura, perché leggere del codice che non si conosce è sempre un po' inquietante, ma ha lo svantaggio di costare più fatica. Rende anche il programma più difficile da modificare, perché i commenti rimangono legati al codice più strettamente. Questo stile si chiama *programmazione letteraria*. I capitoli sui progetti, in questo libro, possono essere considerati programmi letterari.

In linea di massima, strutturare i programmi costa energia. Nelle fasi iniziali di un progetto, quando non siete ancora sicuri di che cosa vada dove e quali moduli il programma richieda, sempre che li richieda, è molto meglio seguire un'impostazione minimalista e senza struttura. Mettete quel che serve ovunque stia bene, almeno finché il codice non comincia a prendere un aspetto più definito. In questo modo, non sprecherete tempo spostando avanti e indietro pezzi di programma e non vi troverete prigionieri di una struttura, in effetti, poco adatta al vostro programma.

## ***Lo spazio dei nomi***

Quasi tutti i linguaggi di programmazione moderni hanno un ambito di visibilità intermedio tra quello *globale* (accessibile a tutti) e quello *locale* (accessibile solo dalle rispettive funzioni), ma non così JavaScript. Pertanto, tutto quel che deve essere accessibile al di fuori dell'ambito di una funzione di massimo livello è per default visibile *ovunque*.

Di inquinamento dello spazio dei nomi, ossia del problema di brani di codice non collegati tra loro che condividono gli stessi nomi di variabili globali, abbiamo già parlato nel [Capitolo 4](#), dove portavo l'oggetto `Math` come esempio di oggetto che svolge la funzione di un modulo raggruppando funzionalità di tipo matematico.

Sebbene JavaScript non disponga di costrutti per moduli, si possono usare gli oggetti per creare spazi dei nomi separati, accessibili pubblicamente. Si possono inoltre usare le funzioni per creare spazi dei nomi privati, isolati all'interno di un modulo. Più avanti in questo capitolo, illustrerò come realizzare moduli per isolare diversi spazi dei nomi al di sopra dei concetti, piuttosto primitivi, che JavaScript ci mette a disposizione.

## ***Riutilizzo***

In un progetto “piatto”, non strutturato in una serie di moduli, non è chiaro quali parti del codice servano per una particolare funzione. Nel mio programma per spiare i miei nemici ([Capitolo 9](#)), avevo scritto una funzione per leggere i file di configurazione. Se volessi utilizzare quella funzione in un altro progetto, dovrei andare a copiare le parti di quel programma che mi sembrano adatte per la funzionalità che mi interessa e incollarle nel nuovo programma. Dopodiché, se trovassi un errore nel codice, lo correggerei nel programma su cui sto lavorando, ma dimenticherei di andarlo a sistemare nell'altro.

Se avete tanti brani di codice duplicati e condivisi, scoprirete che si perde un sacco di tempo e di energia a spostarli e mantenerli aggiornati.

Raggruppare funzionalità indipendenti in file separati o in moduli semplifica il lavoro di ricerca, aggiornamento e condivisione, perché tutti i brani di codice che vogliono far uso del modulo lo caricano dallo stesso file fisico.

Quest’impostazione diventa ancora più potente quando le relazioni tra moduli, ossia le dipendenze tra un modulo e l’altro, sono definite esplicitamente. In questo modo, potete automatizzare il processo di installare e aggiornare dei moduli (librerie) esterni.

Estendendo ancora il concetto, immaginate un servizio online che raccolga e distribuisca centinaia di migliaia di librerie, e vi permetta di cercare la funzionalità che desiderate e impostare il vostro progetto per scaricarla automaticamente.

Questo servizio esiste e si chiama NPM (<http://npmjs.org>). NPM offre un database di moduli e di strumenti per scaricare e aggiornare i moduli su cui si basano i vostri programmi. Si è sviluppato a partire da Node.js, l’ambiente JavaScript senza browser di cui parleremo nel [Capitolo 20](#), ma vi aiuterà anche quando programmerete per i browser.

## Sdoppiamento

Un altro importante ruolo dei moduli è di isolare tra loro diversi brani di codice, in maniera simile a come operano le interfacce degli oggetti descritte nel [Capitolo 6](#). I moduli meglio progettati offrono un’interfaccia per il codice esterno. Anche se il modulo viene aggiornato per risolvere bachi o aggiungere nuova funzionalità, l’interfaccia rimane la stessa (in quanto è *stabile*) in modo che altri moduli possano usare l’ultima versione senza bisogno di modifiche interne.

Notate che un’interfaccia stabile non implica che non siano aggiunte nuove funzioni, nuovi metodi o nuove variabili: significa semplicemente che non viene eliminata la funzionalità esistente e non ne viene cambiato il significato.

Una valida interfaccia al modulo deve consentirne la crescita, senza interferire con l’interfaccia stessa. Ciò significa che bisogna esporre il meno possibile dei concetti interni del modulo e nello stesso tempo rendere il “linguaggio” esposto dall’interfaccia abbastanza forte e potente da poter essere applicato a una vasta gamma di situazioni.

Nelle interfacce che espongono un solo concetto preciso, come può fare un lettore di file di configurazione, questo tipo di design viene spontaneo. In altri casi, per esempio nel caso di un editor di testi, che ha molti aspetti a cui il codice esterno può richiedere accesso (contenuti, stili, azioni utente e così via), ci vuole una progettazione attenta.

## Le funzioni come spazi dei nomi

Le funzioni sono l’unica cosa di JavaScript che può creare nuovi ambiti di visibilità. Pertanto, se vogliamo che i nostri moduli abbiano ambiti di visibilità propri, dovremo basarli su funzioni.

Prendete questo semplice modulo che associa dei numeri ai nomi dei giorni della settimana, così come potrebbero essere restituiti da un metodo `getDay` dell’oggetto `Date`:

```
var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
             "Thursday", "Friday", "Saturday"];
function dayName(number) {
    return names[number];
}
console.log(dayName(1));
// → Monday
```

---

La funzione `dayName` fa parte dell’interfaccia del modulo, ma non la variabile `names`, in quanto preferiamo non renderla disponibile nell’ambito globale. Possiamo pertanto specificare:

```
var dayName = function() {
    var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday", "Saturday"];
    return function(number) {
        return names[number];
    };
}();
console.log(dayName(3));
// → Wednesday
```

---

Ora, `names` è una variabile locale di una funzione senza nome. Questa funzione viene creata e immediatamente richiamata, e il suo valore di restituzione (la funzione `dayName` stessa) viene memorizzato in una variabile. Questa funzione potrebbe contenere pagine e pagine di codice, con centinaia di variabili locali, e tutto rimarrebbe interno al modulo: visibile all’interno del modulo stesso, ma non al codice esterno.

Possiamo usare uno schema simile per isolare completamente il codice dal mondo esterno. Il seguente modulo registra un valore sulla console, ma non offre valori utilizzabili da altri moduli:

```
(function() {
    function square(x) { return x * x; }
    var hundred = 100;
    console.log(square(hundred));
})();
// → 10000
```

---

Il codice si limita a dare il risultato del quadrato di 100, ma nel mondo reale potrebbe servire come modulo per aggiungere un metodo a un prototipo o impostare un widget su una pagina Web. Il codice è avvolto in una funzione che impedisce alle variabili utilizzate

internamente di inquinare l'ambito di visibilità globale.

Perché abbiamo avvolto la funzione spazio dei nomi tra parentesi? Il motivo sta in un difetto della sintassi di JavaScript. Se *un'espressione* inizia con la parola chiave `function`, si tratta di una funzione. Se però una *dichiarazione* inizia con `function`, si tratta della dichiarazione di una funzione che richiede un nome e, non essendo un'espressione, non può essere richiamata facendola seguire da parentesi. Le parentesi aggiunte servono pertanto per forzare l'interpretazione della funzione come un'espressione.

## Oggetti come interfacce

Immaginate ora di voler aggiungere un'altra funzione al modulo per il giorno della settimana, una che va dal nome del giorno al numero corrispondente. Non possiamo semplicemente restituire la funzione, ma dobbiamo avvolgere le due funzioni in un oggetto.

---

```
var weekDay = function() {
    var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
                 "Thursday", "Friday", "Saturday"];
    return {
        name: function(number) { return names[number]; },
        number: function(name) { return names.indexOf(name); }
    };
}();
console.log(weekDay.name(weekDay.number("Sunday")));
// → Sunday
```

---

Per moduli più grandi, diventa difficile raccogliere tutti i valori *esportati* in un oggetto al termine della funzione, perché le funzioni esportate possono essere molto lunghe e potreste preferire impostarle altrove, vicino al codice interno al quale sono collegate. Una comoda alternativa è di dichiarare un oggetto (che per convenzione si chiama `exports`) e aggiungere a quest'oggetto delle proprietà ogni volta che definiamo qualcosa che va esportato. Nell'esempio seguente, la funzione-modulo accetta come argomento la propria interfaccia e consente al codice al di fuori della funzione di crearla e memorizzarla in una variabile. Ricordate che fuori da una funzione, `this` fa riferimento all'oggetto visibilità globale.

---

```
(function(exports) {
    var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday", "Saturday"];
    exports.name = function(number) {
        return names[number];
    };
})()
```

---

```
};

exports.number = function(name) {
    return names.indexOf(name);
};

})(this.weekDay = {});

console.log(weekDay.name(weekDay.number("Saturday")));
// → Saturday
```

---

## Staccarsi dall'ambito globale

La struttura che abbiamo appena visto si usa di solito nei moduli JavaScript per browser. Il modulo si riserva una sola variabile globale e ne avvolge il codice in una funzione per avere il proprio spazio dei nomi privato. Quest'impostazione, però, continua a dare problemi se c'è più di un modulo che riserva lo stesso nome o se volete caricare contemporaneamente due versioni dello stesso modulo.

Con un po' di lavoro, possiamo creare un sistema dove un modulo può richiedere espressamente l'oggetto interfaccia di un altro modulo senza passare dall'ambito globale. Quel che vogliamo è una funzione `require` che, dato il nome di un modulo, ne carica il file (da disco o dal Web, a seconda della piattaforma di esecuzione) e restituisce il valore di interfaccia corretto.

Quest'impostazione risolve i problemi appena visti e ha il vantaggio aggiuntivo di rendere esplicite le dipendenze del programma, evitando l'uso accidentale di moduli non espressamente richiamati.

Per la funzione `require` servono due cose. Primo, vogliamo una funzione `readFile`, che restituisca come stringa il contenuto del file specificato (in realtà, una funzione di questo tipo non esiste in JavaScript, anche se alcuni ambienti JavaScript, come i browser e Node.js, offrono maniere di accedere ai file; per il momento, facciamo finta che la funzione esista). Secondo, dobbiamo poter eseguire la stringa come codice JavaScript.

## Elaborare i dati come codice

Ci sono molti modi per prendere dei dati (una stringa di codice) ed eseguirli all'interno del programma corrente.

Il modo più ovvio è dato dall'operatore speciale `eval`, che esegue una stringa di codice nell'ambito di visibilità *corrente*. Il che non è una buona idea, perché interferisce con alcune delle proprietà più utili degli ambiti di visibilità, qual è quella di essere isolati dal mondo esterno.

---

```
function evalAndReturnX(code) {
```

```
eval(code);
return x;
}
console.log(evalAndReturnX("var x = 2"));
// → 2
```

---

Un modo migliore per interpretare i dati come codice viene dal costruttore `Function`, che accetta due argomenti: una stringa che contiene un elenco di nomi di argomenti, separati da virgole, e una stringa che contiene il corpo della funzione.

---

```
var plusOne = new Function("n", "return n + 1;");
console.log(plusOne(4));
// → 5
```

---

Il che è esattamente quel che ci vuole per i nostri moduli. Possiamo avvolgere il codice di un modulo in una funzione, in modo che l'ambito di visibilità della funzione diventi l'ambito del modulo.

## La funzione require

Ecco come si può esprimere la forma più elementare di `require`:

---

```
function require(name) {
  var code = new Function("exports", readFile(name));
  var exports = {};
  code(exports);
  return exports;
}
console.log(require("weekDay").name(1));
// → Monday
```

---

Poiché il costruttore `new Function` avvolge il codice del modulo in una funzione, non dobbiamo stare a scrivere una funzione di spazio dei nomi che avvolga il file del modulo stesso. E poiché dichiariamo `exports` come argomento della funzione `module`, non la deve dichiarare il modulo. Questo elimina un bel po' di confusione dal nostro esempio.

---

```
var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
             "Thursday", "Friday", "Saturday"];
exports.name = function(number) {
  return names[number];
};
```

```
exports.number = function(name) {
    return names.indexOf(name);
};
```

---

Con questa struttura, i moduli iniziano di solito con alcune dichiarazioni di variabili che caricano i moduli dai quali dipendono:

---

```
var weekDay = require("weekDay");
var today = require("today");
console.log(weekDay.name(today.dayNumber()));
```

---

L'implementazione semplicistica di `require` che abbiamo dato sopra ha diversi problemi. Intanto, carica ed esegue un modulo ogni volta: pertanto, se ci sono diversi moduli con la stessa dipendenza o se si inserisce una chiamata a `require` all'interno di una funzione che viene richiamata più volte, si sprecano tempo ed energie.

La soluzione sta nel memorizzare in un oggetto i moduli già caricati e restituirne il valore quando uno di essi viene ricaricato più volte.

Il secondo problema è che i moduli non possono esportare direttamente un valore diverso dall'oggetto `exports`, quale sarebbe una funzione. Per esempio, potremmo volere che il modulo esporti solo il costruttore di un oggetto del tipo definito. A questo punto, ciò non è possibile perché `require` utilizza sempre l'oggetto `exports` appena creato come valore di esportazione.

La soluzione tradizionale sta nell'aggiungere ai moduli un'altra variabile, `module`, che è un oggetto con una sua proprietà `exports`. Questa proprietà inizialmente punta all'oggetto vuoto creato da `require`, ma può essere sovrascritta con un diverso valore se vogliamo esportare qualcosa di diverso.

---

```
function require(name) {
    if (name in require.cache)
        return require.cache[name];
    var code = new Function("exports, module", readFile(name));
    var exports = {}, module = {exports: exports};
    code(exports, module);
    require.cache[name] = module.exports;
    return module.exports;
}
require.cache = Object.create(null);
```

---

Abbiamo ora un sistema di moduli con una sola variabile globale (`require`) per consentire ai moduli di trovarsi e usarsi a vicenda senza passare dall'ambito globale.

Questo stile di sistema modulare si chiama *CommonJS Modules*, dal nome della

convenzione pseudo-standard dove venne definito in origine, e fa parte del sistema Node.js. Le sue applicazioni reali fanno molto più di quanto vi ho mostrato in questi esempi. Soprattutto, offrono modi molto più intelligenti per passare dal nome di un modulo a un brano di codice, in quanto consentono di specificare sia percorsi relativi al file corrente, sia nomi di modulo che puntano direttamente ai moduli installati localmente.

## Moduli lenti da caricare

Sebbene sia possibile, usare lo stile dei moduli CommonJS per scrivere per il browser è piuttosto complesso. La ragione è che leggere un file (un modulo) dal Web è molto più lento di quando lo si legge dal disco rigido. Durante l'esecuzione degli script nel browser, nient'altro può succedere al sito relativo, e le ragioni di questo diventeranno evidenti nel [Capitolo 14](#). Ciò significa che, se tutte le chiamate a `require` dovessero recuperare qualcosa da un server remoto, la pagina si bloccherebbe per tempi lunghissimi intanto che il browser carica gli script.

Un modo per aggirare il problema è di provare il vostro codice con un programma come Browserify (<http://browserify.com/>) prima di utilizzarlo in pagine Web. Browserify cerca le chiamate a `require`, risolve le dipendenze e raccoglie il codice necessario in un unico, grande file. Il sito può a questo punto caricare il singolo file invece dei moduli che lo compongono.

Un'altra soluzione è di avvolgere il codice del vostro modulo in una funzione, in modo che la funzione che carica il modulo cominci con le dipendenze prima di richiamare la funzione che inizializza il modulo. Questo è quel che fa il sistema di moduli AMD (Asynchronous Module Definition).

Il nostro semplice programma, con le sue dipendenze, viene letto così in AMD:

---

```
define(["weekDay", "today"], function(weekDay, today) {
  console.log(weekDay.name(today.dayNumber()));
});
```

---

La funzione `define` è il punto cruciale di quest'alternativa. Accetta prima di tutto un array di nomi di moduli, poi una funzione che accetta un argomento per ciascuna dipendenza. Carica poi le dipendenze in background (se già non sono state caricate) in modo che la pagina continui a funzionare intanto che vengono recuperati i file. Una volta caricate tutte le dipendenze, `define` richiama la funzione con le interfacce di quelle dipendenze come argomenti.

I moduli così caricati devono a loro volta contenere una chiamata a `define`. Il valore usato come loro interfaccia è il valore di restituzione della funzione passata a `define`. Ecco un'altra versione del modulo `weekDay`:

---

```
define([], function() {
  var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
```

```

        "Thursday", "Friday", "Saturday"];
    return {
        name: function(number) { return names[number]; },
        number: function(name) { return names.indexOf(name); }
    };
});

```

---

Per potervi mostrare un minimo di funzionalità per `define`, facciamo finta di avere una funzione `backgroundReadFile` che accetta un nome file e una funzione, che richiama col contenuto del file appena ha finito di caricarlo. Nel [Capitolo 17](#) dimostreremo come impostare quella funzione.

Per poterne seguire il caricamento, quest'impostazione di `define` usa oggetti che descrivono lo stato dei moduli, indicando se sono già disponibili; quando ciò avviene, forniscono l'interfaccia dei moduli.

La funzione `getModule` accetta un nome, restituisce l'oggetto corrispondente e fa in modo che il modulo venga messo in coda per essere caricato. Per evitare di caricare due volte lo stesso oggetto, fa uso di un oggetto `cache`.

```

var defineCache = Object.create(null);
var currentMod = null;
function getModule(name) {
    if (name in defineCache)
        return defineCache[name];
    var module = {exports: null,
                  loaded: false,
                  onLoad: []};
    defineCache[name] = module;
    backgroundReadFile(name, function(code) {
        currentMod = module;
        new Function("", code)();
    });
    return module;
}

```

---

Dobbiamo presumere che il file caricato contenga una (e una sola) chiamata per `define`. La variabile `currentMod` serve per indicare alla chiamata l'oggetto modulo che si sta caricando, in modo che possa aggiornare l'oggetto quando il modulo è stato caricato completamente. Torneremo tra poco su questo meccanismo.

La funzione `define` utilizza `getModule` per recuperare o creare gli oggetti modulo per le dipendenze del modulo corrente. Il suo compito è di programmare l'esecuzione della

funzione `moduleFunction` (che contiene il codice del modulo) non appena sono state caricate le rispettive dipendenze. A questo scopo, definisce una funzione `whenDepsLoaded` che viene aggiunta all'array di `onLoad` che fa riferimento alle dipendenze non ancora caricate. Questa funzione restituisce immediatamente se ci sono altre dipendenze da caricare; pertanto, svolgerà il suo lavoro una sola volta, ossia quando l'ultima dipendenza è stata caricata completamente. Questa funzione viene inoltre chiamata immediatamente da `define` quando non ci sono dipendenze da caricare.

---

```
function define(depNames, moduleFunction) {
  var myMod = currentMod;
  var deps = depNames.map(getModule);
  deps.forEach(function(mod) {
    if (!mod.loaded)
      mod.onLoad.push(whenDepsLoaded);
  });
  function whenDepsLoaded() {
    if (!deps.every(function(m) { return m.loaded; }))
      return;
    var args = deps.map(function(m) { return m.exports; });
    var exports = moduleFunction.apply(null, args);
    if (myMod) {
      myMod.exports = exports;
      myMod.loaded = true;
      myMod.onLoad.forEach(function(f) { f(); });
    }
  }
  whenDepsLoaded();
}
```

---

Quando sono disponibili tutte le dipendenze, `whenDepsLoaded` richiama la funzione che contiene il modulo, passandole le interfacce delle dipendenze come argomenti.

La prima cosa che `define` fa è di memorizzare il valore che `currentMod` aveva al momento della chiamata in una variabile `myMod`. Ricorderete che `get-Module`, appena prima di analizzare il codice per un modulo, aveva registrato l'oggetto `module` corrispondente in `currentMod`. In questo modo, `whenDepsLoaded` può registrare il valore di restituzione della funzione modulo nella proprietà `exports` del modulo relativo, impostarne su `true` la proprietà `loaded` e richiamare tutte le funzioni che attendono che il modulo sia caricato.

Questo codice è molto più difficile da leggere della funzione `require`. La sua esecuzione non segue un percorso semplice e prevedibile. Invece, bisogna impostare

diverse operazioni che si svolgeranno in un momento impreciso del futuro, il che oscura l'esecuzione del codice.

Anche in questo caso, le soluzioni AMD sono molto più abili nel risolvere i nomi dei moduli in URL e generalmente più robuste di quanto abbiamo mostrato qui. Il progetto RequireJS (<http://requirejs.org>) offre un buon esempio di questo stile di funzioni di caricamento dei moduli.

## Progettazione delle interfacce

Progettare le interfacce per moduli e tipi di oggetto è uno degli aspetti più delicati della programmazione, in quanto non esiste un criterio univoco per modellare funzionalità meno che triviali. Trovare un sistema che funzioni bene richiede pertanto buone capacità di analisi e previsione dei risultati.

Il modo migliore per imparare l'importanza di una buona progettazione delle interfacce è di usarne tante: alcune saranno buone, altre meno. L'esperienza vi insegnereà che cosa funziona e che cosa no. Non date mai per scontato che un'interfaccia che non funziona sia “inevitabile”. Sistematela, oppure avvolgetela in un'altra che funzioni meglio.

### Prevedibilità

Se siamo in grado di prevedere come funziona una certa interfaccia, non perderemo troppo tempo a cercare di scoprire come la possiamo usare. Ecco perché dovete sforzarvi di seguire delle convenzioni. Quando altri moduli o parti dell'ambiente JavaScript standard svolgono azioni simili a quelle che vi interessano, è una buona idea imitare la struttura dell'interfaccia esistente, in modo che la vostra interfaccia risulti familiare per chi conosce quella esistente.

Un'altra area dove la prevedibilità conta è nel *comportamento* del codice. Magari vi sembra una buona idea sviluppare un'interfaccia inutilmente astrusa, con la giustificazione che sia più comoda; per esempio perché accetta qualunque tipo e combinazione di argomenti e “fa la cosa giusta” per tutti. Oppure, perché dispone di dozzine di funzioni specializzate che offrono molte versioni, di poco diverse tra loro, della funzionalità del modulo. Se è vero che il codice che sfrutta queste interfacce potrebbe essere leggermente più breve, diventa comunque difficile per chi le deve usare costruirsi un chiaro modello mentale del comportamento del modulo.

### Componibilità

Cercate di usare le strutture dati più semplici possibile e impostare le funzioni perché svolgano una sola azione specifica. Se appena potete, impostatele come funzioni pure ([Capitolo 3](#)).

Per esempio, non è raro che i moduli offrano degli oggetti collezione simili ad array, ciascuno con la sua interfaccia per contare ed estrarre elementi. Tali oggetti non avranno metodi `map` o `forEach`: pertanto, eventuali funzioni che si aspettino dei veri array non

potranno funzionare. Questo è un esempio di scarsa *componibilità*, in quanto il modulo non può inserirsi facilmente in altro codice.

Un esempio può essere un modulo per il controllo ortografico, che può servire per impostare un editor di testi. Il modulo di controllo potrebbe operare direttamente su tutte le strutture di dati necessarie per l'editor, non importa quanto complesse, e richiamare direttamente le funzioni interne dell'editor per offrire a chi scrive delle alternative corrette. Se seguiamo l'impostazione appena descritta, non potremo usare lo stesso modulo con nessun altro programma. Se invece definiamo l'interfaccia di controllo ortografico in modo che accetti una semplice stringa e restituisca la posizione nella stringa dove si trova un eventuale errore, insieme a un array di possibili correzioni, allora avremo un'interfaccia componibile con altri sistemi, in quanto stringhe e array sono sempre disponibili in JavaScript.

## **Interfacce stratificate**

Succede abbastanza spesso di incorrere in un dilemma quando si programmano interfacce per funzionalità complesse; per esempio, inviare email. Da un lato, non possiamo tediare l'utente con troppi dettagli: non vogliamo che perda 20 minuti per studiare l'interfaccia prima di poter spedire un messaggio. D'altro canto, non si possono nemmeno nascondere tutti i dettagli, in quanto gli utenti devono essere liberi di usare il modulo anche per gestire situazioni più complesse.

Spesso la soluzione sta nel fornire due interfacce: una dettagliata, di *basso livello*, per le situazioni più complesse e una semplice, di *alto livello*, per le operazioni di routine. Di solito, la seconda si costruisce con gli strumenti offerti dalla prima. Nel modulo per la posta elettronica, l'interfaccia di alto livello può essere semplicemente una funzione che accetta un messaggio, un indirizzo per il mittente e uno per il destinatario, e invia l'email. L'interfaccia di basso livello fornirà il controllo completo sugli header dell'email, gli allegati, la formattazione HTML e così via.

## **Riepilogo**

I moduli offrono struttura ai programmi più grandi separando il codice in file e spazi dei nomi distinti. Dotare i moduli di interfacce ben definite li rende più semplici da usare e riutilizzare, mantenendone la stessa utilità non importa come e quanto si evolvano.

Sebbene JavaScript offra ben poco in quest'ambito, la flessibilità delle sue funzioni e dei suoi oggetti consente di definire dei sistemi di moduli piuttosto sofisticati. Gli ambiti di visibilità delle funzioni si possono sfruttare nei moduli come spazi dei nomi interni, memorizzando in oggetti gli insiemi di valori esportati.

Le due soluzioni (ben definite) più popolari per questo tipo di moduli sono *CommonJS Modules* e *AMD*. La prima si sviluppa intorno a una funzione `require`, che recupera un modulo per nome e ne restituisce l'interfaccia. AMD si basa invece su una funzione `define`, che accetta un array di nomi di moduli e una funzione, e, dopo aver caricato i

moduli, esegue la funzione con le rispettive interfacce come argomenti.

## Esercizi

### *Nomi dei mesi*

Scrivete un semplice modulo, simile a `weekday`, che converta i numeri dei mesi (a partire da zero, come nel tipo `data`) in nomi e riconverte i nomi in numeri. Assegnategli un suo spazio dei nomi, perché ci vuole un array interno di nomi dei mesi, e usate codice JavaScript semplice, senza far conto su sistemi di caricamento di moduli esterni.

### *Ritorno alla vita elettronica*

Facciamo conto che abbiate ancora ben presente il contenuto del [Capitolo 7](#). Pensate al sistema definito in quel capitolo e trovate un modo per separare il codice in moduli. Per rinfrescarvi la memoria, ecco l'elenco delle funzioni e dei tipi definiti in quel capitolo, in ordine di apparizione:

1. `Vector`
2. `Grid`
3. `directions`
4. `randomElement`
5. `BouncingCritter`
6. `elementFromChar`
7. `World`
8. `charFromElement`
9. `Wall`
10. `View`
11. `directionNames`
12. `WallFollower`
13. `dirPlus`
14. `LifelikeWorld`
15. `Plant`
16. `PlantEater`
17. `SmartPlantEater`
18. `Tiger`

Non esagerate però nel creare troppi moduli. Un libro che apre un nuovo capitolo a

ogni pagina non piace a nessuno, non fosse altro che per lo spreco di spazio per i titoli. Lo stesso vale se dovessimo aprire 10 file per leggere un progetto piccolino: puntate a tre, cinque moduli al massimo.

Potete scegliere di rendere alcune funzioni interne al rispettivo modulo e pertanto inaccessibili dagli altri.

Questo esercizio non ha una sola risposta giusta: l'organizzazione dei moduli è soprattutto questione di gusti.

## **Dipendenze circolari**

Un aspetto delicato nella gestione delle dipendenze sono le dipendenze circolari, dove il modulo A dipende da B, che a sua volta dipende da A. Molti sistemi di moduli proibiscono questo comportamento. I moduli CommonJS ne consentono una forma limitata: le dipendenze circolari funzionano solo se i moduli non sostituiscono i rispettivi oggetti exports predefiniti con valori diversi e non tentano di accedere alle relative interfacce solo dopo essere stati caricati completamente.

Pensate a come si potrebbe realizzare questa funzionalità. Tornate a guardare la definizione di require e pensate a che cosa si deve fare per consentire questo comportamento.

# 11

## PROGETTO: UN LINGUAGGIO DI PROGRAMMAZIONE

*L'interprete, che stabilisce il significato delle espressioni in un linguaggio di programmazione, è solo un altro programma.*

Hal Abelson e Gerald Sussman,  
*Structure and Interpretation of Computer Programs*

Realizzare un linguaggio di programmazione è sorprendentemente facile (a condizione di non puntare troppo in alto) e di grande soddisfazione.

La cosa più importante che voglio dimostrare in questo capitolo è che non c'è nulla di magico in questo. Ho spesso pensato che ci sono invenzioni talmente intelligenti e complicate, che non sarei mai stato in grado di capirle. Con qualche lettura e un po' di prove, però, spesso si scopre che le cose sono in realtà piuttosto semplici.

Qui imposteremo un linguaggio di programmazione chiamato Egg. Sarà un linguaggio semplice e limitato, eppure abbastanza potente da esprimere qualunque calcolo vi venga in mente, e consentirà delle semplici astrazioni basate su funzioni.

### L'analisi

La parte immediatamente più visibile di un linguaggio di programmazione è la sua *sintassi*, o notazione. Il *parser* (l'analizzatore) è un programma che legge un brano di testo e produce una struttura dati che riflette la struttura del programma contenuto in quel brano. Se il testo non forma un programma valido, il parser dovrebbe segnalare un errore.

Il nostro linguaggio avrà una sintassi semplice e uniforme. In Egg, tutto è un'espressione. Un'espressione può essere una variabile, un numero, una stringa o un'applicazione. Le applicazioni si usano per le chiamate di funzione, ma anche per costrutti come `if` o `while`.

Per semplificare il parser, le stringhe non offrono supporto a caratteri come le barre rovesciate: sono solo sequenze di caratteri, che non siano virgolette doppie, racchiuse tra virgolette doppie. I numeri sono sequenze di cifre. I nomi di variabile possono usare qualsiasi carattere che non sia spazio vuoto e non abbia un significato sintattico speciale.

Le applicazioni si impostano come in JavaScript, aprendo parentesi dopo un'espressione e inserendo tra parentesi un numero indefinito di argomenti, separati da virgole.

```
do(define(x, 10),
  if(>(x, 5),
    print("large"),
    print("small")))
```

---

L'uniformità di Egg significa che quel che in JavaScript è un operatore (come `>`) è una variabile normale in questo linguaggio e si applica come tutte le altre funzioni. Poiché la sintassi non prevede il concetto di blocco, dobbiamo usare un costrutto `do` per rappresentare azioni sequenziali.

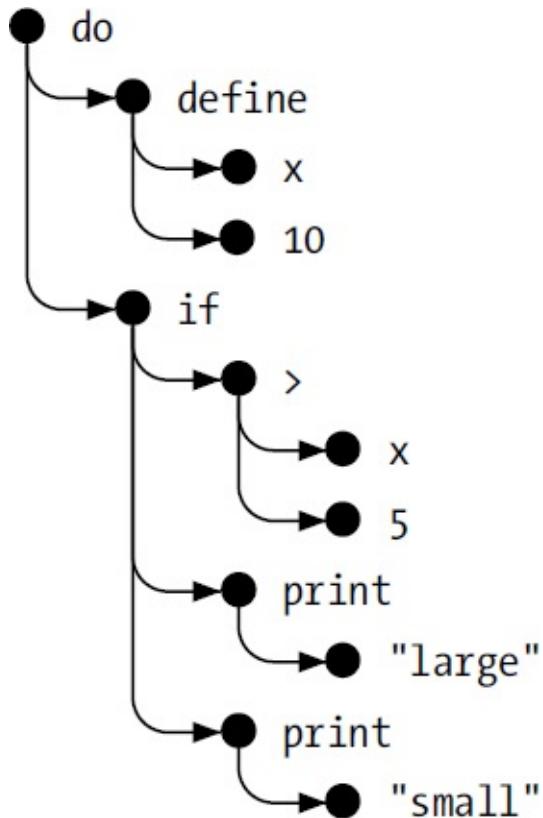
Per descrivere un programma, il parser userà una struttura dati che consisterà di oggetti espressione, ciascuno con una proprietà `type` che indica il tipo della relativa espressione e altre proprietà per descriverne il contenuto.

Le espressioni di tipo "value" rappresentano stringhe di lettere o numeri. La loro proprietà `value` contiene la stringa o il numero che rappresentano. Le espressioni di tipo "word" sono utilizzate per gli identificatori (nomi). Questi oggetti hanno una proprietà `name` che contiene il nome dell'identificatore sotto forma di stringa. Infine, le espressioni di tipo "apply" rappresentano le applicazioni. Hanno una proprietà `operator`, che fa riferimento all'espressione che viene applicata, e una proprietà `args` che fa riferimento a un array di espressioni argomento. La parte `>(x, 5)` del programma precedente viene pertanto rappresentata come segue:

```
{
  type: "apply",
  operator: {type: "word", name: ">"},
  args: [
    {type: "word", name: "x"},
    {type: "value", value: 5}
  ]
}
```

---

Questa struttura di dati si chiama *albero sintattico*. Se immaginate gli oggetti come punti e i collegamenti tra essi come righe che collegano i punti, la struttura ha la forma di un albero. Il fatto che le espressioni contengano altre espressioni, che a loro volta possono contenere più espressioni, ricorda le divisioni dei rami.



Confrontate questa rappresentazione con il parser che avevamo impostato per il formato del file di configurazione nel [Capitolo 9](#), che aveva una struttura semplice: suddivideva l'input in righe e le manipolava una alla volta, e c'erano solo poche forme semplici che ogni riga poteva avere.

Qui dobbiamo trovare un'impostazione diversa. Le espressioni non sono divise in righe e hanno una struttura ricorsiva: le espressioni di applicazione ne contengono altre.

Fortunatamente, questo problema si risolve elegantemente impostando una funzione ricorsiva, che rifletta la natura ricorsiva del linguaggio.

Definiamo una funzione `parseExpression`, che accetta una stringa e restituisce un oggetto contenente la struttura dati per l'espressione all'inizio della stringa, insieme alla parte della stringa che resta dopo aver analizzato l'espressione. Si può richiamare di nuovo questa funzione per analizzare le sottoespressioni (per esempio, l'argomento di un'applicazione) e ottenere l'espressione argomento insieme al testo restante. Questo testo può a sua volta contenere altri argomenti oppure essere la parentesi che conclude l'elenco degli argomenti.

Ecco la prima parte del parser:

---

```

function parseExpression(program) {
  program = skipSpace(program);
  var match, expr;
  if (match = /^"([^"]*)"/.exec(program))
    expr = {type: "value", value: match[1]};
  else if (match = /\d+\b/.exec(program))
    expr = {type: "value", value: Number(match[0])};
  
```

```

else if (match = /^[^\s(),"]+/.exec(program))
  expr = {type: "word", name: match[0]};
else
  throw new SyntaxError("Unexpected syntax: " + program);
return parseApply(expr, program.slice(match[0].length));
}

function skipSpace(string) {
  var first = string.search(/\S/);
  if (first == -1) return "";
  return string.slice(first);
}

```

---

Poiché Egg non pone limiti allo spazio vuoto tra elementi, dobbiamo continuamente eliminare gli spazi all'inizio della stringa del programma: ecco dove interviene la funzione `skipSpace`.

Dopo aver saltato lo spazio vuoto iniziale, `parseExpression` usa tre espressioni regolari per trovare i tre semplici (atomici) elementi che Egg supporta: stringhe, numeri e parole. Il parser costruisce una diversa struttura dati a seconda di quel che trova. Se l'input non corrisponde a nessuno degli elementi, allora non è un'espressione valida e fa scattare un errore. `SyntaxError` è un tipo standard di oggetto errore, che scatta quando si tenta di eseguire un programma JavaScript non valido.

Possiamo quindi eliminare la parte del testo che ha trovato corrispondenza e passarla, insieme all'oggetto per l'espressione, a `parseApply`, che verifica se l'espressione è un'applicazione. In questo caso, analizza un elenco di argomenti tra parentesi.

---

```

function parseApply(expr, program) {
  program = skipSpace(program);
  if (program[0] != "(")
    return {expr: expr, rest: program};
  program = skipSpace(program.slice(1));
  expr = {type: "apply", operator: expr, args: []};
  while (program[0] != ")") {
    var arg = parseExpression(program);
    expr.args.push(arg.expr);
    program = skipSpace(arg.rest);
    if (program[0] == ",")
      program = skipSpace(program.slice(1));
    else if (program[0] != ")")
      throw new SyntaxError("Expected ',' or ')'");
  }
}
```

```
}

return parseApply(expr, program.slice(1));
}
```

---

Se il carattere che segue non è una parentesi aperta, non si tratta di un'applicazione e `parseApply` restituisce semplicemente l'espressione data.

Altrimenti, salta la parentesi e crea l'oggetto albero sintattico per quest'applicazione. Richiama quindi ricorsivamente `parseExpression` per analizzare ciascuno degli argomenti, finché non trova la parentesi chiusa. La ricorsività è indiretta, in quanto `parseApply` e `parseExpression` si richiamano l'un l'altro.

Poiché si può applicare anche un'espressione applicazione (come in `multiplier(2)(1)`), `parseApply`, dopo aver analizzato un'applicazione, deve richiamarsi di nuovo per verificare se trova un'altra coppia di parentesi.

Questo è tutto quel che serve per analizzare Egg. Avvolgiamo il codice in una funzione `parse`, che verifica di aver raggiunto la fine della stringa di input dopo aver analizzato l'espressione (un programma Egg è una sola espressione) e ci dà la struttura dati del programma.

---

```
function parse(program) {
  var result = parseExpression(program);
  if (skipSpace(result.rest).length > 0)
    throw new SyntaxError("Unexpected text after program");
  return result.expr;
}

console.log(parse("+ (a, 10)"));
// → {type: "apply",
//   operator: {type: "word", name: "+"},
//   args: [{type: "word", name: "a"},
//           {type: "value", value: 10}]} 
```

---

Funziona! Non offre informazioni molto utili quando le cose non funzionano e non registra la riga e la colonna dove iniziano le singole espressioni, cosa che servirà più avanti quando si tratta di lanciare degli errori, ma è un buon inizio.

## L'interprete

Che cosa possiamo fare con l'albero sintattico di un programma? Ma eseguirlo, naturalmente! E questo è quel che fa l'interprete: gli date un albero sintattico e un oggetto ambiente che associa nomi e valori, e l'interprete analizzerà l'espressione che rappresenta l'albero e restituirà il valore che essa produce.

---

```
function evaluate(expr, env) {
  switch(expr.type) {
    case "value":
      return expr.value;
    case "word":
      if (expr.name in env)
        return env[expr.name];
      else
        throw new ReferenceError("Undefined variable: " +
                                  expr.name);
    case "apply":
      if (expr.operator.type == "word" &&
          expr.operator.name in specialForms)
        return specialForms[expr.operator.name](expr.args, env);
      var op = evaluate(expr.operator, env);
      if (typeof op != "function")
        throw new TypeError("Applying a non-function.");
      return op.apply(null, expr.args.map(function(arg) {
        return evaluate(arg, env);
      }));
  }
}

var specialForms = Object.create(null);
```

---

L'interprete contiene codice per tutti i tipi di espressione. Un'espressione di valore letterale ne elabora semplicemente il valore (per esempio, l'espressione 100 si risolve nel numero 100). Per le variabili, dobbiamo prima verificare se sono definite nell'ambiente e, se così è, recuperarne il valore.

Le applicazioni sono più complicate. Se sono di un modello (form) speciale, come if, non le elaboriamo ma ci limitiamo a passare le espressioni argomento, insieme all'ambiente, alla funzione che le gestisce. Se sono chiamate normali, calcoliamo l'operatore, verifichiamo che sia una funzione e la richiamiamo col risultato del calcolo degli argomenti.

Useremo semplici valori di funzioni JavaScript per rappresentare i valori delle funzioni di Egg. Ci torneremo sopra quando definiamo il modello speciale fun.

La struttura ricorsiva di evaluate ricorda la struttura del parser, ed entrambi rispecchiano la struttura del linguaggio. Sarebbe anche possibile integrare il parser con l'interprete ed effettuare i calcoli durante l'analisi, ma dividere i due aspetti rende il

programma più facile da leggere.

Questo è tutto quel che serve per interpretare Egg: è davvero semplice. Ma se non definiamo alcuni modelli speciali e aggiungiamo valori significativi all'ambiente, non possiamo ancora far nulla.

## Modelli speciali

L'oggetto `specialForms` definisce la sintassi speciale in Egg, associando parole a funzioni che calcolano i modelli speciali. Per il momento è vuoto: aggiungiamo ora alcuni modelli.

---

```
specialForms["if"] = function(args, env) {
  if (args.length != 3)
    throw new SyntaxError("Bad number of args to if");
  if (evaluate(args[0], env) !== false)
    return evaluate(args[1], env);
  else
    return evaluate(args[2], env);
};
```

Il costrutto `if` di Egg accetta esattamente tre argomenti. Calcola il primo e, se il risultato non è il valore `false`, calcola il secondo. Altrimenti, calcola il terzo. Questo modello `if` è più simile all'operatore ternario `?:` che all'`if` di JavaScript. È un'espressione, non una dichiarazione, e produce un valore, e precisamente il risultato del secondo o del terzo argomento.

Egg si distingue da JavaScript nel modo in cui gestisce il valore condizionale di `if`. Non tratta come `false` risultati come lo zero o le stringhe vuote, ma ne ricava semplicemente il valore esatto `false`.

La ragione per cui dobbiamo rappresentarlo come un modello speciale, invece di una funzione normale, è che tutti gli argomenti delle funzioni sono calcolati prima della chiamata alla funzione; `if`, invece, deve svolgere i calcoli solo sul secondo o sul terzo argomento, a seconda del valore del primo.

La forma `while` è simile.

---

```
specialForms["while"] = function(args, env) {
  if (args.length != 2)
    throw new SyntaxError("Bad number of args to while");
  while (evaluate(args[0], env) !== false)
    evaluate(args[1], env);
  // Since undefined does not exist in Egg, we return false,
  // for lack of a meaningful result.
```

```
    return false;  
};
```

---

Un altro componente fondamentale è `do`, che esegue tutti i suoi argomenti da cima a fondo. Il suo valore è quello prodotto dall'ultimo argomento:

```
specialForms["do"] = function(args, env) {  
    var value = false;  
    args.forEach(function(arg) {  
        value = evaluate(arg, env);  
    });  
    return value;  
};
```

---

Per poter creare variabili e dar loro nuovi valori, creiamo anche un modello `define`, che accetta un valore `word` come primo argomento e un'espressione che produce il valore da assegnare a quella parola come secondo argomento. Poiché `define`, come tutto il resto, è un'espressione, deve restituire un valore. La impostiamo in modo che restituisca il valore assegnato (così come fa l'operatore `=` di JavaScript).

```
specialForms["define"] = function(args, env) {  
    if (args.length != 2 || args[0].type != "word")  
        throw new SyntaxError("Bad use of define");  
    var value = evaluate(args[1], env);  
    env[args[0].name] = value;  
    return value;  
};
```

---

## L'ambiente

L'ambiente per `evaluate` è un oggetto con proprietà i cui nomi corrispondono a nomi di variabile e i cui valori corrispondono ai valori legati a quelle variabili. Definiamo ora un oggetto ambiente che rappresenti l'ambito globale.

Per poter usare il costrutto `if` che abbiamo definito sopra, dobbiamo avere accesso a valori booleani. Poiché i valori booleani sono solo due, non ci serve una sintassi speciale: basta legare due variabili ai valori `true` e `false` e usare quelle variabili.

```
var topEnv = Object.create(null);  
topEnv["true"] = true;  
topEnv["false"] = false;
```

---

Possiamo ora usare [evaluate] per calcolare una semplice espressione che nega un valore booleano.

---

```
var prog = parse("if(true, false, true)");
console.log(evaluate(prog, topEnv));
// → false
```

---

Aggiungeremo inoltre all'ambiente alcuni valori di funzione per dei semplici operatori aritmetici e di confronto. Per non allungare il codice inutilmente, invece di definirle singolarmente usiamo new Function per sintetizzare in un ciclo una serie di funzioni operatore.

---

```
[ "+", "-", "*", "/", "==", "<", ">"].forEach(function(op) {
  topEnv[op] = new Function("a, b", "return a " + op + " b;");
});
```

---

Ci serve anche un modo per stampare dei valori, pertanto avvolgiamo console.log in una funzione, che chiamiamo print.

---

```
topEnv["print"] = function(value) {
  console.log(value);
  return value;
};
```

---

Con questo, abbiamo abbastanza strumenti (elementari) per scrivere dei programmi semplici. La seguente funzione run offre una strada comoda per scriverli ed eseguirli: crea un nuovo ambiente e quindi analizza e interpreta le stringhe che passiamo.

---

```
function run() {
  var env = Object.create(topEnv);
  var program = Array.prototype.slice.call(arguments, 0).join("\n");
  return evaluate(parse(program), env);
}
```

---

Array.prototype.slice.call è una scorciatoia per trasformare un oggetto simile a un array, come un elenco di argomenti, in un vero array sul quale possiamo richiamare join. Accetta tutti gli argomenti passati a run e li tratta come se fossero righe di un programma.

---

```
run("do(define(total, 0),",
  "  define(count, 1),",
  "  while(<(count, 11),",
    "    do(define(total, +(total, count)),",
```

```
"           define(count, +(count, 1))),",
"     print(total))");
// → 55
```

---

Questo è il programma che abbiamo già visto altre volte, che calcola la somma dei numeri da 1 a 10, espresso in Egg. È ovviamente più brutto del programma in JavaScript equivalente, ma niente male per un programma che richiede meno di 150 righe di codice.

## Funzioni

Un linguaggio di programmazione senza funzioni non è un granché. Per fortuna, non è difficile aggiungere un costrutto `fun`, che tratta il suo ultimo argomento come corpo della funzione e gli argomenti che lo precedono come nomi degli argomenti della funzione.

---

```
specialForms["fun"] = function(args, env) {
  if (!args.length)
    throw new SyntaxError("Functions need a body");
  function name(expr) {
    if (expr.type != "word")
      throw new SyntaxError("Arg names must be words");
    return expr.name;
  }
  var argNames = args.slice(0, args.length - 1).map(name);
  var body = args[args.length - 1];
  return function() {
    if (arguments.length != argNames.length)
      throw new TypeError("Wrong number of arguments");
    var localEnv = Object.create(env);
    for (var i = 0; i < arguments.length; i++)
      localEnv[argNames[i]] = arguments[i];
    return evaluate(body, localEnv);
  };
};
```

---

In Egg, le funzioni hanno un ambiente locale proprio, come in JavaScript. Usiamo `Object.create` per costruire l'oggetto che ha accesso alle variabili nell'ambiente esterno (il suo prototipo), ma che può anche contenere nuove variabili senza modificare l'ambito di visibilità esterno.

La funzione creata dal modello `fun` costruisce l'ambiente locale e vi aggiunge le variabili argomento. Calcola poi il corpo della funzione in questo ambiente e restituisce il

risultato.

---

```
run("do(define(plusOne, fun(a, +(a, 1))),",
     "  print(plusOne(10))))";
// → 11
run("do(define(pow, fun(base, exp,",
     "  if(==(exp, 0),",
     "    1,",
     "    *(base, pow(base, -(exp, 1)))))),",
     "  print(pow(2, 10))))";
// → 1024
```

---

## Compilazione

Quel che abbiamo realizzato fin qui è un interprete che, durante l’elaborazione, interviene direttamente sulla rappresentazione del programma prodotta dal parser.

La *compilazione* è un processo che aggiunge un’altra fase tra l’analisi e l’esecuzione del programma, trasformandolo in modo che possa essere elaborato il più velocemente possibile, in quanto parte del lavoro viene svolto in anticipo. Per esempio, nei linguaggi ben progettati i riferimenti sono chiari ogni volta che si incontra una variabile, senza bisogno di eseguire il programma. In questo modo, si evita di cercare la variabile per nome ogni volta che vi si accede, recuperandola invece da una posizione in memoria predeterminata.

Tradizionalmente, la compilazione richiede la conversione del programma in codice macchina, il formato grezzo eseguibile dal processore del computer. Si può pensare, comunque, che tutti i processi che convertono un programma in una rappresentazione diversa siano processi di compilazione.

Potremmo scrivere una strategia di elaborazione alternativa per Egg, dove prima si converte il programma in una sua versione JavaScript, poi si usa `new Function` per invocare il compilatore JavaScript e si esegue poi il risultato. Se impostato correttamente, questo sistema renderebbe Egg molto veloce in esecuzione, pur nella sua semplicità di impostazione.

Se vi interessa l’argomento e volete perderci un po’ di tempo, vi invito a provare a realizzare questo compilatore come esercizio pratico.

## Imbrogli

Avrete notato che le definizioni di `if` e `while` erano semplicemente avvolte intorno ai costrutti `if` e `while` propri di JavaScript. Analogamente, i valori in Egg non sono altro che

valori di JavaScript.

Se confrontate la realizzazione di Egg, che si basa su JavaScript, con la quantità di lavoro e le complessità necessarie per costruire un linguaggio di programmazione in grado di comunicare direttamente con le funzionalità grezze del linguaggio macchina, troverete che la differenza è enorme. Ciononostante, mi auguro che questo esempio vi abbia dato una buona idea di come funzionano i linguaggi di programmazione.

E se si tratta di concludere qualcosa, allora “imbrogliare” ha più efficacia che fare tutto da soli. Sebbene il linguaggio giocattolo di questo capitolo non faccia nulla di più né meglio di quel che può fare JavaScript, in alcune situazioni scrivere dei piccoli linguaggi può aiutare molto.

Questi piccoli linguaggi non devono necessariamente ricordare il tipico linguaggio di programmazione. Se JavaScript non dispone già di espressioni regolari, potremmo scrivere un parser e un interprete per quel tipo di operazioni. Oppure, immaginate di dover costruire un gigantesco robot-dinosauro e programmare il suo comportamento. JavaScript non sarebbe il sistema migliore per farlo.

Potreste, invece, optare per un linguaggio che assomigli al seguente:

---

comportamento cammina

svolgi quando

vede destinazione

azioni

sposta piede sinistro

sposta piede destro

comportamento attacca

svolgi quando

vede Godzilla

azioni

occhiate lancialaser

lancia razzi da braccio

---

Questo è quel che si chiama un *linguaggio specifico per il dominio*, un linguaggio tagliato su misura per esprimere un dominio di conoscenza ristretto. Tali linguaggi possono essere più espressivi di quelli generali, proprio perché sono impostati per esprimere esattamente quel che serve in quel dominio e null'altro.

## Esercizi

### Array

Date a Egg il supporto per gli array, aggiungendo le tre funzioni seguenti all'ambito

superiore: `array(...)` per costruire un array contenente i valori degli argomenti, `length(array)` per recuperare la lunghezza di un array ed `element(array)` per recuperare l'elemento in posizione `n`.

## Chiusura

Così come l'abbiamo definito, `fun` consente alle funzioni di Egg di “chiudersi” all’ambiente che le circonda, consentendo al corpo della funzione di usare valori locali che erano visibili quando abbiamo definito la funzione, proprio come fanno le funzioni in JavaScript.

Il programma che segue illustra questo concetto: la funzione `f` restituisce una funzione che aggiunge il proprio argomento a quello di `f`, il che significa che deve poter accedere all’ambito locale di `f` per poter usare la variabile `a`.

---

```
run("do(define(f, fun(a, fun(b, +(a, b)))),",
      "  print(f(4)(5)))");
// → 9
```

---

Tornate alla definizione del modello `fun` e spiegate quale meccanismo determina il funzionamento di questo brano di codice.

## Commenti

Sarebbe bello poter scrivere dei commenti in Egg. Per esempio, quando incontriamo un segno `#` potremmo trattare il resto della riga come un commento e ignorarlo, come facciamo con `//` in JavaScript.

Non servono grandi modifiche al parser per ottenerlo. Basta modificare `skipSpace` in modo che salti i commenti come se fossero spazio vuoto, in modo che tutte le posizioni dove si richiama `skipSpace` saltino anche i commenti. Modificate il parser con questo obiettivo.

## Sistemare l’ambito di visibilità

Attualmente, l’unico modo per assegnare un valore a una variabile è `define`. Questo costrutto consente sia di definire nuove variabili, sia di assegnare un valore a quelle esistenti.

Quest’ambiguità crea un problema. Quando cercate di assegnare un nuovo valore a una variabile non locale, finite col definirne invece una locale con lo stesso nome (alcuni linguaggi funzionano così per definizione, ma mi sembra proprio un modo sciocco di gestire l’ambito di visibilità).

Aggiungete un modello speciale, `set`, simile a `define`, che assegna un nuovo valore a una variabile, aggiornandola in un ambito esterno se già non esiste nell’ambito interno. Se la variabile non è stata definita, fate in modo che venga lanciato un errore `ReferenceError` (che è un altro tipo di errore standard).

La tecnica di rappresentare gli ambiti di visibilità come semplici oggetti, che ci è tornata tanto comoda finora, a questo punto vi darà un po' fastidio. Potete provare a usare la funzione `Object.getPrototypeOf`, che restituisce il prototipo di un oggetto. Ricordatevi inoltre che, dal momento che gli ambiti di visibilità non derivano da `Object.prototype`, se volette richiamare su di essi `hasOwnProperty` dovrete usare la seguente espressione:

---

```
Object.prototype.hasOwnProperty.call(scope, name);
```

---

Ciò recupera il metodo `hasOwnProperty` dal prototipo di `Object` e quindi lo richiama su un oggetto `scope`.

# **Parte II**

## **IL BROWSER**

## JAVASCRIPT E IL BROWSER

*Il browser è un ambiente di programmazione davvero ostile.*

Douglas Crockford, *The JavaScript Programming Language*  
(videoconferenza)

In questa parte del libro parliamo di Web browser. Senza browser, non ci sarebbe JavaScript. E anche se ci fosse stato, nessuno ci avrebbe fatto caso.

Fin dall'inizio, la tecnologia del Web è stata decentralizzata, non solo tecnicamente, ma anche nella sua evoluzione. I produttori di browser per il Web hanno aggiunto funzionalità specifiche, a volte anche mal progettate, che in alcuni casi sono state adottate da altri e hanno finito col diventare degli standard.

Il che è nello stesso tempo un bene e una condanna. Da una parte, è fantastico che il sistema non sia controllato da entità centralizzate, ma possa essere migliorato da chiunque voglia lavorare in forme di collaborazione informali (che a volte possono tramutarsi in ostilità aperte). Dall'altro, il modo assolutamente casuale in cui si sviluppa il Web significa che il sistema su cui poggia non è esattamente un magnifico esempio di coerenza interna. In effetti, alcune sue parti sono decisamente confuse e sconcertanti.

## Le reti e Internet

Le reti di computer esistono fin dagli anni Cinquanta. Se montate dei cavi tra due o più computer e li mettete in condizione di scambiare dati attraverso quei cavi, potrete fare un sacco di cose meravigliose.

E se collegare tra loro due macchine nello stesso palazzo ci permette di fare cose meravigliose, collegare macchine distribuite in tutto il pianeta dovrebbe essere ancora meglio. La tecnologia che aprì il campo a questa visione venne sviluppata negli anni Ottanta e la rete che ne derivò si chiama *Internet*, e ha ampiamente soddisfatto le aspettative.

Un computer può usare Internet per mandare bit a un altro computer. Perché da questi invii di bit possano nascere delle comunicazioni valide, i computer collegati devono sapere che cosa rappresentano i vari bit. Il significato di ogni sequenza di bit dipende esclusivamente dal tipo di comunicazione che esprime e dal meccanismo di codifica utilizzato.

I protocolli di rete descrivono lo stile delle comunicazioni in rete. Esistono protocolli per inviare o scaricare email, condividere file e persino per controllare computer infetti da

software dannoso.

Per esempio, un semplice protocollo di chat potrebbe consistere in una sequenza dove un computer invia i bit che rappresentano il testo “CHAT?” a un altro computer, che a sua volta risponde “OK!” per confermare di capire il protocollo. I due computer possono ora scambiarsi stringhe di testo, leggere il testo inviato dall’altro computer e visualizzare quel che ricevono sui rispettivi schermi.

Quasi tutti i protocolli sono costruiti sopra altri protocolli. Il nostro esempio di protocollo per chat tratta la rete come un dispositivo di flusso, dove si immettono bit che arrivano alla giusta destinazione nell’ordine prestabilito. Ma fare in modo che ciò accada è di per sé un problema tecnico piuttosto complesso.

Il *protocollo TCP* (Transmission Control Protocol) risolve questo problema. Tutti i dispositivi connessi a Internet lo “parlano” e quasi tutte le comunicazioni che avvengono su Internet sono costruite a partire da esso.

Ecco come funziona una connessione TCP: un computer deve essere in attesa, o *in ascolto*, di qualunque altro computer voglia cominciare a parlargli. Per poter ascoltare contemporaneamente diversi tipi di comunicazioni, il computer in ascolto ha un numero (una *porta*) associato con ciascuno di essi. In generale, i protocolli specificano un numero di porta predefinito. Per esempio, se vogliamo mandare un messaggio email con protocollo SMTP, la macchina attraverso la quale passiamo deve essere in ascolto sulla porta 25.

Un altro computer può quindi stabilire una connessione con la macchina di destinazione usando il numero di porta corretto. Se la macchina di destinazione è raggiungibile ed è in ascolto su quella porta, si stabilisce la connessione. Il computer in ascolto si chiama *server* e il computer che si collega al server si chiama *client*.

Questo tipo di connessione funziona come un condotto a due sensi, nel quale passa il flusso di bit: le macchine alle due estremità possono immettervi dati. Una volta trasmessi, i bit possono essere riletti dalla macchina all’altra estremità del condotto. Questo è un buon modello e si può dire che il protocollo TCP offre un’astrazione della rete.

## Il Web

Il *World Wide Web* (da non confondere con Internet nella sua interezza) è una serie di protocolli e di formati che ci permettono di visitare delle pagine Web in un browser. La parte “Web” (ragnatela) si riferisce al fatto che tali pagine possono facilmente collegarsi l’una all’altra, realizzando così un’immensa ragnatela attraverso la quale ci si può muovere.

Per aggiungere contenuti al Web, basta collegare una macchina a Internet e impostarla in modo che rimanga in ascolto sulla porta 80 attraverso il *protocollo HTTP* (Hypertext Transfer Protocol). Questo protocollo consente ad altri computer di ricevere documenti in rete.

Ogni documento sul Web ha un nome assegnato da un *URL* (Universal Resource Locator), che ha un formato come questo:

---

`http://eloquentjavascript.net/12_browser.html`

| | | |  
protocollo server percorso

---

La prima parte ci dice che questo URL usa il protocollo HTTP (invece, per esempio, del protocollo HTTP crittografato, che viene indicato con *https://*). Segue la parte che identifica il server al quale stiamo richiedendo il documento. Infine, l'ultima stringa identifica il documento (o la risorsa) che ci interessa.

Ciascuna delle macchine connesse a Internet riceve un *indirizzo IP* univoco, che ha un formato come `37.187.37.82`.

Potete usare l'indirizzo IP direttamente per la parte server dell'URL. Poiché è difficile tenere a mente e digitare lunghe file di numeri casuali, potete invece registrare un *nome dominio* in modo che punti a una determinata macchina o gruppo di macchine. Io ho registrato *eloquentjavascript.net* in modo che punti all'indirizzo IP di una macchina su cui ho il controllo e posso pertanto usare quel nome dominio per servire pagine Web.

Se digitate questo URL nella barra degli indirizzi del vostro browser, il browser tenta di recuperare e visualizzare il documento che si trova a quell'URL. Prima, il browser deve scoprire l'indirizzo di riferimento per *eloquentjavascript.net*. Poi usa il protocollo HTTP per effettuare una connessione al server con quell'indirizzo e chiede la risorsa `/12_browser.html`.

Del protocollo HTTP parleremo più in dettaglio nel [Capitolo 17](#).

## HTML

Il formato HTML, che sta per *Hypertext Markup Language*, si applica ai documenti usati come pagine Web. Un documento HTML contiene del testo e dei *tag*, che danno struttura al testo descrivendo aspetti come i link (collegamenti ipertestuali), i paragrafi e i titoli.

Ecco come può presentarsi un semplice documento HTML:

---

```
<!doctype html>
<html>
<head>
  <title>My home page</title>
</head>
<body>
  <h1>My home page</h1>
  <p>Hello, I am Marijn and this is my home page.</p>
```

```
<p>I also wrote a book! Read it  
    <a href="http://eloquentjavascript.net">here</a>.</p>  
</body>  
</html>
```

---

Ed ecco come il browser rende quel documento:

# My home page

Hello, I am Marijn and this is my home page.

I also wrote a book! Read it here.

I tag, avvolti in parentesi angolari (< e >), forniscono informazioni sulla struttura del documento. Il testo compreso tra tag è semplice testo.

Il documento si apre con <!doctype html>, che indica al browser di interpretarne il contenuto come HTML *moderno* e non come uno dei suoi dialetti che venivano usati in passato.

I documenti HTML hanno un'intestazione (*head*) e un corpo (*body*). L'intestazione contiene informazioni *sul* documento e il corpo contiene il documento stesso. Nell'esempio, abbiamo innanzitutto dichiarato che il titolo del documento è "My home page"; abbiamo poi specificato un documento che contiene un titolo (<h1>, che significa "heading 1" o titolo di primo livello; i tag da <h2> a <h6> producono titoli di livelli inferiori) e due paragrafi (<p>).

Esistono diverse forme di tag. Ogni elemento, come il corpo, un paragrafo o un link, si marca con un *tag di apertura*, come <p>, e termina con un *tag di chiusura*, come </p>. Alcuni tag di apertura, come quello per i link (<a>), contengono informazioni aggiuntive sotto forma di coppie nome="valore": i cosiddetti *attributi*. In questo caso, la destinazione del link è indicata da href="http://eloquentjavascript.net", dove href sta per "hypertext reference", o riferimento ipertestuale.

Alcuni tipi di tag non racchiudono nulla e pertanto non devono essere chiusi. Un esempio è , che visualizzerà l'immagine che si trova all'URL dato.

Per poter inserire delle parentesi angolari nel testo di un documento, al di fuori del loro significato speciale in HTML, è stata introdotta un'altra forma di notazione: una parentesi angolare aperta va scritta nella forma &lt; ("less than", minore di) e una chiusa come &gt; ("greater than", maggiore di). In HTML, il carattere & seguito da una parola e da un punto e virgola (;) si chiama *entità* e viene sostituito dal carattere che rappresenta.

Il che è analogo a come si usano le barre rovesciate in JavaScript. Poiché questo meccanismo dà un significato speciale anche ai caratteri & presenti nel testo, dovremo indicarli con la stessa notazione, come &amp; ("ampersand"). All'interno di un attributo,

che va racchiuso tra virgolette doppie, si usa " per indicare il carattere “virgolette”.

L’HTML viene analizzato in modo estremamente tollerante. Quando mancano tag che dovrebbero esserci, il browser li ricostruisce. Questo procedimento è stato standardizzato e tutti i browser moderni lo seguono in modo coerente.

Il seguente documento viene trattato proprio come quello riportato in precedenza:

---

```
<!doctype html>
<title>My home page</title>
<h1>My home page</h1>
<p>Hello, I am Marijn and this is my home page.
<p>I also wrote a book! Read it
<a href="http://eloquentjavascript.net">here</a>.
```

---

Sono spariti completamente i tag <html>, <head> e <body>. Il browser, però, sa che <title> va nell’intestazione e <h1> nel corpo. Inoltre, non ho chiuso esplicitamente i paragrafi, perché aprirne uno nuovo o chiudere il documento forza il browser a chiuderli implicitamente. Sono sparite anche le virgolette che racchiudevano il link di destinazione.

In questo libro di solito ometto dagli esempi i tag <html>, <head> e <body> per risparmiare spazio e renderli più chiari. Chiuderò, comunque, *sempre* i tag e racchiuderò *sempre* gli attributi tra virgolette.

Di solito ometterò anche i tag <!doctype>. Il che non va preso come un incoraggiamento a farlo! Anzi, i browser possono fare delle cose davvero ridicole se li lasciate fuori. Pertanto, dovrete considerarli come implicitamente presenti in tutti gli esempi, anche quando non sono riportati testualmente.

## HTML e JavaScript

Nel contesto di questo libro, il tag HTML più importante è <script>, che è quello che permette di inserire in un documento un brano di JavaScript.

---

```
<h1>Testing alert</h1>
<script>alert("hello!");</script>
```

---

Lo script viene eseguito appena il browser legge l’HTML e incontra il tag <script>. Quando viene aperta la pagina che contiene il codice qui sopra, quindi, si apre una finestra di dialogo alert .

Inserire documenti di grandi dimensioni direttamente all’interno di pagine HTML è in genere poco pratico. Ecco perché al tag <script> si può assegnare un attributo src per recuperare il file dello script (un file di testo che contiene il programma in JavaScript) da un URL.

---

```
<h1>Testing alert</h1>
<script src="code/hello.js"></script>
```

---

Questo file *code/hello.js* contiene lo stesso semplice programma, `alert("hello!")`. Quando in una pagina HTML si fa riferimento ad altri URL, come file di immagini o di script, il browser li recupera immediatamente e li inserisce nella pagina.

Il tag `<script>` va sempre chiuso con `</script>`, anche se fa solo riferimento a un file e non contiene codice. Se dimenticate il tag di chiusura, tutto il resto della pagina viene interpretato come se facesse parte dello script.

Anche alcuni attributi possono contenere programmi JavaScript. Il tag `<button>`, che viene visualizzato come un pulsante, ha un attributo `onclick`, il cui contenuto viene eseguito quando si preme il pulsante:

---

```
<button onclick="alert('Boom!')">DO NOT PRESS</button>
```

---

Noteate che ho dovuto usare virgolette semplici per la stringa nell'attributo `onclick`, perché le virgolette doppie avvolgono già tutto l'attributo. Avrei potuto usare invece `"`, rendendo però il programma più difficile da leggere.

## Nell'ambiente di prova

Eseguire programmi scaricati da Internet può essere pericoloso. Non sapete quasi nulla della gente che sta dietro i siti che visitate e non potete essere sicuri che siano tutti bene intenzionati. Eseguire programmi di chi non ha intenzioni serie porta a conseguenze gravi, come trovarsi col computer infettato da virus, farsi rubare dati e rischiare che altri accedano ai vostri account esterni.

La grande attrattiva del Web, invece, è che lo si può navigare senza bisogno di fidarsi di tutte le pagine che si visitano. Ecco perché i browser pongono limiti severi a quel che i programmi in JavaScript possono fare: in particolare, non consentono a quei programmi di vedere i file sul vostro computer, né modificare cose che non siano parte della pagina che li contiene.

Quando un ambiente di programmazione viene isolato in questo modo, si parla di *sandbox*, in quanto richiama l'idea del giocare in una sabbiera, senza far male a nessuno. Questo tipo di sabbiera, però, è ricoperta da una gabbia di barre d'acciaio ed è pertanto molto diversa da quella di un parco giochi.

La difficoltà con questo sistema sta nel lasciare ai programmi abbastanza spazio per essere utili, impedendo allo stesso tempo che facciano danni. Molte funzionalità potenzialmente utili, come comunicare con altri server o leggere il contenuto degli appunti, sono anche potenzialmente dannose e possono compromettere la privacy.

Ogni tanto, qualcuno inventa nuovi modi per aggirare le limitazioni imposte dai browser e provocare danni, che vanno dalla perdita di informazioni riservate al prendere il

controllo di tutto il computer. Gli sviluppatori di browser replicano sistemando i bachi e tutto torna a funzionare come dovrebbe, finché non si scopre un nuovo problema e, si spera, se ne parla pubblicamente e non si lascia che venga segretamente sfruttato dai governi o dalla mafia.

## Compatibilità e guerre tra browser

Agli albori di Internet, dominava il mercato un browser chiamato Mosaic. Dopo pochi anni, il dominio era passato a Netscape, che venne a sua volta soppiantato da Internet Explorer. Tutte le volte che un solo browser aveva il predominio, gli sviluppatori di quel browser si sentivano autorizzati a inventare unilateralmente nuove funzionalità per il Web. Dal momento che la stragrande maggioranza degli utenti faceva affidamento allo stesso browser, i siti Web cominciavano a usare quelle funzionalità, ignorando il resto dei browser (e degli utenti).

Era questo il medioevo della compatibilità e in questo periodo si sentiva spesso parlare di *browser wars* (guerre dei browser). E gli sviluppatori si trovavano non con un solo Web, ma con due o tre piattaforme incompatibili. A peggiorare le cose, i browser in uso intorno al 2003 erano pieni di bachi, ovviamente tutti diversi da un browser all'altro. La vita era dura per chi scriveva pagine Web.

Mozilla Firefox, un'iniziativa senza scopo di lucro partita da Netscape, mise fine all'egemonia di Internet Explorer verso la fine del primo decennio del 2000. Poiché Microsoft non aveva interesse a rimanere competitiva all'epoca, Firefox si assicurò una fetta consistente di mercato. In quegli stessi anni, Google lanciò il suo browser, Chrome, mentre guadagnava popolarità anche il browser di Apple, Safari, e il campo si divise tra quattro contendenti invece di uno solo.

I nuovi browser godevano tutti di un atteggiamento più serio nei confronti degli standard e di migliori impostazioni strutturali, il che portò via via a sempre meno incompatibilità e bachi. Anche Microsoft, vedendosi assottigliare la quota di mercato, dovette cambiare atteggiamento e adottare gli stessi principi. Se cominciate ora a studiare lo sviluppo di applicazioni Web, siete fortunati: le ultime versioni dei browser più importanti si comportano ormai in modo uniforme e hanno relativamente pochi bachi.

Il che non significa che la situazione sia perfetta. C'è ancora gente che, per questioni di inerzia o di politiche aziendali, usa il Web con browser molto vecchi. Finché non spariranno del tutto, costruire siti Web che funzionano con quei browser richiederà un sacco di conoscenze arcane sui relativi difetti e comportamenti strani. Questo libro però non si occupa di quei problemi: punta invece a presentare uno stile moderno, ragionevole per la programmazione sul Web.

# IL MODELLO A OGGETTI DEL DOCUMENTO: DOM

Quando apriete una pagina Web nel browser, il browser recupera il codice HTML della pagina e lo analizza, proprio come il parser che abbiamo visto nel [Capitolo 11](#) analizzava i programmi. Il browser costruisce quindi un modello della struttura del documento, sulla base del quale traccia la pagina sullo schermo.

Questa rappresentazione del documento è uno dei gingilli che un programma JavaScript ha a disposizione nello spazio sandbox. Potete leggere il modello e modificarlo. Agisce come una struttura dati *viva*: quando la modificate, la pagina sullo schermo si aggiorna per riflettere le modifiche.

## Struttura dei documenti

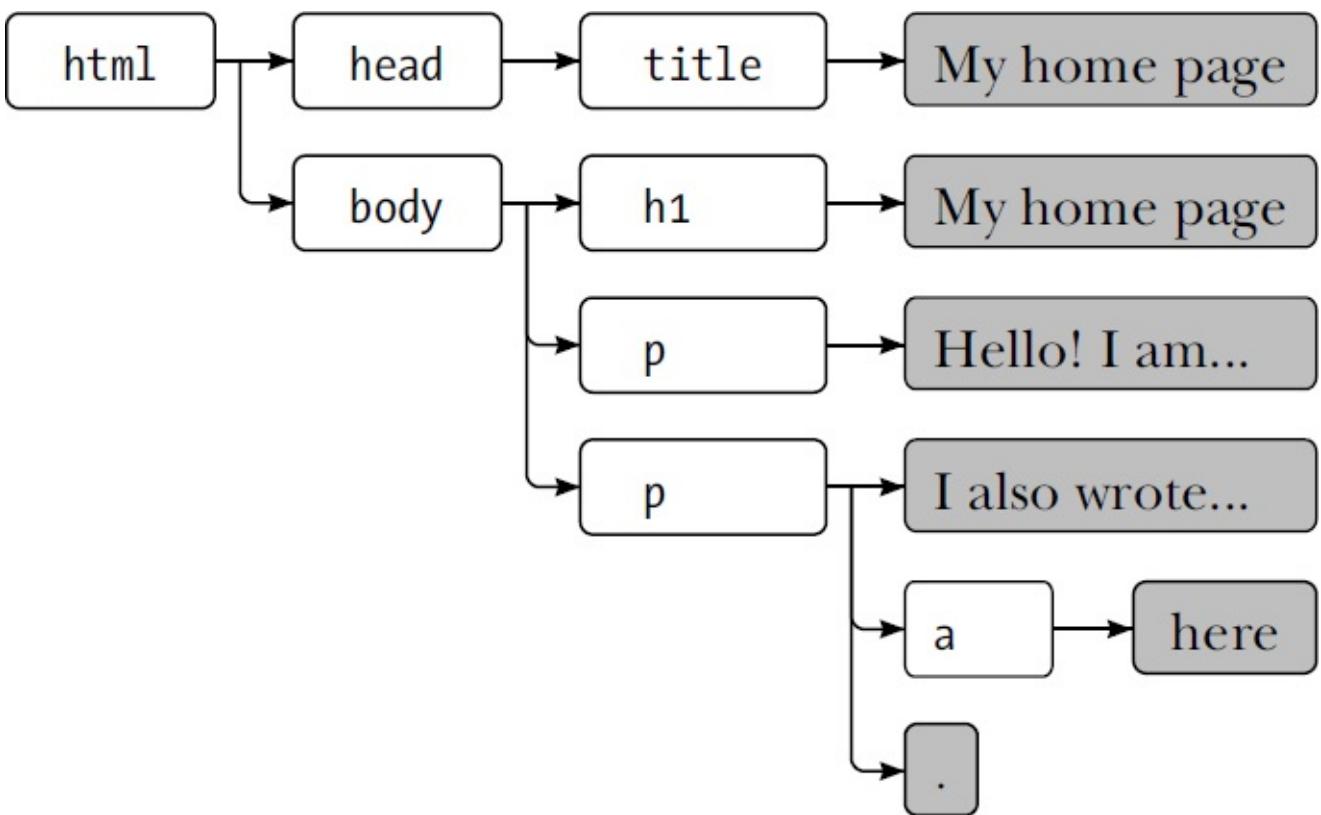
Potete pensare a un documento HTML come a una serie di scatole una dentro l'altra. Tag come `<body>` e `</body>` racchiudono altri tag o testo. Ecco il documento di esempio del capitolo precedente:

---

```
<!doctype html>
<html>
<head>
  <title>My home page</title>
</head>
<body>
  <h1>My home page</h1>
  <p>Hello, I am Marijn and this is my home page.</p>
  <p>I also wrote a book! Read it
    <a href="http://eloquentjavascript.net">here</a>.</p>
</body>
</html>
```

---

Questa pagina ha la seguente struttura:



La struttura dati che il browser usa per rappresentare il documento rispecchia questa forma. Ciascuna scatola corrisponde a un oggetto, col quale possiamo interagire per scoprire, per esempio, che cosa rappresenta il tag HTML e quali altre scatole e testo esso contiene. Questa rappresentazione si chiama *DOM*, che sta per *Document Object Model*.

La variabile globale `document` ci dà accesso a questi oggetti. La sua proprietà `documentElement` fa riferimento all’oggetto che rappresenta il tag `<html>` e dà accesso alle proprietà `head` e `body`, che contengono gli oggetti per i rispettivi elementi.

## Alberi

Tornate per un momento agli alberi sintattici del [Capitolo 11](#). Le loro strutture sono molto simili a quella di un documento in un browser. Ciascuno dei *nodi* può fare riferimento ad altri nodi *figli*, che a loro volta possono avere nodi figli. Questa forma è tipica delle strutture nidificate, dove ogni elemento può contenere altri elementi, che a loro volta ne possono contenere altri e così via.

Quando parliamo di strutture ad *albero*, ci riferiamo a strutture dati di forma ramificata, senza cicli (un nodo non può contenere se stesso, né direttamente, né indirettamente) e con un’unica “radice” ben definita. Nel caso del modello DOM, la radice è `document.documentElement`.

L’immagine dell’albero si presenta spesso in informatica. Oltre che per rappresentare strutture ricorsive come i documenti HTML o i programmi, le strutture ad albero sono spesso usate per mantenere serie di dati ordinati, in quanto si trovano o si inseriscono più facilmente elementi in una struttura ad albero che in un array piatto.

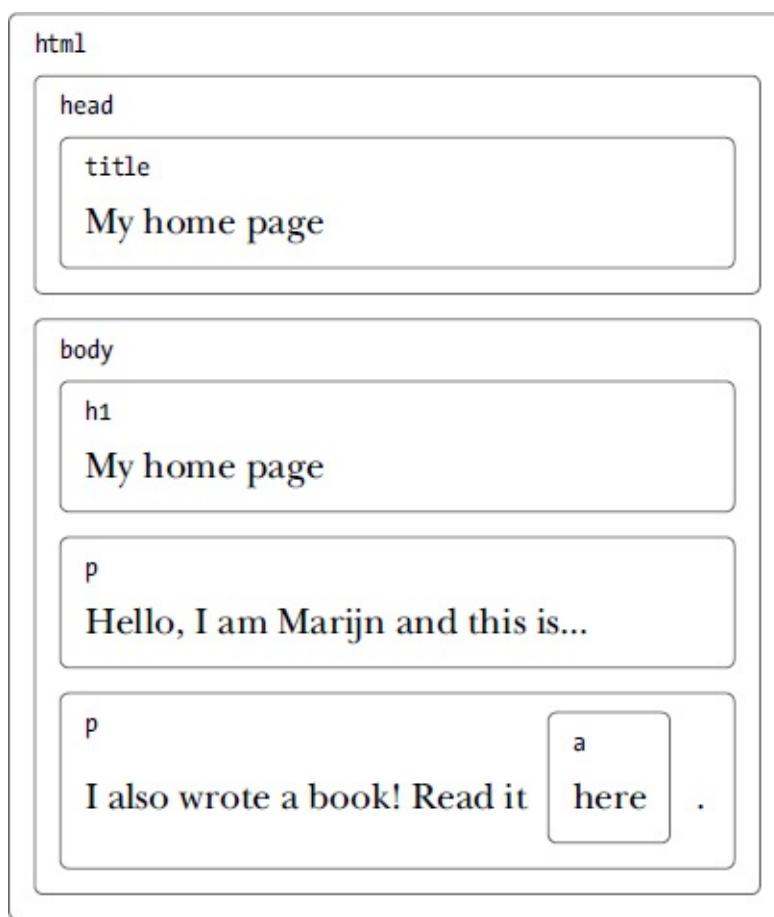
Il tipico albero ha diversi tipi di nodi. L’albero sintattico per il linguaggio Egg aveva

variabili, valori e nodi per le applicazioni. Questi ultimi hanno sempre figli, mentre variabili e valori sono *foglie*, o nodi senza figli.

Lo stesso vale per il modello DOM. I nodi per gli *elementi* normali, che rappresentano i tag HTML, determinano la struttura del documento e possono avere nodi figli. Un esempio è `document.body`. Alcuni di questi nodi figli possono essere nodi a foglia, per esempio quando si tratta di testo o commenti (che in HTML si inseriscono tra `<!--` e `-->`).

Ciascun oggetto nodo ha una proprietà `nodeType`, che contiene un codice numerico che identifica il tipo di nodo. Gli elementi normali hanno valore 1, che è definito anche come proprietà costante `document.ELEMENT_NODE`. I nodi di testo, che rappresentano un brano di testo, hanno valore 3 (`document.TEXT_NODE`). I commenti hanno valore 8 (`document.COMMENT_NODE`).

Ecco dunque un altro modo per rappresentare graficamente l'albero del documento:



Le foglie sono nodi di testo e le frecce indicano le relazioni padre-figlio/ genitore-figlio.

## Lo standard

Usare dei misteriosi codici numerici per rappresentare il tipo di nodo non è una cosa molto da JavaScript. Più avanti in questo capitolo, vedremo che altre parti dell'interfaccia del modello DOM sono strane e scomode. La ragione è che il modello DOM non è specifico per JavaScript: è invece un tentativo di definire un'interfaccia indipendente dal linguaggio, che si possa utilizzare anche in altri sistemi (non solo JavaScript, ma anche

XML, che è un formato dati generico con una sintassi simile a quella dell'HTML).

Peccato però. Gli standard sono generalmente molto utili. In questo caso, comunque, il vantaggio (coerenza in tutti i linguaggi) non è poi fondamentale. Avere un'interfaccia che si integri col linguaggio che usate, vi farà risparmiare più tempo di un'interfaccia comune a più linguaggi.

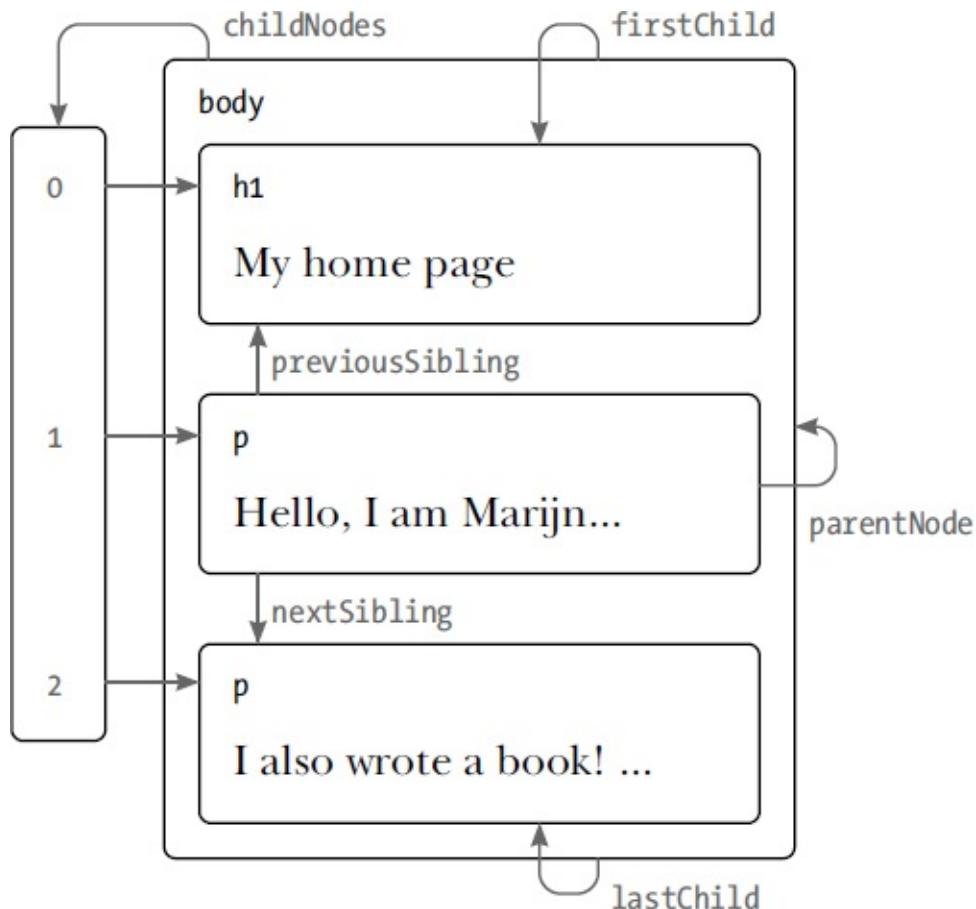
Come esempio di scarsa integrazione, prendete la proprietà `childNodes` degli elementi nodo del DOM. Questa proprietà mantiene un oggetto simile a un array, con una proprietà `length` e altre proprietà distinte da numeri per accedere ai nodi figli. Poiché è un'istanza del tipo `NodeList` e non un vero array, non offre metodi come `slice` e `forEach`.

Ci sono poi aspetti di cattiva progettazione. Per esempio, non c'è modo di creare un nuovo nodo e aggiungervi immediatamente dei figli o degli attributi. Bisogna invece prima crearlo, quindi aggiungere i figli uno per uno, e infine impostare i singoli attributi usando degli effetti collaterali ([Capitolo 3](#)). Il codice che interagisce col DOM finisce con l'essere lungo, ripetitivo e poco elegante.

Questi difetti, però, non sono fatali. Poiché JavaScript consente di creare delle astrazioni, è facile scrivere delle funzioni ausiliarie per esprimere le operazioni da svolgere in modo chiaro e conciso. In effetti, ci sono molte librerie per la programmazione dei browser che offrono strumenti di questo tipo.

## Spostarsi lungo l'albero

I nodi del modello DOM contengono moltissimi collegamenti ai nodi vicini, come illustra il diagramma seguente:



Sebbene il diagramma mostri un solo collegamento per tipo, tutti i nodi hanno una proprietà `parentNode`, che punta al nodo padre relativo. Analogamente, tutti i nodi di tipo elemento (contrassegnati da 1) hanno una proprietà `childNodes` che punta a un oggetto simile a un array che ne contiene i figli.

In teoria, potete spostarvi dove volete seguendo solo i collegamenti tra nodi padri e figli. JavaScript, però, dà accesso anche ad altri comodi collegamenti. Le proprietà `firstChild` e `LastChild` puntano rispettivamente agli elementi primo e ultimo figlio, oppure hanno valore `null` per i nodi senza figli. Poi, `previousSibling` e `nextSibling` puntano a nodi adiacenti, ossia nodi con lo stesso padre che si trovano immediatamente prima o dopo il nodo in questione. Per il primo figlio, `previousSibling` avrà valore `null`; per l'ultimo figlio, `nextSibling` avrà valore `null`.

Quando si lavora con strutture dati nidificate come queste, vengono spesso utili le funzioni ricorsive. La seguente funzione ricorsiva effettua la scansione di un documento, cercando i nodi che contengono una determinata stringa, e restituisce `true` quando la trova:

---

```

function talksAbout(node, string) {
  if (node.nodeType == document.ELEMENT_NODE) {
    for (var i = 0; i < node.childNodes.length; i++) {
      if (talksAbout(node.childNodes[i], string))
        return true;
    }
  }
}

```

```
    return false;
} else if (node.nodeType == document.TEXT_NODE) {
    return node.nodeValue.indexOf(string) > -1;
}
}

console.log(talksAbout(document.body, "book"));
// → true
```

---

La proprietà `nodeValue` di un nodo di testo fa riferimento alla stringa testuale che rappresenta.

## Trovare gli elementi

Con la funzione precedente, che passa in esame tutto il documento, abbiamo visto come ci possiamo spostare tra genitori, figli e fratelli. Se, però, vogliamo trovare un nodo specifico, arrivarcì partendo da `document.body` e seguendo ciecamente un percorso di collegamenti fissi non è una buona idea, in quanto introduce nel programma delle supposizioni sulla struttura esatta del documento (una struttura che magari vogliamo cambiare in un secondo tempo). Un'altra complicazione viene dal fatto che vengono creati nodi di testo anche per lo spazio vuoto tra nodi. Il tag `<body>` del documento di esempio non ha solo tre figli (un elemento `<h1>` e due `<p>`), anzi ne ha ben sette: oltre a quei tre, ci sono gli spazi prima, dopo e tra di essi.

Se pertanto vogliamo arrivare all'attributo `href` del link che si trova nel documento, non possiamo farlo dicendo “Recupera il secondo figlio del sesto figlio del corpo del documento”. Meglio chiedere di recuperare il primo link del documento, cosa che si fa così:

```
var link = document.body.getElementsByTagName("a")[0];
console.log(link.href);
```

---

Tutti i nodi `element` hanno un metodo `getElementsByTagName`, che va a cercare tutti gli elementi col tag indicato che sono discendenti (figli diretti o indiretti) del nodo dato, e li restituisce come oggetto simile a un array.

Per trovare un solo nodo specifico, potete dargli un attributo `id` e usare invece `document.getElementById`.

---

```
<p>My ostrich Gertrude:</p>
<p></p>
<script>
    var ostrich = document.getElementById("gertrude");
    console.log(ostrich.src);
</script>
```

---

Un terzo metodo, simile ai due appena visti, è `getElementsByClassName`, che, come `getElementsByTagName`, va a cercare nel contenuto di un nodo elemento e recupera tutti gli elementi che contengono la stringa data nel proprio attributo `class`.

## Modificare il documento

Della struttura del DOM si può cambiare quasi tutto. I nodi elemento hanno una serie di metodi con i quali possiamo modificarne il contenuto. Il metodo `removeChild` elimina il nodo figlio dato. Per aggiungere un nodo figlio, possiamo usare `appendChild`, che lo aggiunge alla fine dell'elenco di figli, o `insertBefore`, che lo inserisce come primo argomento, prima del nodo dato come secondo argomento.

---

```
<p>One</p>
<p>Two</p>
<p>Three</p>
<script>
  var paragraphs = document.body.getElementsByTagName("p");
  document.body.insertBefore(paragraphs[2], paragraphs[0]);
</script>
```

---

Ogni nodo può esistere in una sola posizione del documento. Pertanto, se inseriamo il paragrafo “Three” prima di “One”, quel paragrafo viene spostato dalla fine all’inizio del documento, che diventerà “Three/One/Two”. Tutte le operazioni che inseriscono un nodo da qualche parte hanno l’effetto collaterale di eliminarlo dalla sua posizione corrente (se esiste).

Il metodo `replaceChild` serve per sostituire un nodo figlio con un altro. Accetta come argomenti due nodi: il nuovo nodo e quello da sostituire. Il nodo sostituito deve essere figlio dell’elemento su cui si richiama il metodo. Notate che sia `replaceChild`, sia `insertBefore` si aspettano il nuovo nodo come primo argomento.

## Creare i nodi

Nell’esempio che segue, vogliamo uno script che sostituisca tutte le immagini del documento (tag `<img>`) col testo contenuto nei rispettivi attributi `alt`, che specificano una rappresentazione testuale alternativa all’immagine.

Questo implica non solo eliminare le immagini, ma anche aggiungere per ciascuna un nuovo nodo di testo che le sostituisca. Per far ciò, usiamo il metodo `document.createElementNode`.

---

```
<p>The  in the
.</p>
```

```
<p><button onclick="replaceImages()">Replace</button></p>
<script>

    function replaceImages() {
        var images = document.body.getElementsByTagName("img");
        for (var i = images.length - 1; i >= 0; i--) {
            var image = images[i];
            if (image.alt) {
                var text = document.createTextNode(image.alt);
                image.parentNode.replaceChild(text, image);
            }
        }
    }
</script>
```

---

Data una stringa, `createTextNode` ci dà un nodo di tipo 3 (un nodo di testo nel DOM), che possiamo inserire nel documento in modo da visualizzarlo sullo schermo.

Il ciclo che analizza le immagini parte dalla fine dell'elenco dei nodi. Ciò è necessario perché l'elenco dei nodi restituito da un metodo come `getElementsByName` (o una proprietà come `childNodes`) è vivo, ossia viene aggiornato man mano che cambia il documento. Se partissimo dall'inizio, togliere la prima immagine eliminerebbe il primo elemento dell'elenco; pertanto, la seconda volta il ciclo trova che `i` è 1, e si ferma perché anche la lunghezza dell'elenco nel frattempo è diventata 1.

Se volete una collezione *fissa* di nodi, invece di una dinamica, potete convertirla in un vero array richiamando il metodo `slice`:

---

```
var arrayish = {0: "one", 1: "two", length: 2};
var real = Array.prototype.slice.call(arrayish, 0);
real.forEach(function(elt) { console.log(elt); });
// → one
// two
```

---

Per creare nodi di elementi normali (tipo 1), potete usare il metodo `document.createElement`, che accetta un nome di tag e restituisce un nodo vuoto del tipo dato.

L'esempio seguente definisce un'utilità `elt`, che crea un nodo element e tratta il resto degli argomenti come figli di quel nodo. Si usa poi questa funzione per aggiungere un semplice attributo a una citazione.

---

```
<blockquote id="quote">
```

No book can ever be finished. While working on it we learn just enough to find it immature the moment we turn away from it.

```

</blockquote>

<!-- trad.: Nessun libro può essere veramente compiuto. Mentre vi
lavoriamo sopra, impariamo abbastanza da trovarlo immaturo nel momento in
cui ce ne distacchiamo. Karl Popper, prefazione alla seconda edizione di
La società aperta e i suoi nemici, 1950 -->

<script>

function elt(type) {
    var node = document.createElement(type);
    for (var i = 1; i < arguments.length; i++) {
        var child = arguments[i];
        if (typeof child == "string")
            child = document.createTextNode(child);
        node.appendChild(child);
    }
    return node;
}

document.getElementById("quote").appendChild(
elt("footer", "-",
elt("strong", "Karl Popper"),
", preface to the second editon of ",
elt("em", "The Open Society and Its Enemies"),
", 1950"));

</script>

```

---

Ecco come viene visualizzato il documento:

No book can ever be finished. While working on it we learn just enough to  
find it immature the moment we turn away from it.

—**Karl Popper**, preface to the second editon of *The Open Society and Its  
Enemies*, 1950

## Attributi

Si può accedere ad alcuni attributi, come href per i link, attraverso una proprietà dello stesso nome per l'oggetto DOM dell'elemento. È questo il caso per un piccolo gruppo di attributi standard di uso comune.

La sintassi HTML vi permette di impostare qualunque attributo sui nodi, cosa che risulta molto utile per inserire nel documento informazioni aggiuntive. Se però create dei nuovi attributi, con nuovi nomi, quegli attributi non saranno disponibili come proprietà del nodo dell'elemento. Dovrete invece usare i metodi getAttribute per recuperarli e

setAttribute per impostarli:

---

```
<p data-classified="secret">The launch code is 00000000.</p>
<p data-classified="unclassified">I have two feet.</p>
<script>
  var paras = document.body.getElementsByTagName("p");
  Array.prototype.forEach.call(paras, function(para) {
    if (para.getAttribute("data-classified") == "secret")
      para.parentNode.removeChild(para);
  });
</script>
```

---

Consiglio sempre di anteporre ai nomi di questi attributi personalizzati un prefisso, per esempio data-, per evitare possibili conflitti con altri attributi.

Come esempio, imposteremo un “evidenziatore di sintassi” che cerca i tag `<pre>` (“preformatted”, preformattato, usati per il codice e simili brani di testo semplice) con un attributo `data-language` e cerca di mettere in evidenza le parole chiave per quel linguaggio.

---

```
function highlightCode(node, keywords) {
  var text = node.textContent;
  node.textContent = ""; // Clear the node
  var match, pos = 0;
  while (match = keywords.exec(text)) {
    var before = text.slice(pos, match.index);
    node.appendChild(document.createTextNode(before));
    var strong = document.createElement("strong");
    strong.appendChild(document.createTextNode(match[0]));
    node.appendChild(strong);
    pos = keywords.lastIndex;
  }
  var after = text.slice(pos);
  node.appendChild(document.createTextNode(after));
}
```

---

La funzione `highlightCode` accetta un nodo `<pre>` e un’espressione regolare, con l’opzione “globale” attiva, che corrisponde alle parole chiave del linguaggio di programmazione contenuto nell’elemento.

La proprietà `textContent` recupera tutto il testo del nodo e viene poi impostata su una stringa vuota, in modo da svuotare il nodo. Passiamo in ciclo su tutte le occorrenze

dell'espressione per le parole chiave, aggiungendo il testo tra di esse come semplici nodi di testo, e il testo che corrisponde alle parole chiave come nodi di testo chiusi tra elementi `<strong>` (grassetto).

Possiamo mettere automaticamente in evidenza tutti i programmi della pagina facendo passare il ciclo su tutti gli elementi `<pre>` che hanno un attributo `data-language` e richiamando `highlightCode` su ciascuno di essi con l'espressione regolare appropriata per il linguaggio.

---

```
var languages = {
    javascript: /\b(function|return|var)\b/g /* ... etc */
};

function highlightAllCode() {
    var pres = document.body.getElementsByTagName("pre");
    for (var i = 0; i < pres.length; i++) {
        var pre = pres[i];
        var lang = pre.getAttribute("data-language");
        if (languages.hasOwnProperty(lang))
            highlightCode(pre, languages[lang]);
    }
}
```

---

Ecco un esempio:

---

```
<p>Here it is, the identity function:</p>
<pre data-language="javascript">
function id(x) { return x; }
</pre>
<script>highlightAllCode();</script>
```

---

C'è un attributo di uso comune, `class`, che è una parola riservata in JavaScript. Per ragioni storiche, in quanto in passato alcune versioni di JavaScript non erano in grado di gestire nomi di proprietà che corrispondevano a parole chiave o riservate in JavaScript, la proprietà per accedere a questo attributo si chiama `className`. Attraverso i metodi `getAttribute` e `setAttribute` vi si può accedere anche col suo nome: "class".

## Layout

Avrete notato che la disposizione dei diversi tipi di elementi non è costante. Alcuni, come i paragrafi, `<p>`, e i titoli (`<h1>` e gli altri) occupano tutta la larghezza del documento e sono visualizzati su righe distinte. Questi si chiamano elementi *block-level* (blocchi di testo). Altri, come i link, `<a>`, e l'elemento `<strong>` dell'esempio precedente, sono

visualizzati sulla stessa riga del testo che li circonda e si chiamano elementi *inline*.

Per qualunque documento dato, i browser sono in grado di calcolare il layout, dove a ciascun elemento sono assegnate dimensioni e posizione in base al suo tipo e al suo contenuto. Il layout viene poi usato per tracciare il documento sullo schermo.

JavaScript permette di accedere alle dimensioni e alla posizione di tutti gli elementi. Le proprietà `offsetWidth` e `offsetHeight` danno lo spazio occupato dagli elementi, in *pixel*. Il pixel è l'unità di misura di base del browser e corrisponde di solito al punto più piccolo che lo schermo può visualizzare. Analogamente, `clientWidth` e `clientHeight` danno le dimensioni dello spazio interno dell'elemento, senza contare lo spessore dei bordi.

---

```
<p style="border: 3px solid red">  
  I'm boxed in  
</p>  
<script>  
  var para = document.body.getElementsByTagName("p")[0];  
  console.log("clientHeight:", para.clientHeight);  
  console.log("offsetHeight:", para.offsetHeight);  
</script>
```

---

Assegnare un bordo a un paragrafo traccia un rettangolo che lo circonda.

I'm boxed in

Il modo più efficace per trovare la posizione precisa di un elemento sullo schermo è il metodo `getBoundingClientRect`, che restituisce un oggetto con proprietà `top`, `bottom`, `left` e `right` (sopra, sotto, sinistra e destra) che indicano, in pixel, le posizioni dei lati dell'elemento relative all'angolo superiore sinistro dello schermo. Se le volete relative al documento nel suo insieme, dovete aggiungere la posizione corrente del cursore, data dalle variabili globali `pageXOffset` e `pageYOffset`.

Impostare il layout di un documento non è cosa da poco. Per questioni di velocità, i motori dei browser non ritracciano immediatamente il layout di un documento ogni volta che lo cambiate e tendono a rimandare il più possibile quest'operazione. Quando un programma JavaScript che ha modificato un documento arriva alla fine dell'esecuzione, il browser deve ricalcolare il layout per visualizzare sullo schermo il documento modificato. Il browser dovrà ricalcolare il layout anche quando il programma chiede la posizione o le dimensioni di qualcosa, leggendo proprietà come `offsetHeight` o richiamando `getBoundingClientRect`.

Un programma che alterna continuamente richieste di accesso alle informazioni sul layout e istruzioni di modifica del modello DOM forza un gran numero di modifiche al layout e avrà pertanto tempi di esecuzione molto lunghi. Il codice che segue mostra un esempio di questo comportamento. Contiene due programmi diversi che costruiscono una

riga di X caratteri larga 2.000 pixel e registra il tempo necessario per ciascuna.

---

```
<p><span id="one"></span></p>
<p><span id="two"></span></p>
<script>

    function time(name, action) {
        var start = Date.now(); // Current time in milliseconds
        action();
        console.log(name, "took", Date.now() - start, "ms");
    }

    time("naive", function() {
        var target = document.getElementById("one");
        while (target.offsetWidth < 2000)
            target.appendChild(document.createTextNode("X"));
    });
    // → naive took 32 ms

    time("clever", function() {
        var target = document.getElementById("two");
        target.appendChild(document.createTextNode("XXXXX"));
        var total = Math.ceil(2000 / (target.offsetWidth / 5));
        for (var i = 5; i < total; i++)
            target.appendChild(document.createTextNode("X"));
    });
    // → clever took 1 ms
</script>
```

---

## Gli stili

Abbiamo visto che i vari elementi HTML si comportano in modo diverso. Alcuni sono visualizzati come blocchi (block-level), altri all'interno delle righe (inline). Alcuni aggiungono stili di formato: per esempio, `<strong>` applica il grassetto al testo e `<a>` lo sottolinea e lo colora di blu.

Il modo in cui un tag `<img>` visualizza un'immagine, o un tag `<a>` permette di seguire un link quando ci facciamo clic sopra, è legato strettamente al tipo di elemento. Gli stili associati per default all'elemento, però, come il colore del testo o la sottolineatura, possono essere modificati a piacimento. Ecco un esempio di come possiamo usare la proprietà `style`:

---

```
<p><a href=".">Normal link</a></p>
<p><a href=". " style="color: green">Green link</a></p>
```

---

Il secondo link sarà verde, invece che del colore predefinito:

Normal link

Green link

L'attributo di uno stile può contenere una o più *dichiarazioni*, nel formato dove una proprietà (per esempio, `color` per il colore) è seguita da un due punti e da un valore (come `green` per verde). Quando le dichiarazioni sono più di una, dobbiamo separarle con punti e virgola, come in `"color: red; border: none"`.

Sono moltissimi gli aspetti sui quali possiamo intervenire con gli stili. Per esempio, la proprietà `display` controlla se un elemento viene visualizzato come block-level o come inline.

---

```
This text is displayed <strong>inline</strong>,
<strong style="display: block">as a block</strong>, and
<strong style="display: none">not at all</strong>.
```

---

Il tag con `display: block` va a finire su una sua riga, perché gli elementi di questo tipo non sono visualizzati inline, sulla stessa riga del testo che li precede o li segue. L'ultimo tag non viene visualizzato affatto, in quanto `display: none` fa in modo che l'elemento non sia visibile sullo schermo. Questo è un sistema che potete usare per nascondere certi elementi. Può essere una soluzione migliore che eliminarli del tutto, se avete bisogno di renderli visibili in un secondo tempo.

This text is displayed **inline**,
**as a block**
**, and .**

Il codice JavaScript può manipolare direttamente lo stile degli elementi attraverso la proprietà `style` dei rispettivi nodi. Questa proprietà contiene un oggetto con tutte le proprietà stilistiche possibili. I valori di queste proprietà sono stringhe, che possiamo definire per modificare gli aspetti che ci interessano dello stile dell'elemento.

---

```
<p id="para" style="color: purple">
  Pretty text
</p>
<script>
  var para = document.getElementById("para");
  console.log(para.style.color);
```

```
para.style.color = "magenta";  
</script>
```

---

Alcuni nomi di proprietà contengono trattini, per esempio `font-family`. Poiché questo tipo di nome non è comodo da usare in JavaScript (in quanto andrebbe scritto come `style["font-family"]`), i nomi delle proprietà in questi casi perdono il trattino e fanno diventare maiuscola l'iniziale della seconda parola: `style.fontFamily`.

## Stili a cascata

Il sistema di stili per l'HTML si chiama *CSS*, che sta per *Cascading Style Sheets* (fogli stile a cascata). Un *foglio stile* è una serie di regole che definiscono gli stili per gli elementi di un documento. Può essere definito all'interno di un tag `<style>`:

```
<style>  
  strong {  
    font-style: italic;  
    color: gray;  
  }  
</style>  
<p>Now <strong>strong text</strong> is italic and gray.</p>
```

---

L'attributo *a cascata* fa riferimento al fatto che si combinano più regole per produrre lo stile finale di un elemento. Nell'esempio precedente, lo stile di default per i tag `<strong>`, che assegna loro lo stile `font-weight: bold`, viene rimpiazzato dalla regola nel tag `<style>`, che aggiunge `font-style: italic` (corsivo) e `color: gray` per il colore.

Quando più regole definiscono un valore per la stessa proprietà, quella letta per ultima prende la massima precedenza. Pertanto, se la regola nel tag `<style>` avesse specificato `font-weight: normal`, in conflitto con la regola di default per `font-weight`, il testo sarebbe normale e non in grassetto. Gli stili in un attributo `<style>` applicato direttamente al nodo hanno la precedenza più alta e pertanto prendono sempre il sopravvento.

Le regole dei CSS permettono di operare anche su cose diverse dai nomi dei tag. Le regole per `.abc` si applicano a tutti gli elementi che hanno "abc" nel proprio attributo `class`. Le regole per `#xyz` si applicano all'elemento che ha "xyz" come attributo `id`, che deve comparire una sola volta nel documento.

```
.subtle {  
  color: gray;  
  font-size: 80%;  
}  
  
#header {
```

```
background: blue;
color: white;
}
/* p elements, with classes a and b, and id main */
p.a.b#main {
margin-bottom: 20px;
}
```

---

La norma di precedenza sulla regola definita per ultima in ordine di tempo è valida solo quando eventuali altre regole hanno la stessa *specificità*. La specificità di una regola è la misura della precisione con cui descrive gli elementi corrispondenti, stabilita in base al numero e al tipo (tag, class o id) dei relativi aspetti. Per esempio, una regola che si applica a `p.a` è più specifica di una definita solo per `p` o solo per `.a`, e prende pertanto la precedenza rispetto a queste ultime.

La notazione `p > a { ... }` applica gli stili dati a tutti i tag `<a>` che sono figli diretti di tag `<p>`. Analogamente, `p a { ... }` si applica a tutti i tag `<a>` all'interno di tag `<p>`, non importa se siano figli diretti o indiretti.

## Selettori di query

In questo libro non faremo grande uso di fogli stile. Sebbene sia fondamentale comprendere come funzionano per programmare nell'ambiente browser, ci vogliono un paio di altri libri per spiegare dettagliatamente le proprietà su cui intervengono e le interazioni tra di esse.

La ragione principale per cui ho parlato di sintassi dei *selettori*, la notazione dei fogli stile che determina su quali elementi si applicano certi stili, è che possiamo usare lo stesso mini-linguaggio per trovare velocemente gli elementi del DOM.

Il metodo `querySelectorAll`, definito sia per l'oggetto documento, sia per i nodi degli elementi, accetta una stringa come selettore e restituisce un oggetto simile a un array che contiene tutti gli elementi corrispondenti.

```
<p>And if you go chasing
<span class="animal">rabbits</span></p>
<p>And you know you're going to fall</p>
<p>Tell 'em a <span class="character">hookah smoking
<span class="animal">caterpillar</span></span></p>
<p>Has given you the call</p>
<script>
function count(selector) {
  return document.querySelectorAll(selector).length;
```

```

}

console.log(count("p"));           // All <p> elements
// → 4

console.log(count(".animal"));    // Class animal
// → 2

console.log(count("p .animal"));  // Animal inside of <p>
// → 2

console.log(count("p > .animal")); // Direct child of <p>
// → 1

</script>

```

---

Diversamente da `getElementsByName`, l'oggetto restituito non è “vivo” e pertanto non cambia quando modificate il documento.

Il metodo `querySelector`, senza la parte `All`, funziona in modo simile. È utile se volete un solo elemento specifico e restituisce solo il primo elemento corrispondente o `null` se non ne trova nessuno.

## Posizionamento e animazione

La proprietà di `<style>` `position` influenza il layout in modo potente. Per default ha valore `static` (statico), che significa che l'elemento sta al suo posto normale nel documento. Quando è impostata su `relative`, l'elemento occupa sempre il suo spazio nel documento, con la differenza che le sue proprietà `top` e `left` permettono di spostarlo rispetto alla posizione normale. Quando è impostata su `absolute`, l'elemento viene sollevato dal flusso normale del documento: non occupa più spazio all'interno del documento, ma può sovrapporsi ad altri elementi. Inoltre, le sue proprietà `top` e `left` permettono di posizionare l'elemento rispetto all'angolo superiore sinistro dell'elemento circoscritto la cui proprietà `position` non è `static`, oppure rispetto all'angolo superiore sinistro del documento se non è iscritto in un altro elemento.

Possiamo usare questa proprietà per impostare un'animazione. Il documento che segue mostra l'immagine di un gatto che si sposta lungo una traiettoria ellittica:

```

<p style="text-align: center">
  
</p>
<script>
  var cat = document.querySelector("img");
  var angle = 0, lastTime = null;
  function animate(time) {
    if (lastTime != null)

```

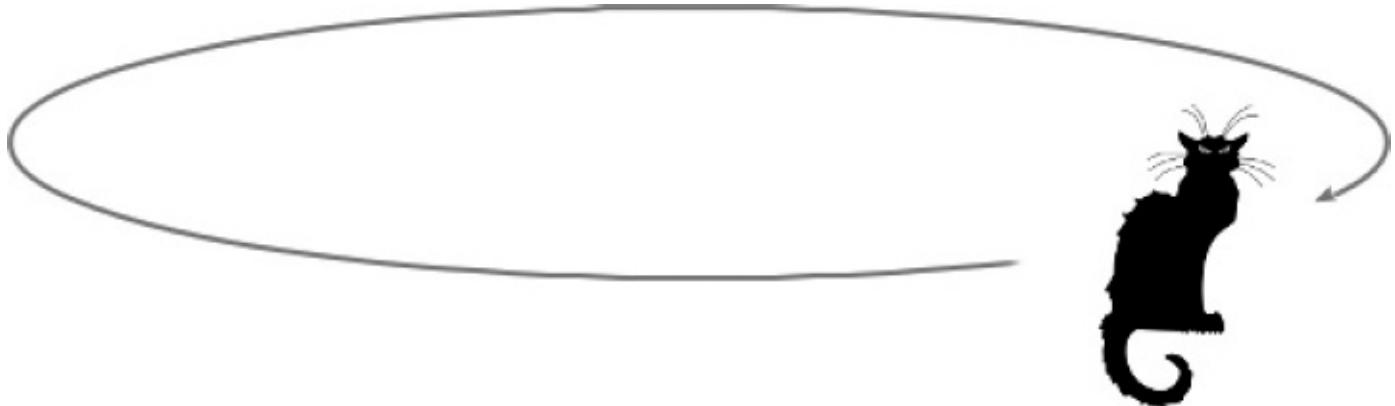
```

        angle += (time - lastTime) * 0.001;
        lastTime = time;
        cat.style.top = (Math.sin(angle) * 20) + "px";
        cat.style.left = (Math.cos(angle) * 200) + "px";
        requestAnimationFrame/animate);
    }
    requestAnimationFrame/animate);
</script>

```

---

La freccia grigia mostra il percorso di spostamento dell'immagine.



L'immagine è centrata sulla pagina e ha l'attributo position impostato su relative. Per spostare l'immagine, aggiorniamo ripetutamente gli stili per top e left.

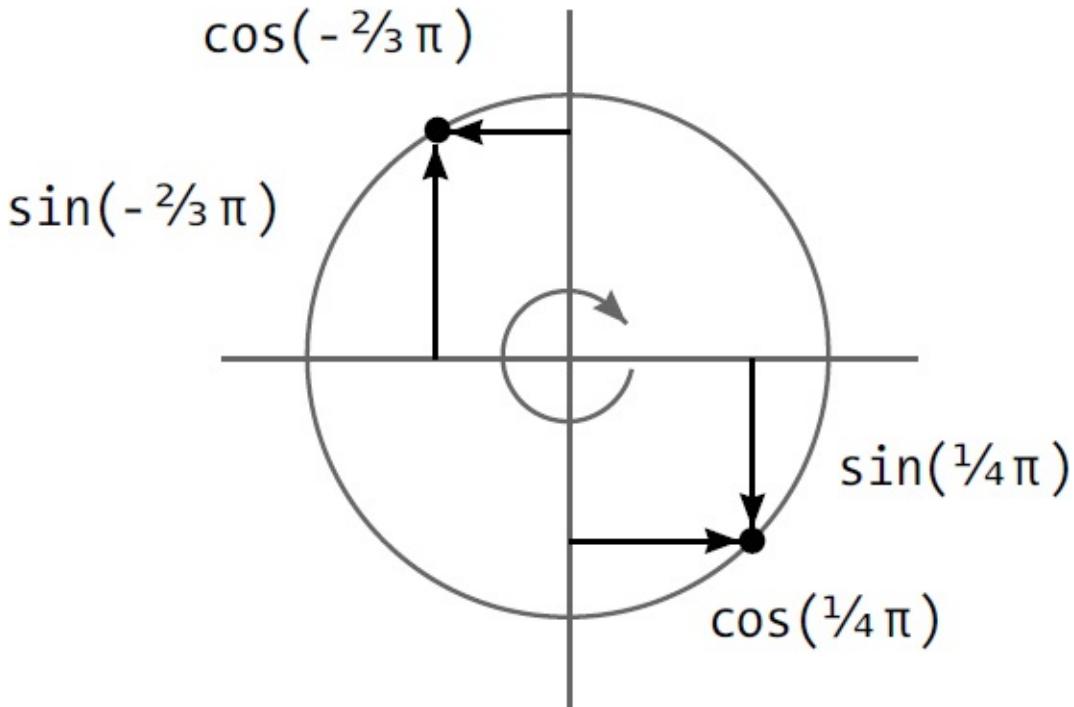
Lo script usa `requestAnimationFrame` per impostare la funzione `animate` in modo che venga eseguita appena il browser è pronto per ritracciare lo schermo. La funzione `animate` richiama poi `requestAnimationFrame` per fissare la scadenza dell'aggiornamento successivo. Quando la finestra (o la scheda) del browser è attiva, gli aggiornamenti si succedono alla velocità di circa 60 al secondo, che produce un buon effetto di animazione.

Se ci limitassimo ad aggiornare il DOM in un ciclo, la pagina si bloccherebbe e sullo schermo non si vedrebbe nulla. I browser non aggiornano lo schermo mentre è in esecuzione un programma JavaScript e non consentono di interagire con la pagina. Ecco perché dobbiamo avere `requestAnimationFrame`, in quanto consente di comunicare al browser che abbiamo finito (per ora) e che può continuare a fare il suo lavoro, ossia aggiornare lo schermo e rispondere alle azioni dell'utente.

La nostra funzione di animazione riceve come argomento l'ora corrente, che confronta con l'ora che aveva rilevato in precedenza (la variabile `lastTime`) per dare stabilità al movimento del gatto per millisecondo e assicurare che l'animazione sia fluida. Se si spostasse di un tanto per volta, il movimento risulterebbe a scatti se, per esempio, ci fosse un altro processo contemporaneo che usa risorse del computer, bloccando l'esecuzione della funzione anche solo per una frazione di secondo.

Il moto circolare si ottiene con le funzioni trigonometriche `Math.cos` e `Math.sin`. Se non le conoscete, ne parlo qui velocemente perché torneranno ancora in altri capitoli del libro.

`Math.cos` e `Math.sin` servono per trovare i punti che giacciono su una circonferenza intorno al punto (0,0) con raggio di una unità. Entrambe le funzioni interpretano il proprio argomento come la posizione lungo la circonferenza, dove zero denota il punto all'estrema destra e ci si muove in direzione oraria finché  $2\pi$  (che è circa 6,28) ci ha portato lungo tutto il cerchio. `Math.cos` indica la coordinata  $x$  del punto che corrisponde alla posizione data sulla circonferenza, mentre `Math.sin` indica la coordinata  $y$ . Sono valide anche posizioni (o angoli) maggiori di  $2\pi$  o minori di 0: la rotazione si ripete, tanto che  $a+2\pi$  riporta alla stessa posizione di  $a$ .



Il codice per l'animazione del gatto mantiene un contatore, `angle`, per l'angolo corrente dell'animazione e lo incrementa in proporzione al tempo trascorso ogni volta che viene richiamata la funzione `animate`. Può quindi usare quest'angolo per calcolare la posizione corrente dell'elemento `<img>`. Lo stile per la posizione `top` è calcolato da `Math.sin` e moltiplicato per 20, che è il raggio del cerchio. Lo stile per la posizione `left` si basa su `Math.cos` ed è moltiplicato per 200, così che il cerchio sia più largo che alto e il moto segua un'ellisse invece di un cerchio.

Si noti che gli stili possono richiedere che sia specificata *l'unità di misura*. In questo caso, aggiungiamo "px" dopo il numero per indicare che i calcoli vanno fatti in pixel (piuttosto che in centimetri, "em" o altre unità). Attenti a non dimenticarvene: se usate dei numeri senza specificare l'unità di misura, il vostro stile sarà ignorato. L'unica eccezione è quando il numero è 0, che significa sempre la stessa cosa a prescindere dall'unità di misura.

## Riepilogo

I programmi JavaScript possono ispezionare e interferire col documento corrente visualizzato dal browser attraverso una struttura dati, chiamata DOM. Questa struttura dati rappresenta il modello del documento nel browser e i programmi JavaScript possono

intervenire su questo modello per modificare il documento visualizzato.

Il modello DOM è una struttura ad albero, dove gli elementi sono organizzati gerarchicamente secondo la struttura del documento. Gli oggetti che rappresentano gli elementi hanno proprietà come `parentNode` e `childNodes`, che possono essere usati per navigare attraverso l’albero.

L’aspetto di un documento può essere influenzato dall’applicazione di *stili* (CSS), sia assegnandoli direttamente ai nodi, sia definendo regole solo per alcuni nodi. Esistono molte proprietà stilistiche, comprese quelle per il colore e la visibilità. JavaScript può modificare lo stile di un elemento direttamente, attraverso la sua proprietà `style`.

## Esercizi

### **Costruire una tabella**

Nel [Capitolo 6](#) abbiamo impostato delle tabelle in testo semplice. L’HTML semplifica grandemente questo compito. La tabella si imposta con la seguente struttura di tag:

---

```
<table>
  <tr>
    <th>name</th>
    <th>height</th>
    <th>country</th>
  </tr>
  <tr>
    <td>Kilimanjaro</td>
    <td>5895</td>
    <td>Tanzania</td>
  </tr>
</table>
```

---

Per ogni riga (della tabella), il tag `<table>` contiene un tag `<tr>`. All’interno dei tag `<tr>` possiamo inserire gli elementi per le celle: celle di testata (`<th>`) o celle normali (`<td>`).

Gli stessi dati usati nel [Capitolo 6](#) sono disponibili nella variabile `MOUNTAINS` della sandbox, <http://eloquentjavascript.net/code/>, dalla quale si possono anche scaricare.

Scrivete una funzione `buildTable` che, dato un array di oggetti che hanno la stessa serie di proprietà, costruisce una struttura DOM che rappresenta una tabella. La tabella deve avere una riga di testata coi nomi delle proprietà avvolti in elementi `<th>`, seguita da una riga per ciascuno degli oggetti dell’array, coi valori delle rispettive proprietà inseriti in elementi `<td>`.

Qui può essere utile la funzione `Object.keys`, che restituisce un array contenente i nomi delle proprietà di un oggetto.

Una volta ottenuta la tabella base, impostate le celle contenenti numeri in modo che siano allineate a destra, specificando "right" per la loro proprietà `style textAlign`.

## ***Elementi per nome di tag***

Il metodo `getElementsByName` restituisce tutti gli elementi figli che hanno il nome di tag dato. Impostatene una vostra versione come funzione regolare, non come metodo, che accetta un nodo e una stringa (il nome del tag) come argomenti e restituisce un array contenente tutti i nodi degli elementi discendenti col nome dato.

Per trovare il nome del tag di un elemento, usate la sua proprietà `tagName`. Notate però che il valore di restituzione sarà in lettere maiuscole e dovete usare i metodi di stringa `toLowerCase` o `toUpperCase` per convertirlo come necessario.

## ***Il cappello del gatto***

Estendete l'animazione descritta in precedenza in modo che sia il gatto, sia il suo cappello (``) girino alle estremità opposte dell'ellisse.

Oppure, fate girare il cappello intorno al gatto o ancora, modificate l'animazione in qualche altro modo interessante.

Per semplificare il compito di posizionare più oggetti, può essere una buona idea passare al posizionamento assoluto. Ciò significa che le posizioni `top` e `left` sono calcolate relativamente all'angolo superiore sinistro del documento. Per evitare di usare coordinate negative, aggiungete un numero fisso di pixel ai rispettivi valori.

## GESTIRE GLI EVENTI

*Avete potere sulla vostra mente, non sugli eventi esterni. Rendetevene conto e troverete forza.*

Marco Aurelio, *Meditazioni*

Alcuni programmi funzionano con l'input diretto dell'utente, ossia le interazioni con mouse e tastiera. Tempi e ordine di questo tipo di input non sono prevedibili a priori. Per questo, ci vuole un approccio al flusso di controllo diverso da quello seguito finora.

### Gestori degli eventi

Immaginate un'interfaccia dove l'unico modo per scoprire se è stato premuto un tasto sia di leggere lo stato corrente dello stesso. Per poter reagire alla pressione dei tasti, dovreste leggerne in continuazione lo stato e catturarlo prima che il tasto venga rilasciato. Sarebbe pericoloso svolgere altri calcoli che richiedono computazioni complesse e che potrebbero farvi perdere uno dei movimenti del tasto.

Così funzionava la gestione di quel tipo di input sulle macchine più vecchie. Un passo avanti è avvenuto quando l'hardware o il sistema operativo rilevavano la pressione di un tasto e la inserivano in una coda: il programma poteva poi controllare periodicamente la coda degli eventi e reagire a quel che ci trovava.

Naturalmente, doveva ricordarsi di andare a vedere che cosa c'era in coda e doveva farlo spesso, perché i tempi di attesa tra la pressione del tasto e il momento in cui il programma si accorge dell'evento danno l'impressione che il software non risponda. Questo tipo di approccio si chiama *polling* e i programmatori lo evitano tutte le volte che possono.

Un meccanismo migliore è quando il sistema su cui si basa il programma dà la possibilità di reagire agli eventi man mano che si verificano. I browser lo fanno dando la possibilità di registrare dei *gestori*, ossia delle funzioni che gestiscono determinati eventi.

---

```
<p>Click this document to activate the handler.</p>
<script>
  addEventListener("click", function() {
    console.log("You clicked!");
  });
</script>
```

```
</script>
```

---

La funzione `addEventListener` fa sì che il suo secondo argomento venga registrato e richiamato ogni volta che si verifica l'evento descritto dal suo primo argomento.

## Eventi e nodi del DOM

I gestori degli eventi del browser sono tutti registrati in un contesto. Quando richiamate `addEventListener` come ho mostrato in precedenza, il metodo si applica a tutta la finestra, perché nel browser l'ambito globale equivale all'oggetto `window`. Tutti gli elementi del DOM hanno un proprio metodo `addEventListener`, che consente di rimanere in attesa degli eventi specifici di ciascun elemento.

---

```
<button>Click me</button>
<p>No handler here.</p>
<script>
  var button = document.querySelector("button");
  button.addEventListener("click", function() {
    console.log("Button clicked.");
  });
</script>
```

Il codice dell'esempio collega un gestore degli eventi al nodo del pulsante. Pertanto, i clic sul pulsante fanno scattare il gestore, mentre quelli sul resto del documento non fanno nulla.

Assegnare a un nodo un attributo `onclick` ha un effetto simile. Poiché ogni nodo ha un solo attributo `onclick`, con questa soluzione potete registrare un solo gestore per nodo. Il metodo `addEventListener` vi permette invece di aggiungere tutti i gestori che volete e non vi succederà di sostituire per sbaglio un gestore che avevate già registrato.

Il metodo `removeEventListener`, richiamato con argomenti simili a quelli per `addEventListener`, elimina un gestore.

---

```
<button>Act-once button</button>
<script>
  var button = document.querySelector("button");
  function once() {
    console.log("Done.");
    button.removeEventListener("click", once);
  }
  button.addEventListener("click", once);
</script>
```

---

Per poter annullare la registrazione di una funzione gestore, le diamo un nome (come per esempio `once`) e la passiamo sia ad `addEventListener`, sia a `removeEventListener`.

## Oggetti evento

Anche se non ne abbiamo parlato negli esempi precedenti, alle funzioni gestore si passa un argomento: l'*oggetto evento*. Quest'oggetto ci dà altre informazioni sull'evento. Per esempio, se vogliamo sapere *quale* pulsante del mouse è stato premuto, possiamo rilevarlo dalla proprietà `which` dell'oggetto evento.

---

```
<button>Click me any way you want</button>
<script>
  var button = document.querySelector("button");
  button.addEventListener("mousedown", function(event) {
    if (event.which == 1)
      console.log("Left button");
    else if (event.which == 2)
      console.log("Middle button");
    else if (event.which == 3)
      console.log("Right button");
  });
</script>
```

---

Le informazioni memorizzate in un oggetto evento variano a seconda del tipo di evento. Parleremo dei diversi tipi più avanti in questo capitolo. La proprietà `type` dell'oggetto contiene sempre una stringa che identifica l'evento (per esempio, "click" o "mousedown").

## Propagazione

I gestori degli eventi registrati sui nodi con figli ricevono inoltre alcuni eventi che hanno luogo nei figli. Se fate clic su un pulsante all'interno di un paragrafo, ricevono l'evento `click` anche i gestori degli eventi del paragrafo.

Se sia il paragrafo, sia il pulsante hanno un proprio gestore, il primo a rispondere sarà il più specifico, ossia quello sul pulsante. Si dice che l'evento *si propaga* verso l'esterno, dal nodo dove è avvenuto al nodo del rispettivo genitore e via via fino alla radice del documento. Alla fine, dopo che tutti i gestori registrati su un certo nodo sono stati chiamati in causa, possono rispondere i gestori registrati per la finestra nel suo insieme.

Un gestore di eventi può richiamare in qualunque momento il metodo `stopPropagation` sull'oggetto evento per impedire ai gestori che lo seguono di ricevere

l'evento. Ciò risulta utile, per esempio, quando avete un pulsante all'interno di un altro elemento cliccabile e non volete che i clic sul pulsante attivino il comportamento previsto dal clic sull'elemento esterno.

L'esempio seguente registra gestori "mousedown" sia per un pulsante, sia per il paragrafo che lo contiene. Quando si fa clic col pulsante destro del mouse, il gestore del pulsante richiama `stopPropagation`, impedendo la reazione del gestore sul paragrafo. Quando si fa clic con un diverso pulsante del mouse, vengono eseguiti entrambi i gestori.

---

```
<p>A paragraph with a <button>button</button>.</p>
<script>
  var para = document.querySelector("p");
  var button = document.querySelector("button");
  para.addEventListener("mousedown", function() {
    console.log("Handler for paragraph.");
  });
  button.addEventListener("mousedown", function(event) {
    console.log("Handler for button.");
    if (event.which == 3)
      event.stopPropagation();
  });
</script>
```

---

Quasi tutti gli oggetti evento hanno una proprietà `target` che fa riferimento al nodo dove hanno avuto inizio. Potete usare questa proprietà per evitare di gestire qualcosa che arriva da un nodo per il quale non volete gestori di eventi.

La proprietà `target` serve anche per lanciare una rete che catturi un certo tipo di eventi. Per esempio, se avete un nodo con una serie di pulsanti, vi può convenire registrare un solo gestore di eventi `click` sul nodo esterno, che usa la proprietà `target` per individuare il pulsante premuto, invece di avere un gestore per ciascuno dei pulsanti.

---

```
<button>A</button>
<button>B</button>
<button>C</button>
<script>
  document.body.addEventListener("click", function(event) {
    if (event.target.nodeName == "BUTTON")
      console.log("Clicked", event.target.textContent);
  });
</script>
```

---

## Azioni predefinite

Molti eventi hanno associata un'azione predefinita. Se fate clic su un link, verrete portati alla sua destinazione. Se fate clic sulla freccia in giù, il browser fa scorrere la pagina verso il basso. Se fate clic col pulsante destro, vi viene presentato il menu contestuale, e così via.

Per quasi tutti gli eventi, i gestori impostati in JavaScript vengono richiamati *prima* del comportamento predefinito. Quando il gestore deve impedire che scatti il comportamento predefinito, di solito perché ha già assunto il ruolo di gestire l'evento, può richiamare il metodo `preventDefault` per l'oggetto evento.

In questo modo, potete impostare scorciatoie da tastiera o menu contestuali personalizzati e potete anche interferire col comportamento normale che l'utente si aspetta. Per esempio, questo link non può essere seguito:

---

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  var link = document.querySelector("a");
  link.addEventListener("click", function(event) {
    console.log("Nope.");
    event.preventDefault();
  });
</script>
```

---

Evitate però questo tipo di comportamento se non avete delle ragioni legittime. Per chi usa la vostra pagina, infatti, può essere sgradevole scoprire che il comportamento che si aspetta non funziona.

Alcuni eventi, che non sono gli stessi per tutti i browser, non possono essere intercettati. Per esempio, in Chrome le scorciatoie di tastiera per chiudere la scheda corrente (CTRL-W o CMD-W) non possono essere modificate da JavaScript.

## Eventi per i tasti

Quando si preme un tasto della tastiera, il browser fa scattare un evento "keydown". Quando lo si rilascia, scatta un evento "keyup".

---

```
<p>This page turns violet when you hold the V key.</p>
<script>
  addEventListener("keydown", function(event) {
    if (event.keyCode == 86)
      document.body.style.background = "violet";
  });
  addEventListener("keyup", function(event) {
```

```
if (event.keyCode == 86)
    document.body.style.background = "";
});
</script>
```

---

A dispetto del nome, "keydown" non scatta solo quando il tasto viene premuto fisicamente. Quando si preme e si tiene premuto un tasto, l'evento scatta tutte le volte che si ripete la pressione. In alcune situazioni, per esempio se volette aumentare l'accelerazione di un carattere (durante un gioco) quando si preme un tasto freccia e diminuirla quando viene rilasciato, dovete far attenzione a non aumentare la velocità per ogni ripetizione del tasto se non volette finire con valori assurdamente alti.

Nell'esempio precedente si esaminava la proprietà `keyCode` dell'oggetto evento. Questo è il modo per identificare quale tasto viene premuto o rilasciato. Purtroppo, non è sempre ovvio come tradurre il relativo codice numerico nel tasto corrispondente.

Per i tasti con lettere e numeri, il codice associato è quello del carattere Unicode per la lettera maiuscola o per il numero indicati sul tasto. Il metodo `charCodeAt` delle stringhe ci dà questo codice.

```
console.log("Violet".charCodeAt(0));
// → 86
console.log("1".charCodeAt(0));
// → 49
```

---

Ad altri tasti corrispondono codici meno prevedibili. Il modo migliore per trovare i codici che vi servono è di fare delle prove: per esempio, potete registrare un gestore di eventi dei tasti che raccoglie i codici dei tasti che vi interessano.

I tasti dei caratteri speciali, come SHIFT/MAIUSC, CTRL (CMD sui Mac), ALT e META generano eventi proprio come tutti gli altri. Quando, però, cercate combinazioni di tasti, potete scoprire se sono premuti questi tasti speciali esaminando le proprietà `shiftKey`, `ctrlKey`, `altKey` e `metaKey` degli eventi di mouse e tasti.

```
<p>Press Ctrl-Space to continue.</p>
<script>
    addEventListener("keydown", function(event) {
        if (event.keyCode == 32 && event.ctrlKey)
            console.log("Continuing!");
    });
</script>
```

---

Gli eventi "keydown" e "keyup" danno informazioni sui tasti fisici che vengono premuti. Ma come si fa col testo che viene scritto? Recuperarlo dai codici dei tasti è scomodo. Esiste invece un altro evento, "keypress", che scatta immediatamente dopo

"keydown" (e viene ripetuto insieme a "keydown" quando si tiene premuto), ma solo per i tasti che producono caratteri come input. La proprietà `charCode` dell'oggetto evento contiene un codice che dà il rispettivo carattere Unicode. Possiamo usare la funzione `String.fromCharCode` per convertire il codice nella stringa corrispondente (da un solo carattere).

---

```
<p>Focus this page and type something.</p>
<script>
  addEventListener("keypress", function(event) {
    console.log(String.fromCharCode(event.charCodeAt(0)));
  });
</script>
```

---

Il nodo del DOM dove ha origine l'evento key dipende dall'elemento che risulta attivo quando si preme il tasto. I nodi normali non possono essere attivi, se non quando avete assegnato loro un attributo `tabindex` (indice di tabulazione), mentre possono esserlo elementi come i link, i pulsanti e i campi dei form. Torneremo sui campi dei form nel [Capitolo 18](#). Quando nulla di specifico è attivo, `document.body` assume il ruolo di nodo di destinazione degli eventi di tastiera.

## Clic del mouse

Anche premere un pulsante del mouse fa scattare una serie di eventi. Gli eventi "mousedown" e "mouseup" sono simili a "keydown" e "keyup", e scattano quando si preme e si rilascia il pulsante. Gli eventi intervengono sui nodi del DOM che in quel momento si trovano immediatamente al di sotto del puntatore.

Dopo l'evento "mouseup", scatta un evento "click" sul nodo più specifico che conteneva gli eventi (giù e su) del pulsante. Per esempio, se premo il pulsante del mouse su un paragrafo e mi sposto su un altro paragrafo prima di rilasciare il pulsante, l'evento "click" scatta sull'elemento che conteneva entrambi i paragrafi.

Se due clic hanno luogo a breve distanza, scatta anche un evento "dblclick" (doppio clic) dopo il secondo clic.

Per ottenere informazioni precise sulla posizione dove è avvenuto un evento del mouse, potete esaminare le sue proprietà `pageX` e `pageY`, che contengono le coordinate dell'evento (in pixel) relative all'angolo superiore sinistro del documento.

Il codice seguente imposta un programma di disegno primitivo. Ogni volta che fate clic sul documento, aggiunge un punto sotto il puntatore del mouse. Nel [Capitolo 19](#) troverete un programma di disegno meno primitivo.

```
<style>
  body {
```

```

height: 200px;
background: beige;
}
.dot {
height: 8px; width: 8px;
border-radius: 4px; /* rounds corners */
background: blue;
position: absolute;
}
</style>
<script>
addEventListener("click", function(event) {
var dot = document.createElement("div");
dot.className = "dot";
dot.style.left = (event.pageX - 4) + "px";
dot.style.top = (event.pageY - 4) + "px";
document.body.appendChild(dot);
});
</script>

```

---

Le proprietà `clientX` e `clientY` sono simili a `pageX` e `pageY`, ma relative alla parte del documento correntemente inquadrata nello schermo. Possono servire per confrontare le coordinate del mouse con quelle restituite da `getBoundingClientRect`, che pure restituisce coordinate relative allo spazio di visualizzazione.

## Movimenti del mouse

Ogni volta che si sposta il puntatore del mouse, scatta un evento "mousemove", che si può usare per seguire la posizione del mouse. Una situazione dove ciò può essere utile è quando volete impostare qualche funzionalità di trascinamento del mouse.

Per esempio, il programma che segue visualizza una barra e imposta dei gestori di eventi in modo che trascinandola (col mouse) a sinistra o a destra la barra diventa più corta o più lunga:

---

```

<p>Drag the bar to change its width:</p>
<div style="background: orange; width: 60px; height: 20px">
</div>
<script>
var lastX; // Tracks the last observed mouse X position

```

```

var rect = document.querySelector("div");
rect.addEventListener("mousedown", function(event) {
    if (event.which == 1) {
        lastX = event.pageX;
        addEventListener("mousemove", moved);
        event.preventDefault(); // Prevent selection
    }
});
function buttonPressed(event) {
    if (event.buttons == null)
        return event.which != 0;
    else
        return event.buttons != 0;
}
function moved(event) {
    if (!buttonPressed(event)) {
        removeEventListener("mousemove", moved);
    } else {
        var dist = event.pageX - lastX;
        var newWidth = Math.max(10, rect.offsetWidth + dist);
        rect.style.width = newWidth + "px";
        lastX = event.pageX;
    }
}
</script>

```

---

Ecco la pagina corrispondente:

**Drag the bar to change its width:**



Notate che il gestore di "mousemove" è registrato su tutta la finestra. Anche se il mouse si sposta fuori dalla barra intanto che trasciniamo, vogliamo che le dimensioni siano aggiornate finché non smettiamo di trascinare rilasciando il pulsante del mouse.

Dobbiamo interrompere il ridimensionamento della barra quando il pulsante viene rilasciato. Purtroppo, non tutti i browser assegnano agli eventi "mousemove" una proprietà `which` appropriata. Esiste una proprietà standard, `buttons` (pulsanti), che offre informazioni simili, ma anche questa non è disponibile in tutti i browser. Per fortuna, i browser più usati offrono l'una o l'altra di queste proprietà: pertanto, la funzione

`buttonPressed` dell'esempio prova prima con `buttons` e poi con `which` se `buttons` non è disponibile.

Ogni volta che il puntatore del mouse entra in un nodo, scatta un evento "mouseover"; quando lo lascia, scatta un evento "mouseout". Questi due eventi servono, tra l'altro, per creare effetti hover ("mouse hover", abbreviato in mouseover), che visualizzano o modificano gli stili di qualche componente quando il mouse passa sopra l'elemento relativo.

Purtroppo, impostare effetti di questo tipo non è immediato, e non basta iniziare un effetto su "mouseover" e interromperlo su "mouseout". Quando il mouse si sposta da un nodo a uno dei suoi figli, "mouseout" scatta sul nodo padre anche se il mouse non ne ha ancora lasciato l'area. Tanto per complicare le cose, questi eventi si propagano come tutti gli altri, il che significa che scatteranno eventi "mouseout" anche quando il mouse abbandona uno dei nodi figli del nodo sul quale è registrato il gestore.

Per aggirare questo problema, possiamo usare la proprietà `relatedTarget` degli oggetti `event` creati per questi eventi. Questa proprietà indica, nel caso di "mouseover", su quale elemento stava il puntatore prima di spostarsi; e nel caso di "mouseout", a quale elemento è diretto. Vogliamo modificare l'effetto hover solo quando `relatedTarget` (la destinazione) è al di fuori del nodo che ci interessa. Solo in quel caso, infatti, questo evento rappresenta un passaggio dall'esterno all'interno del nodo (o viceversa).

---

```
<p>Hover over this <strong>paragraph</strong>.</p>
<script>
  var para = document.querySelector("p");
  function isInside(node, target) {
    for (; node != null; node = node.parentNode)
      if (node == target) return true;
  }
  para.addEventListener("mouseover", function(event) {
    if (!isInside(event.relatedTarget, para))
      para.style.color = "red";
  });
  para.addEventListener("mouseout", function(event) {
    if (!isInside(event.relatedTarget, para))
      para.style.color = "";
  });
</script>
```

La funzione `isInside` segue i collegamenti a monte del nodo dato finché raggiunge la radice del documento (quando `node` diventa `null`) o trova il nodo padre che stiamo cercando.

Dovrei aggiungere che un effetto come questo si ottiene molto più semplicemente attraverso lo *pseudoselettor* :hover dei CSS, come dimostra il prossimo esempio. Ma quando l'effetto hover che volete richiede qualcosa di più complicato del modificare lo stile sul nodo di destinazione, dovete optare per gli eventi "mouseover" e "mouseout".

---

```
<style>
  p:hover { color: red }
</style>
<p>Hover over this <strong>paragraph</strong>.</p>
```

---

## Eventi di scorrimento

Tutte le volte che un elemento scorre, scatta un evento "scroll". Ciò può essere utile per scoprire che cosa sta guardando l'utente (per esempio, per disattivare delle animazioni sullo schermo o per inviare rapporti di spionaggio ai Grandi Capi Cattivi) o per visualizzare delle indicazioni di avanzamento (per esempio, evidenziando parte di un sommario o riportando un numero di pagina).

L'esempio seguente traccia una barra di avanzamento nell'angolo superiore destro del documento e la aggiorna, riempendola, man mano che fate scorrere (la pagina) verso il basso.

---

```
<style>
  .progress {
    border: 1px solid blue;
    width: 100px;
    position: fixed;
    top: 10px; right: 10px;
  }
  .progress > div {
    height: 12px;
    background: blue;
    width: 0%;
  }
  body {
    height: 2000px;
  }
</style>
<div class="progress"><div></div></div>
<p>Scroll me...</p>
```

```
<script>
  var bar = document.querySelector(".progress div");
  addEventListener("scroll", function() {
    var max = document.body.scrollHeight - innerHeight;
    var percent = (pageYOffset / max) * 100;
    bar.style.width = percent + "%";
  });
</script>
```

---

Assegnare a un elemento una posizione `fixed` (fissa) ha praticamente lo stesso effetto di una posizione `absolute` (assoluta), salvo che ne impedisce lo scorrimento col resto del documento. L'effetto che si ottiene è che la barra di avanzamento rimane ferma nel suo angolo. All'interno della barra si trova un altro elemento, che viene ridimensionato per indicare l'avanzamento corrente. Nell'impostare la larghezza dell'elemento, usiamo come unità di misura valori percentuali invece che in pixel, in modo che l'elemento sia ridimensionato in rapporto alla barra intera.

La variabile interna `innerHeight` ci dà l'altezza della finestra, che dobbiamo sottrarre dall'altezza di avanzamento totale; questo perché non potete continuare ad avanzare una volta raggiunto il fondo del documento (esiste anche una variabile `innerWidth` analoga). Dividendo la posizione di scorrimento corrente, `pageYOffset`, per la posizione di avanzamento massima e moltiplicando il risultato per 100, otteniamo la percentuale per la barra di avanzamento.

Richiamare `preventDefault` su un evento `scroll` non impedisce l'avanzamento: in effetti, il gestore dell'evento viene richiamato solo dopo che l'avanzamento è avvenuto.

## Eventi focus

Quando un elemento diventa attivo, il browser fa scattare un evento "focus".

Quando non è più attivo, scatta un evento "blur".

A differenza degli eventi discussi fin qui, questi due eventi non si propagano. I gestori degli elementi padri non sono notificati quando un loro elemento figlio diventa attivo o smette di esserlo.

L'esempio seguente mostra come visualizzare del testo come suggerimento di compilazione per il campo che risulta attivo:

```
<p>Name: <input type="text" data-help="Your full name"></p>
<p>Age: <input type="text" data-help="Age in years"></p>
<p id="help"></p>
<script>
  var help = document.querySelector("#help");
```

```

var fields = document.querySelectorAll("input");
for (var i = 0; i < fields.length; i++) {
    fields[i].addEventListener("focus", function(event) {
        var text = event.target.getAttribute("data-help");
        help.textContent = text;
    });
    fields[i].addEventListener("blur", function(event) {
        help.textContent = "";
    });
}
</script>

```

---

La schermata seguente mostra il suggerimento per il campo Age (età):

Name:	<input type="text" value="Hieronimus"/>
Age:	<input type="text" value=""/>
<b>Age in years</b>	

L'oggetto window riceve eventi "focus" e "blur" quando l'utente si sposta nella o via dalla finestra o dalla scheda del browser che mostra il documento.

## Eventi load

Quando una pagina finisce di caricare, scatta l'evento "load" sugli oggetti finestra e corpo del documento. Questi eventi vengono usati di solito per programmare l'inizializzazione di azioni che richiedono che tutto il documento sia stato visualizzato. Ricorderete che il contenuto dei tag `<script>` viene eseguito immediatamente, appena il browser incontra il tag. Spesso, questo non è il momento giusto, specialmente quando lo script deve intervenire su parti del documento che appaiono dopo il tag `<script>`.

Anche gli elementi che caricano un file esterno, come le immagini e certi script, hanno un evento "load" che indica quando i file relativi sono stati caricati. Come gli eventi focus, gli eventi load non si propagano.

Quando una pagina viene chiusa o superata (per esempio, seguendo un link), scatta un evento "beforeunload". Il suo impiego principale è nell'evitare che l'utente perda del lavoro chiudendo un documento per sbaglio. Diversamente da quel che potreste pensare, impedire la chiusura della pagina non si può fare col metodo `preventDefault`: bisogna invece che il gestore restituisca una stringa. La stringa sarà usata nella finestra di dialogo che chiede all'utente se è sicuro di voler lasciare la pagina.

# Tempi di esecuzione degli script

Ci sono diverse cose che possono far scattare l'esecuzione di uno script: una è quando il browser incontra un tag `<script>`, un'altra quando scatta un evento. Nel [Capitolo 13](#) abbiamo parlato della funzione `requestAnimationFrame`, che programma l'esecuzione di una funzione da richiamare prima di ritracciare la pagina: anche quello è un modo per far partire l'esecuzione di uno script.

È importante capire che anche se gli eventi possono scattare in qualunque momento, l'esecuzione contemporanea di due script nello stesso documento non è *mai* possibile. Se uno script è già in esecuzione, i gestori degli eventi e i brani di codice che devono intervenire in altro modo devono aspettare il loro turno. Ecco perché i documenti si bloccano quando ci sono script con tempi di esecuzione lunghi: il browser non può reagire ai clic e agli altri eventi all'interno del documento, perché non è in grado di eseguire i gestori degli eventi fintanto che lo script corrente non ha finito.

Alcuni ambienti di programmazione permettono l'esecuzione contemporanea di *thread* separati. Poter eseguire contemporaneamente più azioni può in effetti aumentare la velocità di un programma. Quando, però, ci sono azioni diverse che intervengono contemporaneamente sulle stesse parti del sistema, intervenire con un programma diventa molto, molto più complicato.

Il fatto che i programmi in JavaScript svolgano una sola azione per volta ci semplifica la vita. Quando avete veramente bisogno di svolgere qualche operazione complicata in sottofondo e non volete che la pagina si blocchi, i browser offrono delle funzionalità chiamate *web worker*. Un web worker è un ambiente JavaScript isolato, che viene eseguito in parallelo al programma principale per il documento e che può comunicare con esso solo mandando e ricevendo messaggi.

Immaginate di avere il codice seguente in un file, chiamato `code/squareworker.js`:

---

```
addEventListener("message", function(event) {
  postMessage(event.data * event.data);
});
```

---

Immaginate che calcolare il quadrato di un numero richieda calcoli lunghi e impegnativi, che vogliamo svolgere in un thread in secondo piano. Il codice seguente genera un web worker, gli trasmette alcuni messaggi e stampa le risposte:

---

```
var squareWorker = new Worker("code/squareworker.js");
squareWorker.addEventListener("message", function(event) {
  console.log("The worker responded:", event.data);
});
squareWorker.postMessage(10);
squareWorker.postMessage(24);
```

---

La funzione `postMessage` invia un messaggio, che fa scattare un evento "message" al codice ricevente. Lo script che ha creato il web worker invia e riceve messaggi attraverso l'oggetto `worker`, mentre il web worker parla allo script che l'ha creato e rimane in ascolto direttamente sul proprio ambito globale, che è un *nuovo* ambito globale, non condiviso con lo script originario.

## Impostare dei contatori

La funzione `setTimeout` è simile a `requestAnimationFrame`, in quanto programma l'esecuzione di un'altra funzione. Invece di richiamare la funzione quando lo schermo viene ritracciato, rimane in attesa per un determinato numero di millisecondi. Questa pagina cambia colore da blu a giallo dopo due secondi:

---

```
<script>
  document.body.style.background = "blue";
  setTimeout(function() {
    document.body.style.background = "yellow";
  }, 2000);
</script>
```

---

A volte c'è bisogno di annullare una funzione che avevate programmato. Per questo, dovete recuperare il valore restituito da `setTimeout` e richiamarvi sopra `clearTimeout`.

```
var bombTimer = setTimeout(function() {
  console.log("BOOM!");
}, 500);
if (Math.random() < 0.5) { // 50% chance
  console.log("Defused.");
  clearTimeout(bombTimer);
}
```

---

La funzione `cancelAnimationFrame` opera in modo simile a `clearTimeout`, e richiamarla su un valore restituito da `requestAnimationFrame` annulla quel fotogramma (sempre che non sia già stato richiamato).

Per impostare gli intervalli di tempo in modo che si ripetano ogni *X* millisecondi si usano due funzioni simili, `setInterval` e `clearInterval`.

---

```
var ticks = 0;
var clock = setInterval(function() {
  console.log("tick", ticks++);
  if (ticks == 10) {
```

```
    clearInterval(clock);
    console.log("stop.");
}
}, 200);
```

---

## Debouncing

Alcuni tipi di eventi possono scattare velocemente, molte volte di fila (per esempio, gli eventi "mousemove" e "scroll"). Nel gestire eventi di quel tipo, dovete fare attenzione a non svolgere operazioni che richiedano troppo tempo, se volete evitare che l'interazione del gestore col documento rallenti o proceda a scatti.

Se in quel gestore dovete svolgere operazioni meno che triviali, potete usare `setTimeout` per essere sicuri di non esagerare. A questo ci si riferisce col termine *debouncing* (lett.: antirimbalzo). Esistono diversi approcci a questa soluzione.

Nel primo esempio, vogliamo svolgere una qualche azione quando l'utente ha scritto qualcosa, ma non vogliamo farlo immediatamente dopo ogni evento key. Se scrivono velocemente, vogliamo aspettare fino alla prima pausa. Invece di svolgere immediatamente un'azione nel gestore degli eventi, impostiamo pertanto un periodo di timeout. Ripuliamo anche eventuali periodi di timeout preesistenti, in modo che quando gli eventi si succedono a breve distanza di tempo (ossia, meno del nostro periodo di timeout), venga eliminato il timeout dell'evento precedente.

---

```
<textarea>Type something here...</textarea>
<script>
  var textarea = document.querySelector("textarea");
  var timeout;
  textarea.addEventListener("keydown", function() {
    clearTimeout(timeout);
    timeout = setTimeout(function() {
      console.log("You stopped typing.");
    }, 500);
  });
</script>
```

---

Assegnare un valore `undefined` a `clearTimeout` o richiamarlo su un timeout già scattato non ha nessun effetto. Pertanto, non dobbiamo preoccuparci di quando richiamiamo la funzione e lo facciamo per tutti gli eventi.

Possiamo usare una struttura leggermente diversa se vogliamo distribuire le risposte lasciando tra l'una e l'altra un certo intervallo, ma facendole scattare durante una serie di

eventi e non dopo. Per esempio, questo può succedere se volete rispondere a eventi "mousemove" per dare le coordinate correnti del mouse, ma solo ogni 250 millisecondi.

---

```
<script>
    function displayCoords(event) {
        document.body.textContent =
            "Mouse at " + event.pageX + ", " + event.pageY;
    }
    var scheduled = false, lastEvent;
    addEventListener("mousemove", function(event) {
        lastEvent = event;
        if (!scheduled) {
            scheduled = true;
            setTimeout(function() {
                scheduled = false;
                displayCoords(lastEvent);
            }, 250);
        }
    });
</script>
```

---

## Riepilogo

I gestori degli eventi permettono di rilevare e reagire agli eventi su cui non abbiamo controllo. Per registrare queste funzioni di gestione, si usa il metodo `addEventListener`.

Gli eventi hanno un tipo che li identifica ("keydown", "focus" e così via). In genere, gli eventi scattano su elementi specifici del DOM e *si propagano* ai relativi antenati, passando il controllo ai gestori degli elementi associati a questi ultimi.

Quando viene chiamato, il gestore di un evento riceve un oggetto `event` con informazioni aggiuntive sull'evento. Quest'oggetto offre anche metodi che ci permettono di interrompere la propagazione (`stopPropagation`) e impedire il comportamento predefinito del browser per quell'evento (`preventDefault`).

Premere i tasti fa scattare eventi "keydown", "keypress" e "keyup". Premere un pulsante del mouse fa scattare eventi "mousedown", "mouseup" e "click". Spostare il mouse fa scattare "mousemove" e può far scattare eventi "mouseenter" e "mouseout".

Lo scorrimento si rileva attraverso l'evento "scroll" e il passaggio da un elemento attivo a un altro con "focus" e "blur". Quando il documento ha finito di caricare, sulla finestra scatta un evento "load".

Un solo programma JavaScript per volta può essere in esecuzione. Pertanto, i gestori degli eventi e gli altri script previsti devono attendere il loro turno, quando avrà finito lo script in esecuzione.

## Esercizi

### ***La tastiera censurata***

Tra il 1928 e il 2013, la legge turca proibiva l'uso delle lettere Q, W e X nei documenti ufficiali. Ciò faceva parte di un'iniziativa più vasta tendente a soffocare la cultura Curda, in quanto quelle lettere compaiono nel linguaggio curdo, ma non nel turco di Istanbul.

Come esercizio di stupidità tecnologica, vi invito a programmare un campo di testo (un tag `<input type="text">`) dove non si possono digitare quelle lettere.

Non preoccupatevi di copia e incolla o altri trucchi.

### ***La scia del mouse***

Agli albori di JavaScript, quando regnavano le pagine Web piene di colori e di animazioni, la gente aveva scovato modi davvero originali di usare il linguaggio.

Uno di questi erano le “scie del mouse”, una serie di immagini che seguivano il puntatore del mouse man mano che lo si spostava lungo una pagina.

In quest'esercizio, vi chiedo di impostare una scia del mouse. Usate elementi `<div>` con posizionamento assoluto, dimensioni fisse e con un colore di sfondo (fate riferimento al codice nel paragrafo *Clic del mouse* per un esempio). Create tutta una serie di questi elementi e, quando il mouse si sposta, fate in modo che il browser li visualizzi come uno strascico del puntatore.

Ci sono diversi modi di affrontare questo problema. Potete scegliere una soluzione semplice o una complicata, non importa quanto. Una soluzione semplice, tanto per cominciare, è di mantenere un numero fisso di elementi-scia e farli passare in ciclo, spostando il successivo alla posizione corrente del mouse ogni volta che scatta un evento "mousemove".

## ***Schede***

L'interfaccia a schede è un elemento di design comune: permette di selezionare una scheda (un pannello dell'interfaccia) facendo clic su una delle lingue che “sporgono” sopra un elemento.

In questo esercizio imposterete una semplice interfaccia a schede. Scrivete una funzione, `asTabs`, che accetta un nodo del DOM e crea un'interfaccia a schede che riporta gli elementi figli di quel nodo. Dovrete inserire un elenco di elementi `<button>` in cima al nodo, uno per ciascun elemento, contenenti il testo recuperato dall'attributo `data-tabname` del nodo figlio. Tutti i figli, tranne uno, devono essere nascosti (con `display:none`

definito nello stile) e il nodo visibile può essere selezionato facendo clic sui pulsanti.

Quando il programma funziona, estendetelo in modo da assegnare uno stile diverso al pulsante attivo.

## PROGETTO: UN VIDEOGIOCO A PIATTAFORME

*Tutta la realtà è un gioco.*

Iain Banks, *L'impero di Azad*

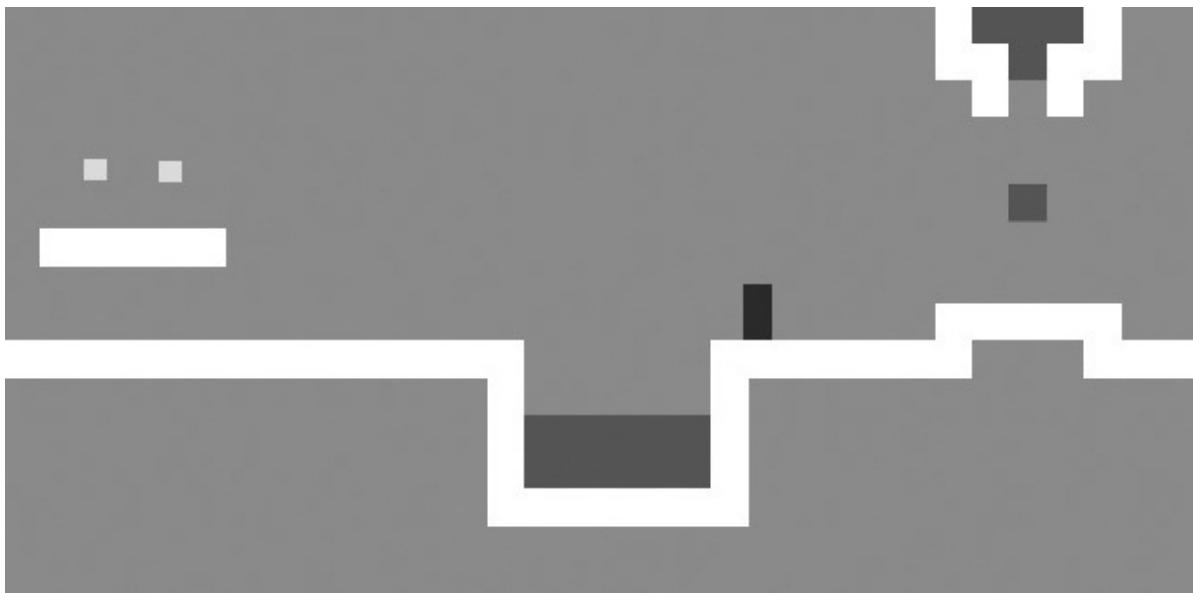
La mia passione iniziale per i computer, come quella di tanti ragazzi, nacque coi videogiochi. Mi lasciavo catturare dai microscopici mondi di simulazioni che potevo manipolare e nei quali si svolgevano (più o meno) delle storie, direi molto più per come riuscivo a proiettare la mia immaginazione su di essi, che per le possibilità pratiche che questi potevano offrirmi.

Non augurerrei a nessuno una carriera nella programmazione di videogiochi. Come per l'industria musicale, la discrepanza tra la quantità di giovani baldanzosi che vorrebbero lavorarci e la richiesta effettiva di forza lavoro crea un ambiente decisamente poco sano. Scrivere, invece, videogiochi per divertimento è bello.

In questo capitolo descrivo la realizzazione di un semplice videogioco a piattaforme. Nei giochi a piattaforme (detti anche “jump and run”, salta e corri) il giocatore deve far muovere una figurina in un mondo, spesso bidimensionale e visto di lato, e fare un sacco di salti sopra e oltre i vari ostacoli.

### Il gioco

Il nostro gioco prenderà spunto da *Dark Blue* di Thomas Palef, [www.lesmilk.com/games/10](http://www.lesmilk.com/games/10). Ho scelto questo gioco perché è nel contempo divertente e minimalista, e perché si può realizzare senza dover scrivere troppo codice. Eccone una schermata:



Il riquadro scuro rappresenta il giocatore, che deve raccogliere i riquadri gialli (monete) evitando di cadere nella roba rossa, che io chiamo lava. Si completa un livello quando tutte le monete sono state raccolte.

Il giocatore si sposta premendo i tasti freccia destra e freccia sinistra, e salta premendo la freccia in su. Il salto è una caratteristica di questo personaggio: può saltare molto in alto, diverse volte la sua altezza, e cambiare direzione a mezz'aria. Non è molto realistico, ma spero dia a chi gioca la sensazione di avere il controllo assoluto dell'avatar (sullo schermo).

Il gioco consiste in uno sfondo fisso, organizzato come una griglia, con gli elementi che si spostano poggiati sopra quello sfondo. I campi della griglia possono essere vuoti, pieni o di lava. Gli elementi che si spostano sono il giocatore, le monete e certi pezzi di lava. Diversamente dalla simulazione di vita artificiale del [Capitolo 7](#), le posizioni di questi elementi non sono rigidamente legate alla griglia: le loro coordinate possono avere valori decimali, il che assicura fluidità di movimento.

## La tecnologia

Useremo il DOM del browser per visualizzare il gioco e leggeremo l'input utente attraverso gestori degli eventi tasti.

Il codice per lo schermo e la tastiera è solo una minima parte del lavoro che dobbiamo svolgere per realizzare il gioco. Poiché tutti gli oggetti sono riquadri colorati, l'aspetto del disegno è semplicissimo: creiamo elementi DOM e usiamo gli stili per assegnare loro colori di sfondo, dimensioni e posizione.

Possiamo rappresentare lo sfondo come una tabella, in quanto è una griglia immutabile di caselle. Gli elementi che si spostano liberamente possono essere “appoggiati” sopra la griglia attraverso il posizionamento assoluto.

Nei giochi, e negli altri programmi che hanno animazioni grafiche e rispondono all'input utente senza apparente ritardo, conta molto l'efficienza. Sebbene non sia stato

progettato per funzioni grafiche ad alto rendimento, il DOM è però migliore di quel che potreste pensare. Avete visto qualche animazione nel [Capitolo 13](#). Su una macchina moderna, un gioco semplice come questo dà buone prestazioni, senza bisogno di preoccuparsi troppo dell'ottimizzazione.

Nel prossimo capitolo, esploreremo un'altra tecnologia del browser, il tag <canvas>, che offre un modo più tradizionale per disegnare, lavorando con forme e pixel invece che con gli elementi del DOM.

## Livelli

Nel [Capitolo 7](#) abbiamo usato degli array di stringhe per descrivere una griglia bidimensionale. Possiamo applicare anche qui lo stesso schema, in modo da poter progettare dei livelli senza bisogno di impostare un editor specifico.

Un livello semplice sarà come il seguente:

---

```
var simpleLevelPlan = [
    "                 ",",
    "                 ",",
    " x             = x ",
    " x     o o     x ",
    " x @      XXXXX x ",
    " XXXXX           x ",
    " x!!!!!!x!!!!x ",
    " XXXXXXXXXXXXXXXX ";
];
```

---

Il piano comprende sia la griglia fissa, sia gli elementi mobili. I caratteri x indicano i muri, gli spazi stanno per lo spazio vuoto e i punti esclamativi rappresentano mattoni di lava fissi, che non si spostano.

Il carattere @ definisce il punto di partenza del giocatore. Gli o sono monete, e l'uguale (=) indica un blocco di lava che si sposta orizzontalmente, avanti e indietro. Notate che la griglia per quelle posizioni sarà impostata in modo da contenere dello spazio vuoto, mentre un'altra struttura dati servirà a mantenere la posizione degli elementi mobili.

Daremo poi supporto ad altri due tipi di lava che si sposta: il carattere |), che indica i blocchi che si spostano in verticale, e la v per la lava che cola, che si sposta verticalmente senza rimbalzare avanti e indietro, in quanto torna semplicemente alla posizione di partenza quando arriva a toccare il pavimento.

Il gioco nel suo insieme consiste di più livelli, che il giocatore deve completare. Il livello si completa quando abbiamo raccolto tutte le monete. Se il giocatore tocca la lava, il livello in corso torna alla posizione iniziale e si riprova.

## Leggere un livello

Il seguente costruttore imposta un oggetto `level`. Il suo argomento è l'array di stringhe che definiscono il livello.

---

```
function Level(plan) {  
    this.width = plan[0].length;  
    this.height = plan.length;  
    this.grid = [];  
    this.actors = [];  
    for (var y = 0; y < this.height; y++) {  
        var line = plan[y], gridLine = [];  
        for (var x = 0; x < this.width; x++) {  
            var ch = line[x], fieldType = null;  
            var Actor = actorChars[ch];  
            if (Actor)  
                this.actors.push(new Actor(new Vector(x, y), ch));  
            else if (ch == "x")  
                fieldType = "wall";  
            else if (ch == "!")  
                fieldType = "lava";  
            gridLine.push(fieldType);  
        }  
        this.grid.push(gridLine);  
    }  
    this.player = this.actors.filter(function(actor) {  
        return actor.type == "player";  
    })[0];  
    this.status = this.finishDelay = null;  
}
```

---

Per brevità, il codice non controlla se l'input è malformato: presume che gli abbiate passato un piano di livello appropriato, completo di posizione del giocatore all'inizio e altre informazioni essenziali.

Di ogni livello si memorizzano larghezza e altezza, insieme a due array: uno per la griglia e uno per gli elementi `actors`, ossia gli elementi dinamici, che d'ora in avanti chiameremo *attori*. La griglia viene rappresentata da un array di array, dove ciascuno degli array interni rappresenta una riga orizzontale e ogni quadrato contiene o `null`, per quelli vuoti, o una stringa che indica il tipo di ostacolo ("wall" o "lava").

L'array degli attori mantiene oggetti che seguono la posizione corrente e lo stato degli elementi dinamici del livello. Ciascuno di essi ha una proprietà `pos`, che ne indica la posizione (come coordinate del rispettivo angolo superiore sinistro), una proprietà `size` per le dimensioni e una proprietà `type` che contiene una stringa che identifica l'elemento ("lava", "coin" o "player").

Dopo aver costruito la griglia, usiamo il metodo `filter` per trovare l'oggetto attore del giocatore (`player`), che memorizziamo in una proprietà del livello. La proprietà `status` tiene in memoria se il giocatore ha vinto o perso. Quando si verifica una di queste situazioni, usiamo `finishDelay` per mantenere attivo il livello per un breve periodo di tempo e mostrare una semplice animazione (ripartire da capo o passare di livello automaticamente non è elegante). Con questo metodo possiamo sapere se un livello è terminato:

---

```
Level.prototype.isFinished = function() {
    return this.status != null && this.finishDelay < 0;
};
```

---

## Gli attori

Per memorizzare posizione e dimensioni di un elemento dinamico, torniamo al nostro fidato tipo `Vector`, che combina una coppia di coordinate `x` e `y` in un oggetto.

---

```
function Vector(x, y) {
    this.x = x; this.y = y;
}
Vector.prototype.plus = function(other) {
    return new Vector(this.x + other.x, this.y + other.y);
};
Vector.prototype.times = function(factor) {
    return new Vector(this.x * factor, this.y * factor);
};
```

---

Il metodo `times` ridimensiona un vettore di un determinato fattore. Tornerà utile quando dovremo moltiplicare un vettore velocità per un intervallo di tempo, per trovare la distanza coperta in quell'intervallo.

Nel paragrafo precedente, il costruttore `Level` usa l'oggetto `actorChars` per associare i caratteri a funzioni di costruzione. Ecco il codice dell'oggetto:

---

```
var actorChars = {
    "@": Player,
```

```
"o": Coin,  
"=". Lava, "|": Lava, "v": Lava  
};
```

---

L'elemento Lava è mappato con tre caratteri. Il costruttore Level passa il carattere sorgente dell'elemento dinamico come secondo argomento del suo costruttore; il costruttore Lava usa quell'argomento per stabilire il proprio comportamento (rimbalzo orizzontale, rimbalzo verticale o caduta).

Il tipo del giocatore, player, si imposta col costruttore che segue. Ha una proprietà speed che mantiene la sua velocità corrente, che servirà a simulare lo slancio e la gravità.

---

```
function Player(pos) {  
    this.pos = pos.plus(new Vector(0, -0.5));  
    this.size = new Vector(0.8, 1.5);  
    this.speed = new Vector(0, 0);  
}  
  
Player.prototype.type = "player";
```

---

Poiché il giocatore è alto una casella e mezza, la sua posizione iniziale è impostata su mezza casella sopra la posizione dove si trova il carattere @, in modo che i due elementi siano allineati in basso.

Per costruire un oggetto Lava dinamico, dobbiamo inizializzare l'oggetto in modo diverso a seconda del carattere che lo contraddistingue. La lava dinamica si sposta alla velocità data finché non incontra un ostacolo. A quel punto, se ha una proprietà repeatPos, torna indietro di colpo alla posizione iniziale e questo movimento corrisponde alla caduta. Diversamente, inverte la velocità e continua nella direzione opposta. Il costruttore impone solo le proprietà necessarie. Scriveremo più avanti il metodo che compie il movimento.

---

```
function Lava(pos, ch) {  
    this.pos = pos;  
    this.size = new Vector(1, 1);  
    if (ch == "=") {  
        this.speed = new Vector(2, 0);  
    } else if (ch == "|") {  
        this.speed = new Vector(0, 2);  
    } else if (ch == "v") {  
        this.speed = new Vector(0, 3);  
        this.repeatPos = pos;  
    }  
}
```

```
Lava.prototype.type = "lava";
```

---

Gli attori moneta (Coin) sono semplici, in quanto stanno fermi dove sono. Ma per movimentare un pochino il gioco, daremo loro un effetto di ondulazione, con piccoli movimenti verticali in su e in giù. Per questo, l'oggetto Coin memorizza una posizione base e una proprietà wobble, che segue la fase del moto. Insieme, questi dati determinano la posizione attuale della moneta, memorizzata nella proprietà pos.

---

```
function Coin(pos) {  
    this.basePos = this.pos = pos.plus(new Vector(0.2, 0.1));  
    this.size = new Vector(0.6, 0.6);  
    this.wobble = Math.random() * Math.PI * 2;  
}  
  
Coin.prototype.type = "coin";
```

---

Nel [Capitolo 13](#), abbiamo visto che `Math.sin` ci dà la coordinata y di un punto su un cerchio. Quella coordinata va avanti e indietro, con un movimento a onda continua, man mano che ci spostiamo lungo il cerchio: questa funzione risulta pertanto utile per impostare un movimento ondeggiante.

Per evitare che tutte le monete si muovano contemporaneamente, si randomizza la fase iniziale di ogni moneta. La *fase* dell'onda di `Math.sin`, ossia l'ampiezza dell'onda prodotta, è  $2\pi$ . Moltiplichiamo pertanto il valore restituito da `Math.random` per quel numero, per dare alla moneta una posizione a caso sull'onda.

Con questo, abbiamo finito di impostare le parti che servono per rappresentare lo stato di un livello.

---

```
var simpleLevel = new Level(simpleLevelPlan);  
console.log(simpleLevel.width, "by", simpleLevel.height);  
// → 22 by 9
```

---

Passiamo ora a visualizzare i livelli sullo schermo e impostare al loro interno tempi e movimenti.

## Il fardello dell'incapsulazione

In generale, il codice di questo capitolo non si cura dell'incapsulazione per due ragioni. La prima è che l'incapsulazione richiede del lavoro in più. Aumenta le dimensioni dei programmi e richiede di introdurre altri concetti e interfacce. Poiché si può proporre al lettore solo una quantità limitata di codice prima che si stanchi, ho fatto uno sforzo per mantenere il programma più piccolo possibile.

In secondo luogo, gli elementi di questo gioco sono talmente legati fra loro che se il comportamento di uno di essi cambiasse, difficilmente gli altri potrebbero rimanere

costanti. Le interfacce tra elementi dovrebbero tradurre in codice troppi presupposti sul funzionamento del gioco. Ciò le renderebbe meno efficaci, in quanto ogni volta che cambiate una parte del sistema, dovrete andare a vedere come va a interferire con le altre parti, le cui interfacce magari non prevedono la nuova situazione.

Alcuni *punti di divisione* in un sistema si prestano bene a essere separati da interfacce rigorose: ma non è sempre così. Tentare di incapsulare qualcosa che non ha confini ben delimitati è un modo per sprecare una gran quantità di energie. Se commettete questo errore, troverete che le vostre interfacce tendono a diventare troppo grandi e dettagliate e richiedono modifiche frequenti, man mano che il programma si evolve.

C'è una sola cosa che incapsuleremo nel codice di questo capitolo: il sottosistema di disegno. La ragione di questo è che nel prossimo capitolo mostreremo lo stesso gioco con un aspetto diverso. Separando le funzioni grafiche con un'interfaccia, sarà sufficiente caricare lo stesso programma per il gioco e sostituire solo il modulo per la visualizzazione.

## Disegni

L'incapsulazione del codice per disegnare si ottiene definendo un oggetto *display*, che visualizza il livello dato. Il tipo che definiamo in questo capitolo per quell'oggetto si chiama **DOMDisplay**, in quanto utilizza semplici elementi del DOM per rendere il livello sullo schermo.

Useremo un foglio stile per impostare i colori e le proprietà invariabili degli elementi che compongono il gioco. Sarebbe possibile anche assegnare direttamente la proprietà *style* dei singoli elementi quando li creiamo, ma in quel modo il programma risulterebbe più lungo.

La seguente funzione ausiliaria offre una scorciatoia per creare un elemento e assegnargli una classe:

---

```
function elt(name, className) {
    var elt = document.createElement(name);
    if (className) elt.className = className;
    return elt;
}
```

---

L'oggetto *display* si crea assegnandogli un elemento genitore, al quale si attacca, e un oggetto *livello*.

---

```
function DOMDisplay(parent, level) {
    this.wrap = parent.appendChild(elt("div", "game"));
    this.level = level;
    this.wrap.appendChild(this.drawBackground());
    this.actorLayer = null;
```

```
this.drawFrame();
}
```

---

Per creare l'elemento che lo avvolge e memorizzarlo nella proprietà `wrap` con una sola dichiarazione, abbiamo approfittato del fatto che `appendChild` restituisce l'elemento definito per ultimo.

Lo sfondo del livello, che non cambia mai, viene tracciato solo una volta. Gli elementi mobili si ritracciano ogni volta che si aggiorna il display. La proprietà `actorLayer` viene usata da `drawFrame` per seguire l'elemento che contiene gli attori, in modo da poterli eliminare e sostituire con facilità.

Coordinate e dimensioni sono in unità di misura basate sulla griglia, dove un'unità di dimensione o di distanza corrisponde a 1 unità griglia. Quando impostiamo le dimensioni in pixel, dovremo moltiplicare queste coordinate, perché se ogni casella della griglia fosse di 1 pixel, gli elementi del gioco sarebbero piccolissimi. La variabile `scale` dà il numero di pixel di un'unità sullo schermo.

---

```
var scale = 20;
DOMDisplay.prototype.drawBackground = function() {
    var table = elt("table", "background");
    table.style.width = this.level.width * scale + "px";
    this.level.grid.forEach(function(row) {
        var rowElt = table.appendChild(elt("tr"));
        rowElt.style.height = scale + "px";
        row.forEach(function(type) {
            rowElt.appendChild(elt("td", type));
        });
    });
    return table;
};
```

---

Come ho accennato prima, lo sfondo viene tracciato come elemento `<table>`, cosa che corrisponde alla struttura della proprietà `grid` nel livello: ogni riga della griglia diventa un riga della tabella (elemento `<tr>`). Le stringhe all'interno della griglia vengono usate come nomi di classi per le celle della tabella (elementi `<td>`). Le regole definite nel foglio stile seguente assegnano i colori di sfondo, in modo da dare alla tabella l'aspetto che vogliamo:

```
.background { background: rgb(52, 166, 251);
              table-layout: fixed;
              border-spacing: 0; }
.background td { padding: 0; }
```

```
.lava { background: rgb(255, 100, 100); }
.wall { background: white; }
```

---

Alcune delle regole (quelle per `table-layout`, `border-spacing` e `padding`) servono solo per annullare comportamenti predefiniti non adatti: non vogliamo, infatti, che la struttura della tabella dipenda dal contenuto delle celle e non vogliamo spazio vuoto tra celle, né spazi di riempimento intorno al loro contenuto.

La regola (per la classe) `.background` imposta il colore di sfondo. Il CSS consente di rappresentare i colori sia in parole ("white" per bianco), sia in formato `rgb(R, G, B)` dove i componenti rosso (R), verde (G) e blu (B) del colore sono definiti ciascuno con valori da 0 a 255. Pertanto, `rgb(52, 166, 251)` indica che il componente rosso ha valore 52, il verde 166 e il blu 251. Poiché il componente blu è maggiore degli altri, il colore risultante tenderà al blu. Nella regola per `.lava`, vedete che il numero più alto è invece il rosso.

Tracciamo ora i singoli attori creando un elemento DOM per ciascuno e impostandone la posizione e le dimensioni in base alle proprietà dell'oggetto attore corrispondente. I valori vanno moltiplicati in scala per convertire in pixel le unità del gioco.

---

```
DOMDisplay.prototype.drawActors = function() {
  var wrap = elt("div");
  this.level.actors.forEach(function(actor) {
    var rect = wrap.appendChild(elt("div", "actor " + actor.type));
    rect.style.width = actor.size.x * scale + "px";
    rect.style.height = actor.size.y * scale + "px";
    rect.style.left = actor.pos.x * scale + "px";
    rect.style.top = actor.pos.y * scale + "px";
  });
  return wrap;
};
```

---

Per assegnare a un elemento più di una classe, separiamo con lo spazio i nomi delle classi. Nel codice CSS riportato più avanti, la classe `.actor` assegna ai singoli elementi posizione assoluta. Il nome del rispettivo tipo viene usato come classe aggiuntiva per assegnare a ciascuno il suo colore. Non dobbiamo ridefinire la classe `.lava` perché riutilizzeremo la classe per le caselle "lava" della griglia che abbiamo definito in precedenza.

---

```
.actor { position: absolute; }
.coin { background: rgb(241, 229, 89); }
.player { background: rgb(64, 64, 64); }
```

---

Quando aggiorna il display, il metodo `drawFrame` prima (se necessario) elimina la

grafica superata degli elementi dinamici e poi li ritraccia nelle nuove posizioni. Se avete la tentazione di riutilizzare gli elementi DOM per questo, pensateci bene: per poterlo fare bisogna, infatti, scambiare un sacco di informazioni tra il codice per la visualizzazione e quello per la simulazione. Dovremmo associare i singoli attori con elementi del DOM e il codice per disegnare il gioco dovrebbe eliminare gli elementi quando i relativi attori scompaiono. Visto che gli elementi dinamici sono pochi, ridisegnarli tutti è meno dispendioso.

---

```
DOMDisplay.prototype.drawFrame = function() {  
    if (this.actorLayer)  
        this.wrap.removeChild(this.actorLayer);  
    this.actorLayer = this.wrap.appendChild(this.drawActors());  
    this.wrap.className = "game " + (this.level.status || "");  
    this.scrollPlayerIntoView();  
};
```

---

Aggiungendo lo stato corrente del livello come nome di classe per l'elemento circoscritto, possiamo assegnare al giocatore uno stile leggermente diverso a seconda se vince o se perde, definendo una regola CSS che abbia effetto solo quando il giocatore ha un elemento padre di una determinata classe.

---

```
.lost .player {  
    background: rgb(160, 64, 64);  
}  
.won .player {  
    box-shadow: -4px -7px 8px white, 4px -7px 8px white;  
}
```

---

Se tocca la lava, il colore del giocatore diventa rosso scuro per suggerire l'idea che si sia scottato. Quando ha raccolto l'ultima moneta, usiamo due ombre ( box-shadow) bianche, una in alto a destra e una in alto a sinistra, per creare l'impressione di un'aureola.

Non possiamo presumere che i livelli siano sempre visibili nella loro interezza nello schermo di destinazione. Ecco perché dobbiamo richiamare scrollPlayerIntoView, che assicura che se il livello non rientra nello schermo, lo facciamo scorrere finché il giocatore si trova più o meno nel centro. La regola CSS che segue assegna all'elemento circoscritto del DOM dei valori di dimensione massima e fa in modo che quel che non sta dentro la “scatola” dell'elemento non sia visibile. Assegniamo poi all'elemento esterno una posizione relativa, in modo che gli attori al suo interno siano posizionati rispetto all'angolo superiore sinistro del livello.

---

```
.game {  
    overflow: hidden;
```

```
max-width: 600px;  
max-height: 450px;  
position: relative;  
}
```

---

Nel metodo `scrollPlayerIntoView`, troviamo la posizione del giocatore e aggiorniamo il punto di scorrimento dell'elemento circoscritto. Modifichiamo la posizione di scorrimento manipolando le proprietà `scrollLeft` e `scrollTop` di quell'elemento quando il giocatore si avvicina troppo ai bordi dello schermo.

---

```
DOMDisplay.prototype.scrollPlayerIntoView = function() {  
    var width = this.wrap.clientWidth;  
    var height = this.wrap.clientHeight;  
    var margin = width / 3;  
    // The viewport  
    var left = this.wrap.scrollLeft, right = left + width;  
    var top = this.wrap.scrollTop, bottom = top + height;  
    var player = this.level.player;  
    var center = player.pos.plus(player.size.times(0.5)).times(scale);  
    if (center.x < left + margin)  
        this.wrap.scrollLeft = center.x - margin;  
    else if (center.x > right - margin)  
        this.wrap.scrollLeft = center.x + margin - width;  
    if (center.y < top + margin)  
        this.wrap.scrollTop = center.y - margin;  
    else if (center.y > bottom - margin)  
        this.wrap.scrollTop = center.y + margin - height;  
};
```

---

Il sistema per trovare il centro del giocatore mostra come i metodi del tipo `Vector` permettono di scrivere in maniera leggibile i calcoli per gli oggetti. Per trovare il centro del giocatore, sommiamo la sua posizione (l'angolo superiore sinistro) a metà delle sue dimensioni. Otteniamo così il centro in coordinate di livello, ma poiché ci servono le coordinate in pixel, dobbiamo moltiplicare il vettore risultante per il valore di ridimensionamento.

Segue poi una serie di verifiche sulla posizione del giocatore, per essere certi che non sia al di fuori dell'intervallo valido. Notate che a volte questo codice imposta delle coordinate insensate, con valori inferiori a zero o al di fuori dell'area disponibile per l'elemento. Non preoccupatevi: il modello DOM le riporterà entro valori accettabili. Impostare `scrollLeft` su -10 lo reimposta su 0.

Sarebbe stato un pochino più semplice riportare sempre il giocatore al centro dello schermo. Questo, però, ha un effetto fastidioso, perché se si sta saltando, la finestra continua a spostarsi in su e in giù. Meglio avere un'area “neutra” nel centro dello schermo, dove ci si può spostare liberamente senza bisogno di far scorrere lo schermo.

Infine, ci serve un modo per ripulire un livello visualizzato, da usare quando il gioco passa al livello successivo o riparte dall'inizio del livello.

---

```
DOMDisplay.prototype.clear = function() {  
    this.wrap.parentNode.removeChild(this.wrap);  
};
```

---

Possiamo ora visualizzare il nostro livello.

---

```
<link rel="stylesheet" href="css/game.css">  
<script>  
    var simpleLevel = new Level(simpleLevelPlan);  
    var display = new DOMDisplay(document.body, simpleLevel);  
</script>
```

---



Il tag `<link>` con l'attributo `rel="stylesheet"` è un modo per caricare un file CSS in una pagina. Il file `game.css` contiene gli stili necessari per il gioco.

## Movimenti e collisioni

A questo punto possiamo cominciare ad aggiungere i movimenti, l'aspetto più interessante del gioco. La soluzione più elementare seguita da giochi come questo è di suddividere il tempo in intervalli brevi e, per ogni intervallo, spostare gli attori di una distanza corrispondente alla loro velocità (distanza percorsa al secondo) moltiplicata per le dimensioni dell'intervallo (in secondi).

Niente di difficile fin qui. La complicazione sta nel gestire le interazioni tra elementi. Quando il giocatore va a sbattere contro un muro o contro il pavimento, non può passarci attraverso. Il programma deve rilevare quando un certo movimento fa sì che un oggetto ne colpisca un altro e rispondere in maniera adeguata. Nel caso dei muri, lo spostamento va interrotto. Per le monete, questa va raccolta e così via.

Risolvere il problema in generale è un grosso lavoro. Si trovano librerie, chiamate *motori fisici* (physics engines), che simulano le interazioni tra oggetti fisici in due o tre dimensioni. Noi qui scegliamo un’alternativa più modesta e ci limitiamo a gestire, in maniera abbastanza primitiva, le collisioni tra oggetti rettangolari.

Prima di spostare il giocatore o un blocco di lava, verifichiamo se lo spostamento li porterebbe all’interno di una parte dello sfondo che non è vuota. In questo caso, annulliamo il movimento. La risposta a questo tipo di collisione dipende dall’attore: il giocatore si ferma, mentre i blocchi di lava rimbalzano.

Questa soluzione richiede intervalli di tempo piuttosto piccoli, perché il movimento si deve interrompere appena prima che gli oggetti si tocchino. Se gli intervalli (e pertanto gli spostamenti) sono troppo grandi, può sembrare che il giocatore “leviti” a una certa distanza dal terreno. Un’altra soluzione, forse migliore, ma più complicata, è di trovare il punto esatto della collisione e spostarsi lì. Scegliamo la soluzione più semplice e ne nascondiamo i problemi facendo in modo che l’animazione si muova a piccoli passi.

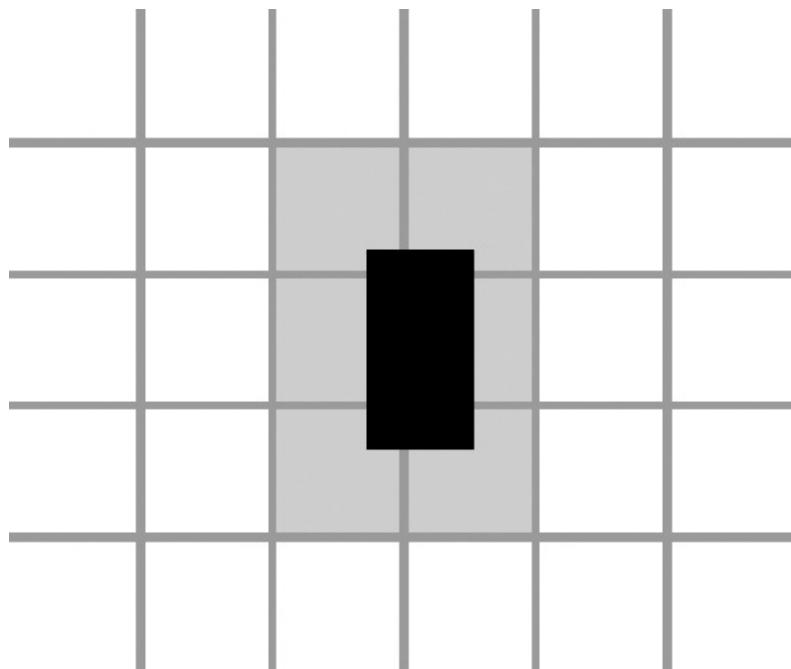
Questo metodo indica se un rettangolo (definito da una posizione e da due dimensioni) si sovrappone a dello spazio non vuoto, sulla griglia di sfondo:

---

```
Level.prototype.obstacleAt = function(pos, size) {
    var xStart = Math.floor(pos.x);
    var xEnd = Math.ceil(pos.x + size.x);
    var yStart = Math.floor(pos.y);
    var yEnd = Math.ceil(pos.y + size.y);
    if (xStart < 0 || xEnd > this.width || yStart < 0)
        return "wall";
    if (yEnd > this.height)
        return "lava";
    for (var y = yStart; y < yEnd; y++) {
        for (var x = xStart; x < xEnd; x++) {
            var fieldType = this.grid[y][x];
            if (fieldType) return fieldType;
        }
    }
};
```

---

Il metodo calcola le caselle della griglia che si sovrappongono al giocatore applicando `Math.floor` e `Math.ceil` sulle sue coordinate. Ricorderete che le caselle della griglia sono di 1x1 unità. Arrotondando le dimensioni della casella (che rappresenta il giocatore) in più e in meno, otteniamo la serie di caselle dello sfondo che la casella tocca.



Se il corpo sporge dal livello, restituiamo "wall" per lo spazio sopra e ai lati, e "lava" per lo spazio in basso. In questo modo, il giocatore muore quando cade fuori dal mondo. Quando il corpo si trova tutto all'interno della griglia, passiamo in ciclo le coordinate di caselle della griglia trovate arrotondando le coordinate e restituendo il contenuto della prima casella non vuota che troviamo.

Le collisioni tra il giocatore e gli altri elementi dinamici (monete, lava che si muove) sono gestiti dopo che il giocatore si è spostato. Quando il movimento lo porta a toccare un altro attore, viene attivato l'effetto corrispondente: raccoglie una moneta o muore.

Questo metodo passa in rassegna l'array di attori, in cerca di uno che si sovrappone a quello dato come argomento:

---

```
Level.prototype.actorAt = function(actor) {
    for (var i = 0; i < this.actors.length; i++) {
        var other = this.actors[i];
        if (other != actor &&
            actor.pos.x + actor.size.x > other.pos.x &&
            actor.pos.x < other.pos.x + other.size.x &&
            actor.pos.y + actor.size.y > other.pos.y &&
            actor.pos.y < other.pos.y + other.size.y)
            return other;
    }
};
```

---

## Attori e azioni

Il metodo `animate` sul tipo `Level` dà a tutti gli attori del livello la possibilità di muoversi. Il suo argomento `step` è l'intervallo di tempo in secondi. L'oggetto `keys` contiene informazioni sui tasti freccia che sono stati premuti.

---

```
var maxStep = 0.05;

Level.prototype.animate = function(step, keys) {
    if (this.status != null)
        this.finishDelay -= step;
    while (step > 0) {
        var thisStep = Math.min(step, maxStep);
        this.actors.forEach(function(actor) {
            actor.act(thisStep, this, keys);
        }, this);
        step -= thisStep;
    }
};
```

Quando la proprietà `status` del livello ha un valore diverso da `null` (il che succede quando il giocatore ha vinto o perso), dobbiamo fare il conteggio alla rovescia con la proprietà `finishDelay`, che tiene nota del tempo che passa tra il momento della vittoria o della sconfitta e il punto dove vogliamo smettere di visualizzare il livello.

Il ciclo `while` frammenta l'intervallo di spostamento dell'animazione in porzioni sufficientemente piccole, facendo in modo che nessuno spostamento sia superiore a `maxStep`. Per esempio, un intervallo di 0,12 secondi viene diviso in due intervalli di 0,05 secondi e uno di 0,02.

Gli oggetti attore hanno un metodo `act`, che accetta come argomenti l'intervallo di tempo, l'oggetto livello e l'oggetto `keys`. Ecco quello per il tipo `Lava`, che ignora l'oggetto `keys`:

---

```
Lava.prototype.act = function(step, level) {
    var newPos = this.pos.plus(this.speed.times(step));
    if (!level.obstacleAt(newPos, this.size))
        this.pos = newPos;
    else if (this.repeatPos)
        this.pos = this.repeatPos;
    else
        this.speed = this.speed.times(-1);
};
```

Il metodo calcola una nuova posizione sommando il prodotto dell'intervallo e della

sua velocità corrente con la sua posizione precedente. Se non ci sono ostacoli nella posizione così calcolata, si sposta lì. Se trova un ostacolo, il comportamento dipende dal tipo di oggetto Lava; la lava che cola ha una proprietà `repeatPos`, alla quale ritorna quando colpisce qualcosa. La lava che ribolle invece inverte la propria velocità (moltiplicandola per -1) per cominciare a spostarsi nella direzione opposta.

Le monete usano il loro metodo `act` per oscillare. Ignorano le collisioni, in quanto si limitano a tremolare dentro il loro riquadro, e le collisioni col giocatore sono gestite dal metodo `act` di quest'ultimo.

---

```
var wobbleSpeed = 8, wobbleDist = 0.07;
Coin.prototype.act = function(step) {
    this.wobble += step * wobbleSpeed;
    var wobblePos = Math.sin(this.wobble) * wobbleDist;
    this.pos = this.basePos.plus(new Vector(0, wobblePos));
};
```

---

Si aggiorna la proprietà `wobble` per tenere nota del tempo e passarla poi come argomento a `Math.sin` per creare un'onda, che viene poi usata per calcolare la nuova posizione.

Rimane a questo punto solo il giocatore. Il suo movimento viene gestito separatamente per ciascun asse, perché toccare il pavimento non deve impedire lo spostamento in orizzontale e toccare il muro non interrompe una caduta o un salto. Questo metodo imposta gli spostamenti orizzontali:

---

```
var playerXSpeed = 7;
Player.prototype.moveX = function(step, level, keys) {
    this.speed.x = 0;
    if (keys.left) this.speed.x -= playerXSpeed;
    if (keys.right) this.speed.x += playerXSpeed;
    var motion = new Vector(this.speed.x * step, 0);
    var newPos = this.pos.plus(motion);
    var obstacle = level.obstacleAt(newPos, this.size);
    if (obstacle)
        level.playerTouched(obstacle);
    else
        this.pos = newPos;
};
```

---

I movimenti orizzontali sono calcolati in base allo stato dei tasti freccia destra e sinistra. Quando un movimento porta il giocatore a colpire qualcosa, si richiama il metodo `playerTouched` del livello, che gestisce cose come la morte nella lava e la raccolta di

monete. Altrimenti, l'oggetto aggiorna la propria posizione.

Il moto verticale funziona in modo simile, ma deve simulare il salto e la forza di gravità.

---

```
var gravity = 30;
var jumpSpeed = 17;

Player.prototype.moveY = function(step, level, keys) {
    this.speed.y += step * gravity;
    var motion = new Vector(0, this.speed.y * step);
    var newPos = this.pos.plus(motion);
    var obstacle = level.obstacleAt(newPos, this.size);
    if (obstacle) {
        level.playerTouched(obstacle);
        if (keys.up && this.speed.y > 0)
            this.speed.y = -jumpSpeed;
        else
            this.speed.y = 0;
    } else {
        this.pos = newPos;
    }
};
```

---

All'inizio, la velocità del giocatore aumenta verticalmente per contrastare la forza di gravità. Quest'ultima, la velocità di salto e praticamente tutte le altre costanti del gioco sono state impostate per tentativi: ho provato diversi valori finché ho trovato una combinazione che mi andava bene.

Dopodiché, andiamo di nuovo a vedere se ci sono ostacoli. Se colpiamo un ostacolo, abbiamo due risultati possibili. Quando teniamo premuta la freccia in su e ci spostiamo verso il basso (e la cosa che andiamo a colpire è pertanto sotto di noi), la velocità è impostata su un valore negativo abbastanza grande: in questo caso, il giocatore salta. Altrimenti, vuol dire che siamo andati a sbattere da qualche altra parte e la velocità viene reimpostata su zero.

Ecco il codice per questo metodo:

---

```
Player.prototype.act = function(step, level, keys) {
    this.moveX(step, level, keys);
    this.moveY(step, level, keys);
    var otherActor = level.actorAt(this);
    if (otherActor)
```

```
    level.playerTouched(otherActor.type, otherActor);
// Losing animation
if (level.status == "lost") {
    this.pos.y += step;
    this.size.y -= step;
}
};
```

---

Dopo che ci siamo spostati, il metodo verifica se ci sono altri attori con i quali possiamo scontrarci e richiama di nuovo `playerTouched` quando ne trova uno. Questa volta, passa l'oggetto attore come secondo argomento, perché se l'altro attore è una moneta, `playerTouched` deve sapere quale va raccolta.

Infine, quando il giocatore muore (perché tocca la lava), impostiamo una piccola animazione che lo rimpicciolisce o lo fa affondare, riducendo l'altezza dell'oggetto `player`.

Ed ecco il metodo che gestisce la collisione tra il giocatore e gli altri oggetti:

---

```
Level.prototype.playerTouched = function(type, actor) {
    if (type == "lava" && this.status == null) {
        this.status = "lost";
        this.finishDelay = 1;
    } else if (type == "coin") {
        this.actors = this.actors.filter(function(other) {
            return other != actor;
        });
        if (!this.actors.some(function(actor) {
            return actor.type == "coin";
        })) {
            this.status = "won";
            this.finishDelay = 1;
        }
    }
};
```

---

Quando il giocatore tocca la lava, lo stato del gioco viene impostato su "lost" (perso). Quando tocca una moneta, quella moneta viene eliminata dall'array di attori e, se si tratta dell'ultima, lo stato del gioco viene impostato su "won" (vinto).

Con questo, arriviamo a un livello che può essere animato: manca solo il codice dell'animazione vera e propria.

## Seguire i tasti

Per un gioco come questo, non vogliamo che i tasti facciano scattare un evento ogni volta che vengono premuti: vogliamo piuttosto che il loro effetto (spostare la figura del giocatore) sia continuo e duri finché rimangono premuti.

Dobbiamo impostare un gestore che memorizzi lo stato corrente dei quattro tasti freccia. Dobbiamo inoltre richiamare `preventDefault` per impedire a quei tasti di far scorrere la pagina.

La seguente funzione, che accetta un oggetto con i codici dei tasti come nomi di proprietà, e nomi dei tasti come valori, restituisce un oggetto che segue la posizione corrente dei tasti corrispondenti. Registra gestori di eventi "keydown" e "keyup" e, quando il codice del tasto nell'evento è presente nella serie di codici da seguire, aggiorna l'oggetto.

---

```
var arrowCodes = {37: "left", 38: "up", 39: "right"};
function trackKeys(codes) {
  var pressed = Object.create(null);
  function handler(event) {
    if (codes.hasOwnProperty(event.keyCode)) {
      var down = event.type == "keydown";
      pressed[codes[event.keyCode]] = down;
      event.preventDefault();
    }
  }
  addEventListener("keydown", handler);
  addEventListener("keyup", handler);
  return pressed;
}
```

---

Notate che la stessa funzione di gestione viene usata per tutti e due i tipi di evento. Esamina la proprietà `type` dell'oggetto evento per stabilire se aggiornare lo stato del tasto su `true` ("keydown") o su `false` ("keyup").

## Eseguire il gioco

La funzione `requestAnimationFrame`, che abbiamo visto nel [Capitolo 13](#), offre un buon modo per animare un gioco. La sua interfaccia, però, è piuttosto primitiva, perché ci costringe a tener nota dell'ora di quando è stata richiamata la nostra funzione l'ultima volta e a richiamare `requestAnimationFrame` dopo ogni fotogramma.

Definiamo una funzione ausiliaria che avvolga quelle parti noiose in una comoda

interfaccia e ci consenta semplicemente di richiamare `runAnimation` passandole una funzione che accetti come argomento un intervallo di tempo e tracci un solo fotogramma. Quando la funzione `frame` restituisce `false`, l'animazione si ferma.

---

```
function runAnimation(frameFunc) {  
    var lastTime = null;  
    function frame(time) {  
        var stop = false;  
        if (lastTime != null) {  
            var timeStep = Math.min(time - lastTime, 100) / 1000;  
            stop = frameFunc(timeStep) === false;  
        }  
        lastTime = time;  
        if (!stop)  
            requestAnimationFrame(frame);  
    }  
    requestAnimationFrame(frame);  
}
```

Ho impostato un passo massimo di 100 millisecondi (un decimo di secondo) per fotogramma. Quando la scheda o la finestra del browser con la nostra pagina è nascosta, le chiamate a `requestAnimationFrame` vengono sospese fino a quando la scheda o la finestra tornano attive. In questo caso, la differenza tra `lastTime` e `time` corrisponde alla durata del periodo durante il quale la pagina era nascosta. Far avanzare il gioco di tutto quel tempo in una volta avrà uno strano effetto e potrebbe rappresentare un sacco di lavoro (ricordate la suddivisione dei tempi nel metodo `animate`).

La funzione inoltre converte gli intervalli di tempo in secondi, che sono più semplici da capire dei millisecondi.

La funzione `runLevel` accetta un oggetto `Level`, un costruttore per un display ed eventualmente una funzione. Visualizza il livello (in `document.body`) e fa giocare l'utente. Quando il livello è completato (perso o vinto), `runLevel` ripulisce lo schermo, interrompe l'animazione e, se gli era stata data una funzione `andThen`, la richiama con lo stato del livello.

---

```
var arrows = trackKeys(arrowCodes);  
function runLevel(level, Display, andThen) {  
    var display = new Display(document.body, level);  
    runAnimation(function(step) {  
        level.animate(step, arrows);  
        display.drawFrame(step);  
    })  
    .catch(function(error) {  
        console.error(error);  
    })  
    .then(function() {  
        if (andThen)  
            andThen();  
        else  
            display.clear();  
    })  
}
```

```
        if (level.isFinished()) {
            display.clear();
            if (andThen)
                andThen(level.status);
            return false;
        }
    });
}
```

---

Un gioco è una sequenza di livelli. Quando il giocatore muore, viene ripristinato il livello corrente. Quando un livello viene completato, ci spostiamo al livello successivo. Questo concetto può essere espresso dalla funzione seguente, che accetta un array di piani di livelli (array di stringhe) e un costruttore per `Display`:

```
function runGame(plans, Display) {
    function startLevel(n) {
        runLevel(new Level(plans[n]), Display, function(status) {
            if (status == "lost")
                startLevel(n);
            else if (n < plans.length - 1)
                startLevel(n + 1);
            else
                console.log("You win!");
        });
    }
    startLevel(0);
}
```

---

Queste funzioni mostrano uno stile di programmazione caratteristico. Sia `runAnimation`, sia `runLevel` sono funzioni di ordine superiore, ma di uno stile diverso da quello che abbiamo visto nel [Capitolo 5](#). L'argomento della funzione viene usato per organizzare cose che devono avvenire in un momento futuro e nessuna delle funzioni restituisce valori significativi. In un certo senso, il loro compito è solo di mettere in programma delle azioni. Avvolgere queste azioni in funzioni, ci dà modo di memorizzarle come valori, per poterle richiamare al momento giusto.

Questo stile di programmazione si chiama anche *programmazione asincrona*. Anche la gestione degli eventi è un esempio di questo stile e ne vedremo molti altri quando avremo a che fare con operazioni che avvengono in tempi non prevedibili, come per esempio le richieste di rete nel [Capitolo 17](#) e, in generale, input e output nel [Capitolo 20](#).

Il sito del libro offre una serie di piani di livelli nella variabile `GAME_LEVELS`, che

potete scaricare da <http://eloquentjavascript.net/code#15>. La pagina seguente le passa a runGame e avvia il gioco:

---

```
<link rel="stylesheet" href="css/game.css">
<script>
  runGame(GAME_LEVELS, DOMDisplay);
</script>
```

---

## Esercizi

### *Game Over*

Per tradizione, nei giochi a piattaforme il giocatore parte con un numero limitato di *vite*, che diminuisce di uno ogni volta che muore. Quando ha finito le vite a disposizione, il giocatore riparte dall'inizio.

Modificate runGame per aggiungere al gioco questa funzionalità. Il giocatore deve partire con tre vite.

### *Interrompere il gioco*

Fate in modo che premendo il tasto ESC il gioco venga sospeso e ripreso.

Potete farlo modificando la funzione runLevel in modo che usi un altro gestore di eventi di tastiera, che interrompe o riprende l'animazione ogni volta che si usa il tasto ESC.

L'interfaccia di runAnimation non sembra un buon candidato a prima vista, ma andrà bene se reimpostate il modo in cui runLevel la richiama.

Quando le cose funzionano, potete provare qualcos'altro. Il modo con cui abbiamo registrato finora i gestori degli eventi da tastiera presenta dei problemi. L'oggetto arrows è una variabile globale e i gestori dei suoi eventi rimangono in circolazione anche quando non ci sono giochi in corso; in altre parole, avete una *falla* nel sistema. Estendete la funzionalità di track-Keys in modo che si possano deregistrare i suoi gestori e modificate poi runLevel in modo che registri i suoi gestori quando si avvia e li deregistri quando termina.

# 16

## DISEGNARE SU UN ELEMENTO CANVAS

*Il disegno è un inganno.*

M.C. Escher, citato da Bruno Ernst  
in *Lo specchio magico di M.C. Escher*

I browser ci offrono diversi modi per visualizzare elementi grafici. Il modo più semplice è di usare gli stili per posizionare e colorare i normali elementi del DOM. Come abbiamo visto nel gioco del capitolo precedente, questo vi porta già abbastanza lontano. Aggiungendo ai nodi delle immagini di sfondo di una certa trasparenza, possiamo dar loro l'aspetto che vogliamo. È anche possibile far ruotare o allungare i nodi usando l'attributo transform degli stili CSS.

In questo modo, però, useremmo il DOM per cose per le quali non è stato progettato. Alcune operazioni, come per esempio il tracciare una linea tra due punti, sono estremamente complesse da realizzare con gli elementi standard dell'HTML.

Esistono due alternative. La prima è basata sul DOM, ma utilizza il formato SVG (*Scalable Vector Graphics*, immagini vettoriali scalabili) invece degli elementi HTML. Potete pensare al formato SVG come a un dialetto per descrivere documenti dal punto di vista delle forme, più che del testo. Potete richiamare un documento SVG in un documento HTML (con un tag `<embed>`), oppure inserirlo con un tag `<img>`.

La seconda alternativa è l'elemento *canvas* (tela), un elemento del DOM che incapsula un'immagine. Offre un'interfaccia di programmazione per tracciare forme nello spazio occupato dal nodo. La differenza principale tra un elemento canvas e un'immagine SVG è che quest'ultima mantiene la descrizione originaria delle forme, che possono essere spostate o rasterizzate in qualunque momento. L'elemento Canvas, invece, converte le figure in pixel (punti colorati su un raster) non appena si formano e non si ricorda a che cosa corrispondono i pixel. L'unico modo per spostare una forma su una tela è di svuotarla (o di svuotarne la parte intorno alla figura) e ridisegnarlo con la forma nella nuova posizione.

## SVG

In questo libro non entro nei dettagli del formato SVG, ma accenno a come funziona. Alla fine del capitolo, tornerò sugli elementi che dovrete sopesare nel decidere quale

meccanismo di disegno sia più adatto per le vostre applicazioni.

Questo è un documento HTML che contiene una semplice immagine SVG:

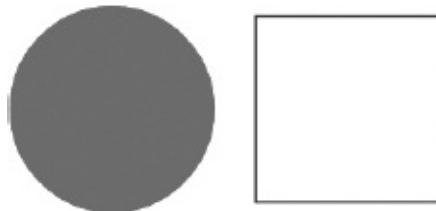
---

```
<p>Normal HTML here.</p>
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="50" cx="50" cy="50" fill="red"/>
  <rect x="120" y="5" width="90" height="90"
        stroke="blue" fill="none"/>
</svg>
```

---

L'attributo `xmlns` modifica un elemento (e i suoi figli) in un diverso *spazio dei nomi XML*. Questo spazio dei nomi, identificato da un URL, specifica il dialetto che stiamo parlando. I tag `<circle>` e `<rect>`, che non esistono in HTML, hanno però un significato in SVG: tracciano le forme corrispondenti (cerchio, rettangolo) con lo stile e la posizione specificati dai loro attributi. Il documento viene reso così:

Normal HTML here.



Questi tag creano degli elementi del DOM proprio come i tag HTML. Per esempio, nel codice seguente l'elemento `<circle>` cambia colore e diventa ciano:

---

```
var circle = document.querySelector("circle");
circle.setAttribute("fill", "cyan");
```

---

## L'elemento canvas

Le immagini canvas si possono disegnare su un elemento `<canvas>`, al quale potete assegnare attributi per larghezza e altezza per fissarne le dimensioni in pixel.

L'elemento di per sé è vuoto: essendo completamente trasparente, nel documento viene reso come spazio vuoto.

Il tag `<canvas>` offre supporto a diversi stili di disegno. Per avere accesso a un'interfaccia di disegno vera e propria, dobbiamo innanzitutto creare un *contesto*, che è un oggetto i cui metodi offrono l'interfaccia di disegno. Esistono attualmente due stili di disegno ampiamente supportati: "2d" per le immagini a due dimensioni e "webgl" per le immagini tridimensionali attraverso l'interfaccia OpenGL.

In questo libro non parliamo di WebGL e ci limitiamo a immagini bidimensionali. Se vi interessano le immagini tridimensionali, vi invito a prendere in considerazione WebGL,

che offre un'interfaccia molto immediata per l'hardware grafico moderno e vi permette pertanto di realizzare con efficienza anche scene molto complicate attraverso JavaScript.

---

Un contesto si crea richiamando il metodo `getContext` sull'elemento `<canvas>`.

```
<p>Before canvas.</p>
<canvas width="120" height="60"></canvas>
<p>After canvas.</p>
<script>
    var canvas = document.querySelector("canvas");
    var context = canvas.getContext("2d");
    context.fillStyle = "red";
    context.fillRect(10, 10, 100, 50);
</script>
```

---

Nell'esempio, dopo aver creato l'oggetto contesto, disegniamo un rettangolo rosso largo 100 pixel e alto 50 pixel, con l'angolo superiore sinistro nelle coordinate (10,10).

Before canvas.



After canvas.

Proprio come in HTML (e in SVG), il sistema di coordinate imposta su (0,0) l'angolo superiore sinistro, con l'asse y positiva che si sposta da quel punto verso il basso. Pertanto, (10,10) si trova 10 pixel sotto e a destra dell'angolo superiore sinistro.

## Riempimento e spessore del bordo

Nell'area di disegno, le forme possono essere *riempite*, assegnando alla loro superficie un determinato colore o motivo, e possono essere *contornate* con una linea che ne traccia il bordo. La stessa terminologia si usa per il formato SVG.

Il metodo `fillRect` riempie un rettangolo. Accetta prima le coordinate x e y dell'angolo superiore sinistro del rettangolo, poi la sua larghezza e infine l'altezza. Un metodo simile, `strokeRect`, traccia il contorno del rettangolo.

Nessuno dei due metodi accetta altri parametri. Il colore di riempimento, lo spessore del bordo e così via non sono determinati da argomenti del metodo (come potreste giustamente pensare), ma piuttosto dalle proprietà dell'oggetto contesto.

Impostare `fillStyle` modifica il riempimento delle forme. Può essere impostato su

una stringa che specifica il colore, in una qualunque delle notazioni ammesse dalla specifica CSS.

La proprietà `strokeStyle` funziona in maniera analoga, ma determina il colore usato per il bordo. Lo spessore di quella linea è determinato dalla proprietà `lineWidth`, che può contenere qualunque numero positivo.

---

```
<canvas></canvas>
<script>
var cx = document.querySelector("canvas").getContext("2d");
cx.strokeStyle = "blue";
cx.strokeRect(5, 5, 50, 50);
cx.lineWidth = 5;
cx.strokeRect(135, 5, 50, 50);
</script>
```

---

Il codice traccia due quadrati azzurri, il secondo dei quali ha un bordo più spesso.



Quando non sono specificati gli attributi `width` (larghezza) o `height` (altezza), come nell'esempio precedente, l'elemento riceve una larghezza predefinita di 300 pixel e un'altezza predefinita di 150 pixel.

## Il tracciato

Un tracciato (`path`) è una sequenza di linee. L'interfaccia 2D dell'area di disegno descrive un tracciato in maniera piuttosto originale, servendosi esclusivamente di effetti secondari. I tracciati non sono valori che possono essere memorizzati e passati. Se volete intervenire su un tracciato, dovrete definire una sequenza di chiamate a metodi per descriverne la forma.

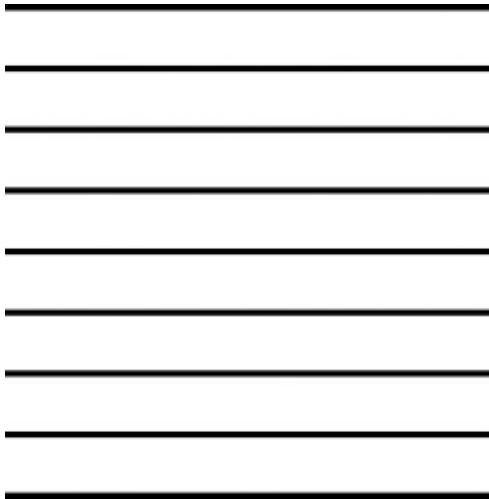
---

```
<canvas></canvas>
<script>
var cx = document.querySelector("canvas").getContext("2d");
cx.beginPath();
for (var y = 10; y < 100; y += 10) {
  cx.moveTo(10, y);
```

```
    cx.lineTo(90, y);
}
cx.stroke();
</script>
```

---

Il tracciato descritto dal programma ha l'aspetto seguente:



L'esempio crea un tracciato con una serie di segmenti di linee orizzontali e ne segna poi il bordo col metodo `stroke`. Ciascuno dei segmenti creati con `lineTo` inizia dalla posizione corrente del tracciato. Quella posizione è di solito la fine dell'ultimo segmento, tranne quando si richiama `moveTo`. In quel caso, il segmento successivo inizia dalla posizione passata a `moveTo`.

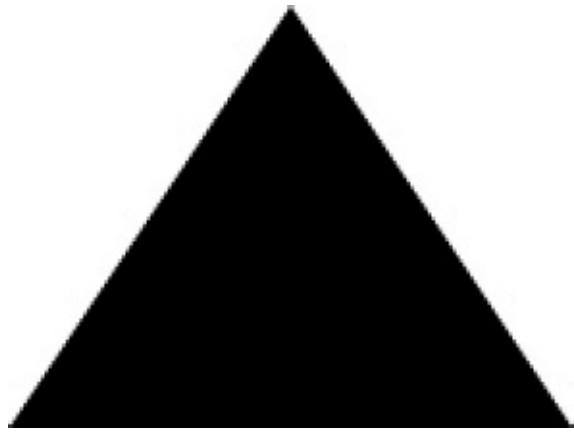
Quando si riempie il tracciato (usando il metodo `fill`), ogni forma viene riempita separatamente. Un tracciato può contenere più forme, una per ogni invocazione di `moveTo`. Il tracciato, però, deve essere *chiuso* (in modo che le posizioni di inizio e di fine coincidano), prima di poter essere riempito. Se il tracciato non è già chiuso, viene aggiunta una linea dalla fine all'inizio e viene riempita la forma racchiusa dal tracciato così completato.

---

```
<canvas></canvas>
<script>
var cx = document.querySelector("canvas").getContext("2d");
cx.beginPath();
cx.moveTo(50, 10);
cx.lineTo(10, 70);
cx.lineTo(90, 70);
cx.fill();
</script>
```

---

Questo esempio traccia un triangolo pieno. Notate che solo due dei lati del triangolo vengono tracciati esplicitamente. Il terzo, dall'angolo inferiore destro al vertice superiore, è implicito e non compare quando disegnate il bordo del tracciato.



Potreste usare anche il metodo `closePath` per chiudere esplicitamente un tracciato, aggiungendo un segmento al suo inizio. Questo segmento viene disegnato col bordo del tracciato.

## Curve

Un tracciato può anche contenere linee curve, che sono purtroppo un po' più complesse da disegnare delle rette.

Il metodo `quadraticCurveTo` traccia una curva in un punto dato. Per stabilire la curvatura della linea, al metodo si passano un punto di destinazione e uno di controllo. Pensate al punto di controllo come a un magnete che attira la linea, dandole la sua curvatura. La linea non attraversa il punto di controllo: invece, la sua direzione nei punti di partenza e di arrivo è tale da allinearsi con le linee che li collegano al punto di controllo.

Il seguente esempio illustra questo concetto:

---

```
<canvas></canvas>
<script>
var cx = document.querySelector("canvas").getContext("2d");
cx.beginPath();
cx.moveTo(10, 90);
// control=(60,10) goal=(90,90)
cx.quadraticCurveTo(60, 10, 90, 90);
cx.lineTo(60, 10);
cx.closePath();
cx.stroke();
</script>
```

---

Il programma produce il tracciato seguente:



Tracciamo una curva quadratica da sinistra a destra, con (60,10) come punto di controllo, e quindi tracciamo due segmenti di retta che attraversano il punto di controllo e tornano all'inizio della linea. Il risultato ricorda vagamente una mostrina di *Star Trek*. Notate l'effetto del punto di controllo: le linee che partono dagli angoli inferiori vanno nella sua direzione e convergono verso la loro destinazione.

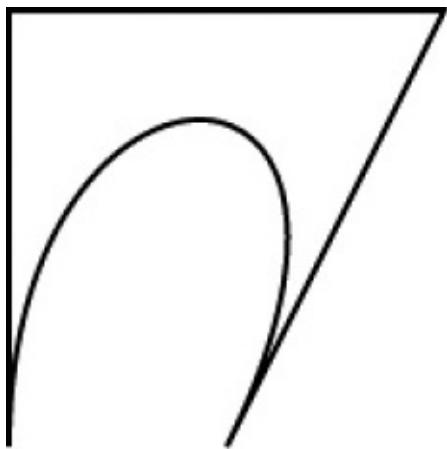
Il metodo `bezierCurveTo` traccia un tipo simile di curva. Invece di un solo punto di controllo, le curve di Bézier ne hanno due, uno per ciascuno dei punti terminali della linea. Ecco un esempio del comportamento di questo tipo di curva:

---

```
<canvas></canvas>
<script>
var cx = document.querySelector("canvas").getContext("2d");
cx.beginPath();
cx.moveTo(10, 90);
// control1=(10,10) control2=(90,10) goal=(50,90)
cx.bezierCurveTo(10, 10, 90, 10, 50, 90);
cx.lineTo(90, 10);
cx.lineTo(10, 10);
cx.closePath();
cx.stroke();
</script>
```

---

I due punti di controllo specificano la direzione alle due estremità della curva. Più distano dai punti (di partenza) corrispondenti, più la curva si piegherà verso quella direzione.



Queste curve non sono facili da disegnare: non è sempre chiaro come trovare i punti di controllo che diano la forma che vi interessa. A volte li potete calcolare e a volte dovrete accontentarvi di procedere per tentativi per trovare un valore soddisfacente.

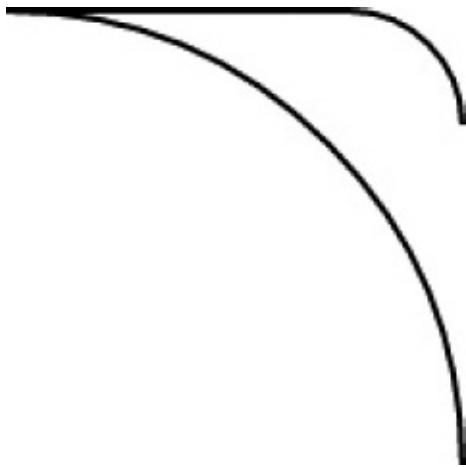
Gli *archi*, frammenti di un cerchio, sono più immediati. Il metodo `arcTo` accetta almeno cinque argomenti. I primi quattro sono simili a quelli di `quadraticCurveTo`. La prima coppia fornisce una specie di punto di controllo e la seconda dà la destinazione della linea. Il quinto argomento dà il raggio dell'arco. Concettualmente, questo metodo proietta un angolo, una linea che va al punto di controllo e poi al punto di destinazione, e ne arrotonda il vertice in modo che formi parte di un cerchio col raggio dato. Il metodo `arcTo` traccia quindi la parte arrotondata e una linea dalla posizione di partenza all'inizio della parte arrotondata.

---

```
<canvas></canvas>
<script>
var cx = document.querySelector("canvas").getContext("2d");
cx.beginPath();
cx.moveTo(10, 10);
// control=(90,10) goal=(90,90) radius=20
cx.arcTo(90, 10, 90, 90, 20);
cx.moveTo(10, 10);
// control=(90,10) goal=(90,90) radius=80
cx.arcTo(90, 10, 90, 90, 80);
cx.stroke();
</script>
```

---

L'esempio produce due angoli arrotondati con raggi diversi.



A dispetto del nome (`arcTo`, arco fino a), il metodo `arcTo` non traccia la linea dalla fine della parte arrotondata alla posizione di destinazione. Se volete aggiungere quella parte di linea, potete farlo seguendo da una chiamata a `lineTo` con le coordinate del punto di destinazione.

Per disegnare un cerchio, potete usare quattro chiamate ad `arcTo`, ciascuna con un angolo di 90 gradi. Il metodo `arc`, però, offre un'alternativa più semplice. Accetta una coppia di coordinate per il centro dell'arco, un raggio, un angolo di partenza e uno di arrivo.

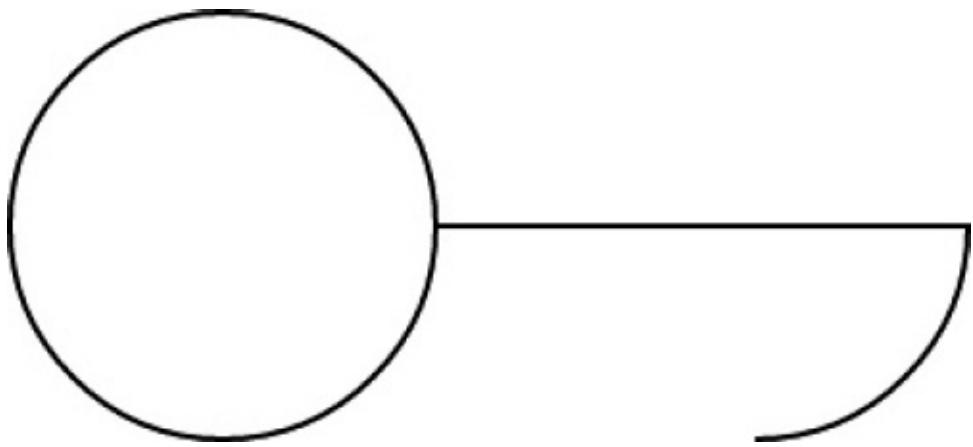
Questi ultimi due parametri permettono di disegnare solo una parte di cerchio, in quanto gli angoli si misurano in radianti e non in gradi. Ciò significa che il cerchio completo ha un angolo di  $2\pi$ , o  $2 * \text{Math.PI}$ , che è circa 6,28. Il conteggio inizia nel punto a destra del centro del cerchio e avanza in senso orario. Per disegnare un cerchio completo, potete partire dal punto 0 e continuare fino a un punto più grande di  $2\pi$  (per esempio, 7).

---

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  // center=(50,50) radius=40 angle=0 to 7
  cx.arc(50, 50, 40, 0, 7);
  // center=(150,50) radius=40 angle=0 to ½π
  cx.arc(150, 50, 40, 0, 0.5 * Math.PI);
  cx.stroke();
</script>
```

---

L'immagine risultante contiene una linea dalla destra del cerchio completo (la prima chiamata ad `arc`) alla destra del quarto di cerchio (seconda chiamata). Come con gli altri metodi per disegnare, la linea tracciata con `arc` è connessa al segmento di tracciato precedente per default. Per evitarlo, dovete richiamare `moveTo` o iniziare un nuovo tracciato.



## Disegnare un grafico a torta

Immaginate di essere appena stati assunti da una grande azienda, dove come primo lavoro vi chiedono di produrre un grafico a torta che illustri i risultati di un'indagine sul grado di soddisfazione della loro clientela.

La variabile `results` contiene un array di oggetti che rappresentano le risposte dei clienti.

---

```
var results = [
  {name: "Satisfied", count: 1043, color: "lightblue"},
  {name: "Neutral", count: 563, color: "lightgreen"},
  {name: "Unsatisfied", count: 510, color: "pink"},
  {name: "No comment", count: 175, color: "silver"}
];
```

---

Per disegnare un diagramma circolare, tracciamo una serie di fette di torta, ciascuna composta da un arco e da due linee che partono dal centro dell'arco. Per calcolare l'ampiezza dell'angolo di ciascuna fetta, dividiamo la circonferenza ( $2\pi$ ) per il totale delle risposte e moltiplichiamo il risultato per il numero di persone che hanno scelto una data risposta.

---

```
<canvas width="200" height="200"></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  var total = results.reduce(function(sum, choice) {
    return sum + choice.count;
  }, 0);
  // Start at the top
  var currentAngle = -0.5 * Math.PI;
  results.forEach(function(result) {
```

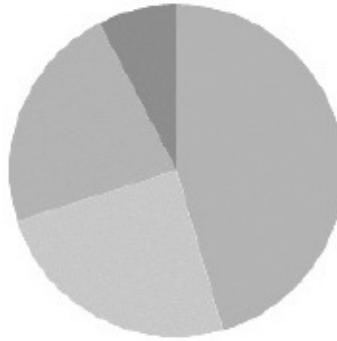
```

var sliceAngle = (result.count / total) * 2 * Math.PI;
cx.beginPath();
// center=100,100, radius=100
// from current angle, clockwise by slice's angle
cx.arc(100, 100, 100,
       currentAngle, currentAngle + sliceAngle);
currentAngle += sliceAngle;
cx.lineTo(100, 100);
cx.fillStyle = result.color;
cx.fill();
});
</script>

```

---

Ecco il diagramma corrispondente:



Un diagramma che non spiega i risultati, però, non è molto utile. Dobbiamo pertanto trovare il modo di aggiungere del testo all'area di disegno.

## Il testo

Il contesto 2D per l'area di disegno offre i metodi `fillText` e `strokeText`. Quest'ultimo si usa per aggiungere un bordo alle lettere; di solito basta usare `fillText`, che riempie il testo dato col colore di riempimento corrente, `fillColor`.

---

```

<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  cx.font = "28px Georgia";
  cx.fillStyle = "fuchsia";
  cx.fillText("I can draw text, too!", 10, 50);
</script>

```

Potete specificare le dimensioni, lo stile e il tipo di carattere attraverso la proprietà

`font`. Nell'esempio ci limitiamo a dare dimensioni e nome di un tipo di carattere. All'inizio della stringa data per `cx.font` potete specificare anche `italic` per il corsivo o `bold` per il grassetto.

Gli ultimi due argomenti di `fillText` (e `strokeText`) indicano la posizione dove si traccia il testo. Nell'impostazione predefinita, questa è la posizione della base del carattere, ossia la riga sulla quale poggiano le lettere senza contare le parti discendenti di lettere come la *g* e la *p*. Potete modificare la posizione orizzontale impostando la proprietà `textAlign` su "end" o "center" e quella verticale impostando `textBaseline` su "top", "middle" o "bottom".

Torneremo al nostro grafico a torta e alla questione di come etichettare le fette nell'esercizio al termine del capitolo.

## Immagini

Nella grafica digitale, si fa distinzione tra immagini *vettoriali* e immagini *bitmap*. Le prime sono quel che abbiamo fatto finora in questo capitolo: immagini specificate da una descrizione logica di forme. Le immagini bitmap, invece, non specificano forme ma lavorano con dati pixel o raster di punti colorati.

Il metodo `drawImage` ci permette di disegnare dei pixel su una tela. I dati possono aver origine da un elemento `<img>` o da altre aree di disegno, e nessuno deve per forza essere visibile nel documento. Nell'esempio che segue, creiamo un elemento separato `<img>`, nel quale carichiamo un'immagine: ma il codice non può cominciare subito a disegnare l'immagine, perché non è detto che il browser l'abbia già caricata. Per risolvere questo problema, registriamo un gestore di eventi "load" e rimandiamo il disegno a dopo aver caricato l'immagine.

```
<canvas></canvas>
```

---

```
<script>
var cx = document.querySelector("canvas").getContext("2d");
var img = document.createElement("img");
img.src = "img/hat.png";
img.addEventListener("load", function() {
  for (var x = 10; x < 200; x += 30)
    cx.drawImage(img, x, 10);
});
</script>
```

Per default, `drawImage` disegna l'immagine nelle sue dimensioni originarie, ma potete aggiungere due argomenti per impostare altre misure per altezza e larghezza.

Quando gli si passano nove argomenti, `drawImage` permette di disegnare solo una parte

di un’immagine. Gli argomenti dal secondo al quinto indicano il rettangolo (x, y, larghezza e altezza) che va copiato dall’immagine sorgente, mentre gli argomenti dal sesto al nono indicano il rettangolo nell’area di disegno dove incollare l’immagine.

In questo modo, potete raggruppare più *sprite* (elementi di immagini) in un solo file e quindi disegnare solo la parte che vi interessa. Prendiamo la nostra immagine di esempio, che contiene un personaggio in diverse posizioni:



Alternando le posizioni, possiamo rappresentare un’animazione dove il personaggio sembra camminare.

Per animare l’immagine, si usa il metodo `clearRect`. È simile a `fillRect`, ma invece di colorare il rettangolo, lo rende trasparente eliminando i pixel disegnati in precedenza.

Sappiamo che ogni *sprite*, ogni fotogramma dell’immagine, è largo 24 pixel e alto 30. Il codice che segue carica l’immagine e imposta un intervallo (un timer ripetuto) per disegnare il *frame* seguente:

---

```
<canvas></canvas>
<script>
var cx = document.querySelector("canvas").getContext("2d");
var img = document.createElement("img");
img.src = "img/player.png";
var spriteW = 24, spriteH = 30;
img.addEventListener("load", function() {
    var cycle = 0;
    setInterval(function() {
        cx.clearRect(0, 0, spriteW, spriteH);
        cx.drawImage(img,
                    // source rectangle
                    cycle * spriteW, 0, spriteW, spriteH,
                    // destination rectangle
                    0,      0, spriteW, spriteH);
        cycle = (cycle + 1) % 8;
    }, 120);
});</script>
```

---

La variabile `cycle` tiene nota dell’avanzamento dell’animazione: aumenta per ogni

fotogramma e ricomincia da 0 nell'intervallo da 0 a 7 usando l'operatore resto. La si usa poi per calcolare la coordinata x per lo sprite della posa corrente.

## Trasformazioni

Se vogliamo che il personaggio cammini verso sinistra invece che verso destra, una soluzione potrebbe essere aggiungere altri sprite. Possiamo anche invertire la direzione dell'immagine programmaticamente nella tela

Richiamando il metodo `scale`, si ridimensiona tutto quel che si disegna dopo di esso. Questo metodo accetta due parametri, uno per il rapporto orizzontale e uno per quello verticale.

---

```
<canvas></canvas>
<script>
var cx = document.querySelector("canvas").getContext("2d");
cx.scale(3, .5);
cx.beginPath();
cx.arc(50, 50, 40, 0, 7);
cx.lineWidth = 3;
cx.stroke();
</script>
```

---

Con la chiamata a `scale`, la larghezza del cerchio triplica e la sua altezza si dimezza.



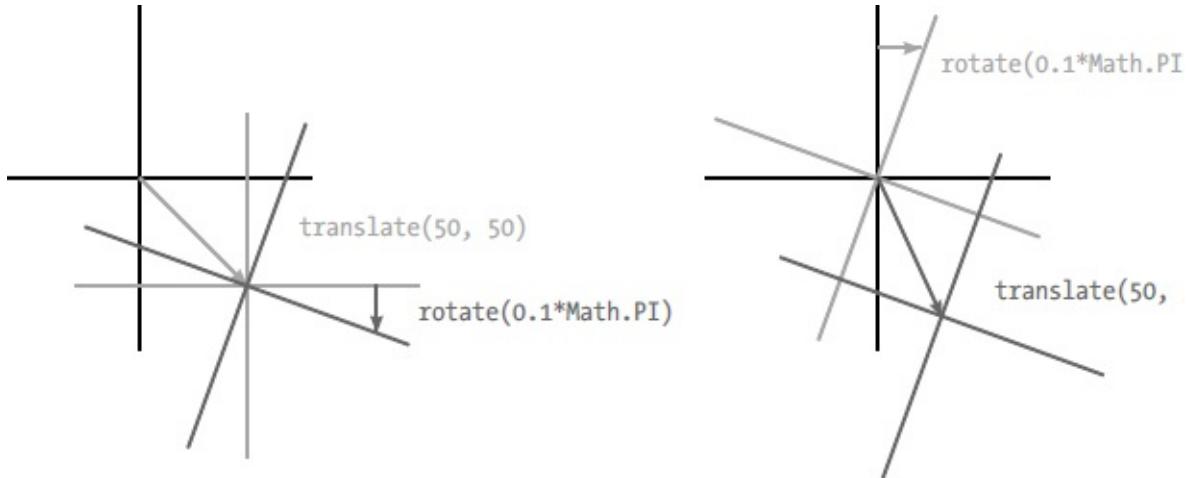
Ridimensionando l'immagine, tutte le sue parti, compreso lo spessore della linea, vengono allargate o compresse secondo quanto specificato. Se come rapporto di scala si sceglie un valore negativo, allora l'immagine si ribalta facendo perno sul punto (0,0). Ciò significa che si inverte anche la direzione del sistema di coordinate. Quando si applica un rapporto di scala di -1, la figura tracciata nella posizione 100 sull'asse x finisce a quella che era la posizione -100.

Pertanto, per far cambiare orientamento a un'immagine non possiamo semplicemente aggiungere `cx.scale(-1, 1)` prima della chiamata a `drawImage`, perché così facendo la figura andrebbe a finire fuori dall'area di disegno e non sarebbe più visibile. Potete invece rettificare le coordinate passate a `drawImage` e compensare tracciando l'immagine nella posizione x -50 invece di 0. Un'altra soluzione, dove il codice che calcola il disegno può ignorare il ridimensionamento, è di regolare l'asse intorno al quale esso avviene.

Ci sono diversi altri metodi, oltre a `scale`, che influenzano il sistema di coordinate per le aree di disegno. Potete ruotare delle forme tracciate in successione col metodo `rotate` e spostarle col metodo `translate`. Un aspetto interessante, e un po' disorientante, è che

tutte le trasformazioni si sommano, nel senso che ciascuna avviene in relazione alla trasformazione precedente.

Pertanto, se applichiamo due volte il metodo `translate` per 10 pixel orizzontali, tutto sarà disegnato a destra di 20 pixel. Se prima spostiamo il centro del sistema di coordinate su (50,50) e poi applichiamo `rotate` per ruotare l'immagine di 20 gradi ( $0,1\pi$  in radianti), quella rotazione avverrà attorno al punto (50,50).



Se prima ruotiamo di 20 gradi e poi applichiamo `translate` con (50,50), la trasposizione avverrà, invece, nel sistema di coordinate già ruotato e avrà un diverso orientamento. L'ordine delle trasformazioni conta!

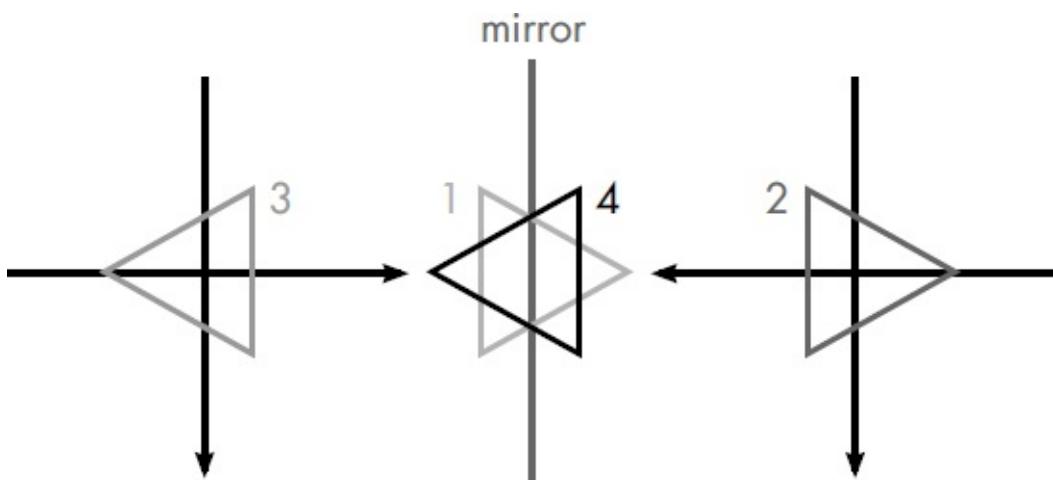
Per riflettere un'immagine lungo la linea verticale in una posizione x data, possiamo scrivere:

---

```
function flipHorizontally(context, around) {
    context.translate(around, 0);
    context.scale(-1, 1);
    context.translate(-around, 0);
}
```

---

Spostiamo l'asse y nel punto dove vogliamo che stia l'immagine specchiata, applichiamo il riflesso e riportiamo l'asse y al suo posto nell'universo specchiato. L'immagine seguente spiega perché ciò avviene:



L'immagine mostra i sistemi di coordinate prima e dopo la riflessione sulla linea centrale. Se disegniamo un triangolo in una posizione x positiva, la sua posizione predefinita è quella dove si trova il triangolo 1. Una chiamata a `flipHorizontally` prima effettua una trasposizione a destra, che ci porta al triangolo 2; poi calcola il ridimensionamento e riflette il triangolo in posizione 3. Quella, però, non è la posizione giusta se fosse riflesso lungo la linea data. La seconda chiamata a `translate` mette a posto le cose: "annulla" la prima trasposizione e mostra il triangolo 4 esattamente dove deve stare.

Possiamo ora tracciare un carattere specchiato nella posizione (100,0) facendo ruotare il mondo intorno al centro verticale del carattere.

---

```
<canvas></canvas>
<script>
  var cx = document.querySelector("canvas").getContext("2d");
  var img = document.createElement("img");
  img.src = "img/player.png";
  var spriteW = 24, spriteH = 30;
  img.addEventListener("load", function() {
    flipHorizontally(cx, 100 + spriteW / 2);
    cx.drawImage(img, 0, 0, spriteW, spriteH,
      100, 0, spriteW, spriteH);
  });
</script>
```

---

## Memorizzare ed eliminare le trasformazioni

Le trasformazioni non vanno via. Tutto quel che disegniamo dopo il carattere specchiato risulterà specchiato e questo potrebbe essere un problema.

È possibile salvare la trasformazione corrente, fare altri disegni e trasformazioni e quindi recuperare la trasformazione salvata. Questa è di solito la cosa giusta da fare per una funzione che deve trasformare solo temporaneamente il sistema di coordinate. Per prima cosa, salviamo la trasformazione usata dal codice che ha richiamato la funzione. Poi, la funzione svolgerà altri calcoli (al di sopra della trasformazione esistente) e magari altre trasformazioni. Alla fine, torniamo alla trasformazione dalla quale eravamo partiti.

I metodi `save` e `restore` del contesto 2D della tela, vi permettono di gestire le trasformazioni in questo modo. Concettualmente, essi mantengono una pila di stati di trasformazione. Quando richiamate `save`, lo stato corrente viene inserito nella pila; quando richiamate `restore`, viene recuperato e usato come trasformazione corrente del contesto lo stato che si trova in cima alla pila.

La funzione `branch` dell'esempio che segue illustra quel che potete fare con una funzione che cambia la trasformazione e quindi richiama un'altra funzione (in questo caso, se stessa), che continua a disegnare con la trasformazione data.

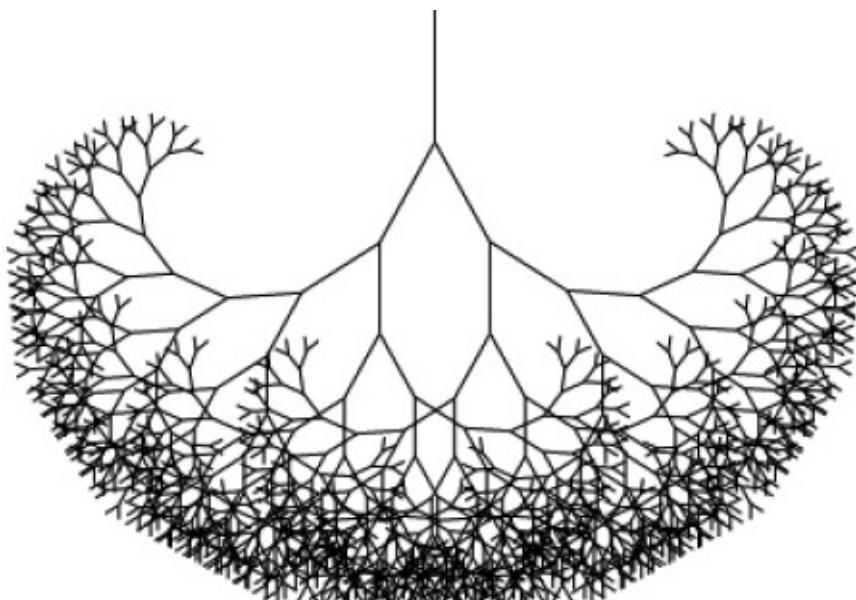
Questa funzione disegna una forma ad albero partendo da una linea, spostando il centro del sistema di coordinate alla fine della linea e richiamando se stessa più volte: la prima volta con una rotazione a destra, la seconda con una a sinistra. Ogni chiamata riduce la lunghezza del ramo e la ricorsione si interrompe quando questa scende al di sotto di 8.

---

```
<canvas width="600" height="300"></canvas>
<script>
var cx = document.querySelector("canvas").getContext("2d");
function branch(length, angle, scale) {
  cx.fillRect(0, 0, 1, length);
  if (length < 8) return;
  cx.save();
  cx.translate(0, length);
  cx.rotate(-angle);
  branch(length * scale, angle, scale);
  cx.rotate(2 * angle);
  branch(length * scale, angle, scale);
  cx.restore();
}
cx.translate(300, 0);
branch(60, 0.5, 0.8);
</script>
```

---

Il risultato è un semplice frattale.



Se non ci fossero le chiamate a `save` e `restore`, la seconda chiamata ricorsiva a `branch` finirebbe con la posizione e la rotazione create dalla prima chiamata. Non ci sarebbe collegamento col ramo corrente ma piuttosto col ramo più interno e sulla destra, tracciato dalla prima chiamata. La forma risultante può essere interessante, ma di certo non è un albero.

## Torniamo al gioco

Sappiamo ora abbastanza di disegni su tela, da poter cominciare a lavorare su questo elemento per il gioco del capitolo precedente. Il nuovo piano di gioco non sarà composto solo da riquadri colorati: useremo invece `drawImage` per disegnare delle immagini che rappresentino gli elementi di gioco.

Definiremo un oggetto `CanvasDisplay`, che accetta la stessa interfaccia di `DOMDisplay` ([Capitolo 15](#)) e in particolare i suoi metodi `drawFrame` e `clear`.

Questo oggetto mantiene qualche dato in più rispetto a `DOMDisplay`. Invece di usare la posizione di scorrimento del suo elemento DOM, segue la propria area di visualizzazione (viewport) e ci dice quale parte del livello stiamo guardando. Tiene inoltre nota dei tempi, per decidere quale fotogramma dell'animazione usare. Infine, mantiene una proprietà `flipPlayer`, così che il giocatore continui a guardare nell'ultima direzione seguita anche quando sta fermo.

---

```
function CanvasDisplay(parent, level) {
    this.canvas = document.createElement("canvas");
    this.canvas.width = Math.min(600, level.width * scale);
    this.canvas.height = Math.min(450, level.height * scale);
    parent.appendChild(this.canvas);
    this.cx = this.canvas.getContext("2d");
    this.level = level;
    this.animationTime = 0;
    this.flipPlayer = false;
    this.viewport = {
        left: 0,
        top: 0,
        width: this.canvas.width / scale,
        height: this.canvas.height / scale
    };
    this.drawFrame(0);
}
CanvasDisplay.prototype.clear = function() {
```

```
this.canvas.parentNode.removeChild(this.canvas);  
};
```

---

Il contatore `animationTime` spiega perché abbiamo passato le dimensioni del passo a `drawFrame` nel [Capitolo 15](#), anche se `DOMDisplay` non ne fa uso. La nuova funzione `drawFrame` usa il contatore per tener nota del tempo e passare da un fotogramma all'altro a seconda del tempo corrente.

```
CanvasDisplay.prototype.drawFrame = function(step) {  
    this.animationTime += step;  
    this.updateViewport();  
    this.clearDisplay();  
    this.drawBackground();  
    this.drawActors();  
};
```

---

Oltre a tener nota del tempo, il metodo aggiorna l'area di visualizzazione per la posizione corrente del giocatore, riempie tutta l'area di disegno con un colore e vi disegna lo sfondo e gli attori. Notate che questa è un'impostazione diversa da quella del [Capitolo 15](#), dove avevamo disegnato lo sfondo una sola volta e spostavamo l'elemento del DOM circoscritto per spostarlo.

Poiché le forme sulla tela virtuale sono solo pixel, dopo averle disegnate non possiamo spostarle, né eliminarle. L'unico modo per aggiornare lo schermo è di ripulirlo e ridisegnare la scena.

Il metodo `updateViewport` è simile al metodo `scrollPlayerIntoView` di `DOMDisplay`: verifica se il giocatore è troppo vicino ai margini dello schermo e in questo caso sposta l'area di visualizzazione.

```
CanvasDisplay.prototype.updateViewport = function() {  
    var view = this.viewport, margin = view.width / 3;  
    var player = this.level.player;  
    var center = player.pos.plus(player.size.times(0.5));  
    if (center.x < view.left + margin)  
        view.left = Math.max(center.x - margin, 0);  
    else if (center.x > view.left + view.width - margin)  
        view.left = Math.min(center.x + margin - view.width,  
                             this.level.width - view.width);  
    if (center.y < view.top + margin)  
        view.top = Math.max(center.y - margin, 0);  
    else if (center.y > view.top + view.height - margin)
```

```
    view.top = Math.min(center.y + margin - view.height,  
                        this.level.height - view.height);  
};
```

---

Le chiamate a `Math.max` e `Math.min` fanno in modo che l'area di visualizzazione non mostri dello spazio esterno al livello. `Math.max(x, 0)` evita che il numero risultante sia inferiore a zero; analogamente, `Math.min` fa in modo che il valore sia compreso nei limiti dati.

Per ripulire lo schermo, useremo un colore leggermente diverso a seconda che il giocatore abbia vinto (con un colore più brillante) o perso (con un colore più scuro).

---

```
CanvasDisplay.prototype.clearDisplay = function() {  
    if (this.level.status == "won")  
        this.cx.fillStyle = "rgb(68, 191, 255)";  
    else if (this.level.status == "lost")  
        this.cx.fillStyle = "rgb(44, 136, 214)";  
    else  
        this.cx.fillStyle = "rgb(52, 166, 251)";  
    this.cx.fillRect(0, 0,  
                    this.canvas.width, this.canvas.height);  
};
```

---

Per disegnare lo sfondo, passiamo in rassegna le caselle che risultano visibili nell'area di visualizzazione, con lo stesso sistema che avevamo usato nel capitolo precedente in `obstacleAt`.

---

```
var otherSprites = document.createElement("img");  
otherSprites.src = "img/sprites.png";  
CanvasDisplay.prototype.drawBackground = function() {  
    var view = this.viewport;  
    var xStart = Math.floor(view.left);  
    var xEnd = Math.ceil(view.left + view.width);  
    var yStart = Math.floor(view.top);  
    var yEnd = Math.ceil(view.top + view.height);  
    for (var y = yStart; y < yEnd; y++) {  
        for (var x = xStart; x < xEnd; x++) {  
            var tile = this.level.grid[y][x];  
            if (tile == null) continue;  
            var screenX = (x - view.left) * scale;  
            var screenY = (y - view.top) * scale;
```

```

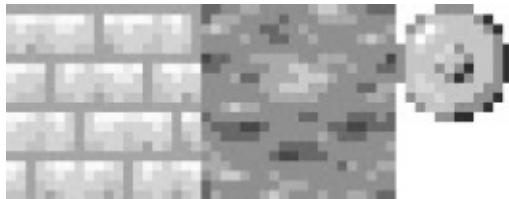
        var tileX = tile == "lava" ? scale : 0;
        this.cx.drawImage(otherSprites,
            tileX,    0, scale, scale,
            screenX, screenY, scale, scale);
    }
}

};


```

---

Disegniamo con `drawImage` le caselle che non sono vuote (null). L'immagine `otherSprites` contiene le figure degli elementi diversi dal giocatore: da sinistra a destra, la tessera per il muro, quella per la lava e lo sprite per la moneta.



Le caselle dello sfondo sono di 20x20 pixel, nella stessa scala che avevamo usato in `DOMDisplay`. Pertanto, lo spostamento per le tessere di lava è 20 (il valore della variabile `scale`) e quello per i muri è 0.

Non aspettiamo che la parte con lo sprite finisca di caricare: richiamare `drawImage` con un'immagine che non è stata caricata non dà alcun risultato. Pertanto, il gioco può risultare in parte incompleto nei primi frame, mentre l'immagine sta caricando. Non è un problema, però: poiché continuiamo ad aggiornare lo schermo, la scena corretta sarà visualizzata appena l'immagine finisce di caricare.

Il personaggio che cammina, che abbiamo visto prima, rappresenta il giocatore. Il codice che lo disegna deve scegliere lo sprite e la direzione giusti a seconda del movimento corrente. I primi otto sprite contengono un'animazione dove il personaggio cammina. Quando il giocatore si sposta lungo la base, li passiamo in ciclo in base alla proprietà `animationTime`. Quest'ultima è misurata in secondi; poiché vogliamo cambiare inquadratura 12 volte al secondo, per prima cosa si moltiplica il tempo per 12. Quando il giocatore sta fermo, disegniamo il nono sprite. Nei salti, che si riconoscono perché la velocità verticale è diversa da zero, usiamo lo sprite più a destra, il decimo.

Poiché gli sprite sono leggermente più larghi dell'oggetto `player` (24 pixel invece di 16, per lasciare un po' di spazio per piedi e braccia), il metodo deve correggere la coordinata x e la larghezza con un valore dato (`playerXOverlap`).

---

```

var playerSprites = document.createElement("img");
playerSprites.src = "img/player.png";
var playerXOverlap = 4;
CanvasDisplay.prototype.drawPlayer = function(x, y, width, height) {
    var sprite = 8, player = this.level.player;

```

```

width += playerXOverlap * 2;
x -= playerXOverlap;
if (player.speed.x != 0)
    this.flipPlayer = player.speed.x < 0;
if (player.speed.y != 0)
    sprite = 9;
else if (player.speed.x != 0)
    sprite = Math.floor(this.animationTime * 12) % 8;
this.cx.save();
if (this.flipPlayer)
    flipHorizontally(this.cx, x + width / 2);
this.cx.drawImage(playerSprites,
                  sprite * width, 0, width, height,
                  x, y, width, height);
this.cx.restore();
};


```

---

Il metodo `drawPlayer` è richiamato da `drawActors`, che ha il compito di disegnare tutti gli attori del gioco.

---

```

CanvasDisplay.prototype.drawActors = function() {
    this.level.actors.forEach(function(actor) {
        var width = actor.size.x * scale;
        var height = actor.size.y * scale;
        var x = (actor.pos.x - this.viewport.left) * scale;
        var y = (actor.pos.y - this.viewport.top) * scale;
        if (actor.type == "player") {
            this.drawPlayer(x, y, width, height);
        } else {
            var tileX = (actor.type == "coin" ? 2 : 1) * scale;
            this.cx.drawImage(otherSprites,
                              tileX, 0, width, height,
                              x, y, width, height);
        }
    }, this);
};


```

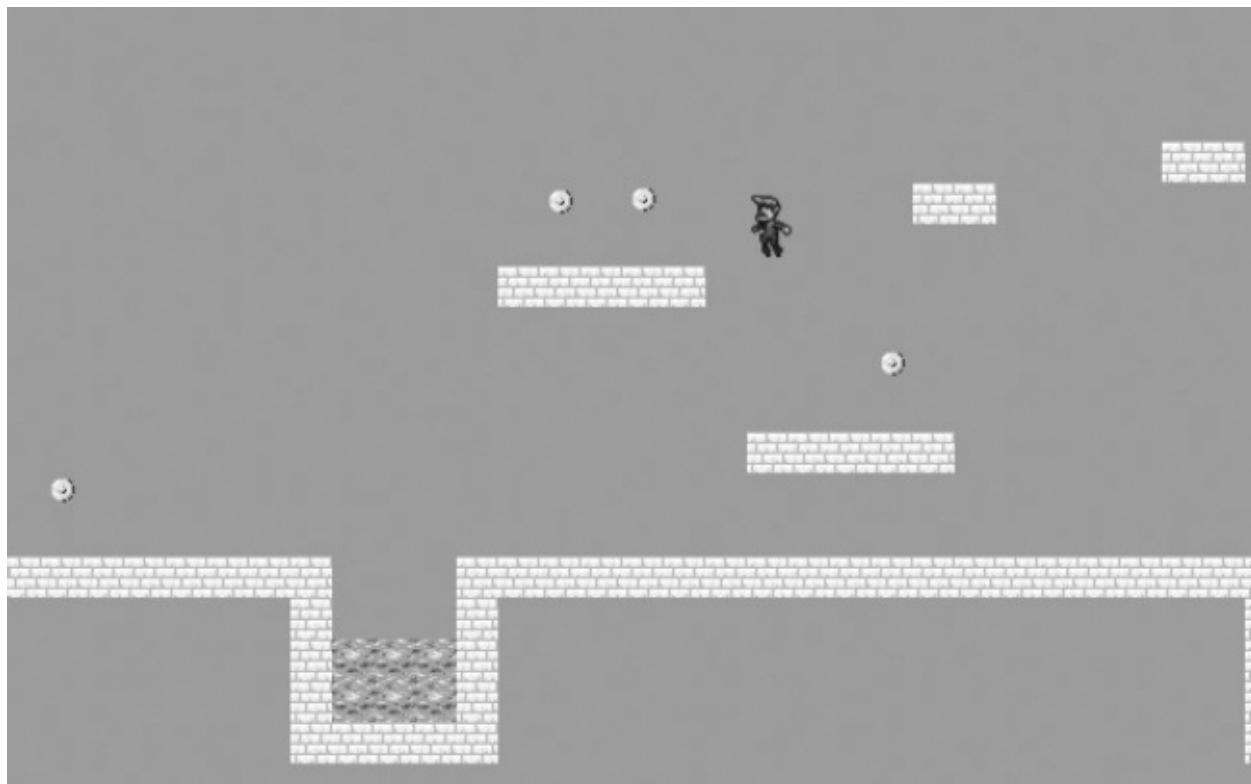
---

Per disegnare qualcosa che non è il giocatore, esaminiamo il suo tipo per trovare lo spostamento dello sprite corrispondente. La tessera per la lava è spostata di 20 pixel e lo

sprite per la moneta di 40 (il doppio della scala).

Dal momento che il punto (0,0) sull'area di disegno corrisponde all'angolo superiore sinistro dell'area di visualizzazione, e non del livello, dobbiamo sottrarre la posizione dell'area di visualizzazione per calcolare la posizione dell'attore. Qui possiamo usare anche `translate`, entrambe le soluzioni vanno bene.

Con questo, abbiamo finito il nuovo sistema di visualizzazione. Il gioco ha ora l'aspetto seguente:



## Scegliere un'interfaccia grafica

Quando volete produrre delle immagini nel browser, potete scegliere tra HTML semplice, SVG e `<canvas>`. Non esiste *una sola* soluzione che vada bene in tutte le situazioni: ciascuna alternativa presenta vantaggi e svantaggi.

Il codice HTML ha il vantaggio di essere semplice e di integrarsi bene col testo. Sia SVG, sia la tela virtuale vi permettono di disegnare del testo, ma senza darvi modo di posizionarlo o di mandarlo a capo se occupa più di una riga. In un'immagine basata su HTML, invece, è facile inserire blocchi di testo.

SVG produce immagini ben definite, nitide a qualunque livello di scala. È più difficile da usare dell'HTML, ma molto più potente.

Sia SVG, sia HTML costruiscono una struttura dati (DOM) che rappresenta l'immagine. Questo permette di modificare gli elementi dopo averli disegnati. Se dovete modificare ripetutamente una piccola parte di un'immagine più grande, in risposta alle azioni dell'utente o perché si tratta di un'animazione, farlo in un'area di disegno rischia di essere inutilmente dispendioso. Oltretutto, il DOM permette di registrare gestori degli

eventi su tutti gli elementi dell’immagine, comprese le forme tracciate con SVG, cosa che non è possibile sulla tela virtuale.

L’approccio basato sui pixel della tela virtuale, però, può essere vantaggioso quando si disegnano tanti piccoli elementi. Il fatto che non costruisca una struttura dati, ma si limiti a ridisegnare la stessa superficie di pixel, si traduce in un costo minore per forma.

Inoltre, ci sono degli effetti, come rendere una scena un pixel per volta (come si fa per tracciare un raggio di luce) o applicare delle trasformazioni con JavaScript (per esempio, effetti come la sfocatura o la distorsione), che si possono ottenere solo con una tecnologia a pixel.

In alcuni casi, si possono combinare diverse di queste tecniche. Per esempio, potete tracciare un grafico con SVG o <canvas> e riportare le informazioni testuali posizionando un elemento HTML sopra l’immagine.

Per le applicazioni più semplici, non ha molta importanza quel che scegliete. La seconda schermata, che abbiamo impostato in questo capitolo per il nostro gioco, si poteva realizzare con una qualunque di queste tecnologie grafiche in quanto non ha bisogno di testo, né di gestire le interazioni col mouse, né di lavorare con grandi quantità di elementi.

## Riepilogo

In questo capitolo, abbiamo parlato delle tecnologie per disegnare immagini nel browser e specialmente dell’elemento <canvas>.

Un nodo canvas rappresenta un’area del documento dove il programma può disegnare. Il disegno avviene attraverso un oggetto di contesto di disegno, creato col metodo `getContext`.

L’interfaccia di disegno 2D ci permette di riempire e assegnare un profilo a diverse forme. La proprietà `fillStyle` del contesto determina il riempimento delle forme. Le proprietà `strokeStyle` e `LineWidth` controllano lo stile e lo spessore delle linee.

Rettangoli e brani di testo si disegnano con una sola chiamata: i metodi `fillRect` e `strokeRect` tracciano rettangoli, mentre `fillText` e `strokeText` tracciano del testo. Per creare altre forme, dobbiamo prima impostare un tracciato.

Richiamando `beginPath` si inizia un nuovo tracciato, dove si hanno a disposizione diversi metodi per aggiungere linee e curve. Per esempio, `lineTo` aggiunge una linea retta. Quando il tracciato è finito, può essere riempito col metodo `fill` o contornato con `stroke`.

Per spostare pixel da un’immagine o da un’altra area di disegno si usa il metodo `drawImage`. Nell’impostazione predefinita, questo metodo disegna tutta l’immagine sorgente, ma accetta parametri per specificare la parte dell’immagine che volete copiare. Abbiamo usato questo metodo nel gioco, copiando le singole pose del personaggio da un’immagine che ne conteneva diverse.

Le trasformazioni vi permettono di disegnare la stessa forma in diversi orientamenti.

La trasformazione corrente del contesto 2D può essere modificata coi metodi `translate`, `scale` e `rotate`. Ogni trasformazione ha effetto sulle operazioni di disegno che la seguono. Col metodo `save` potete salvare lo stato di una trasformazione e recuperarlo col metodo `restore`.

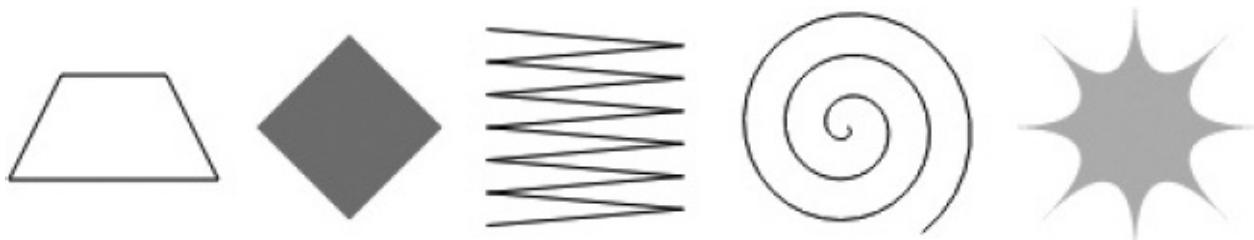
Nelle animazioni su tela, il metodo `clearRect` si usa per ripulire parte dell'area prima di ridisegnarla.

## Esercizi

### Forme geometriche

Scrivete un programma che disegni le seguenti forme geometriche su un elemento `<canvas>`:

1. Un trapezoide (un rettangolo con un lato più largo degli altri)
2. Un rombo rosso (un rettangolo ruotato a  $45^\circ$  o  $\frac{1}{4}\pi$  in radianti)
3. Una linea a zig zag
4. Una spirale costruita con 100 segmenti di linea retta
5. Una stella gialla



Quando arrivate alle ultime due, potete riguardare la spiegazione di `Math.cos` e `Math.sin` del [Capitolo 13](#), che spiega come disporre le coordinate in un cerchio con queste due funzioni.

Vi suggerisco di creare una funzione per ciascuna forma. Passate come parametri la posizione ed eventuali altre proprietà, come le dimensioni e il numero di vertici. L'alternativa, ossia di inserire i numeri nel codice direttamente, rischia di renderlo inutilmente difficile da leggere e modificare.

### Il grafico a torta

In precedenza in questo capitolo, abbiamo visto un esempio di come disegnare un grafico a torta. Modificate quel programma in modo che mostri il nome delle categorie vicino alla fetta che le rappresenta. Cercate un modo per posizionare il testo, che sia piacevole da vedere e funzioni anche per altre serie di dati. Potete presumere che le categorie rappresentino sempre almeno il 5% del totale, ossia che non ce ne saranno troppe, piccolissime, una attaccata all'altra.

Anche per questo esercizio, come nel precedente, potrebbero tornarvi buoni `Math.sin`

e `Math.cos`.

## **Una palla che rimbalza**

Usate la tecnica di `requestAnimationFrame`, descritta nei [Capitoli 13 e 15](#), per disegnare una scatola che contenga una palla che rimbalza. La palla si muove a velocità costante e rimbalza sulle pareti della scatola quando le colpisce.

## **Riflessioni precalcolate**

Un problema delle trasformazioni è che rallentano il disegno delle immagini bitmap. Per la grafica vettoriale, l'effetto è meno grave, in quanto bisogna trasformare solo pochi punti (per esempio, il centro di un cerchio) prima di poter proseguire col disegno. Nelle immagini bitmap, le trasformazioni si applicano invece alla posizione di ogni singolo pixel e, per quanto si possa sperare che i browser diventino più abili in futuro, i calcoli necessari allungano notevolmente i tempi richiesti per ridisegnarle.

In un gioco come il nostro, dove disegniamo un solo sprite trasformato, questo problema non si pone. Immaginate, però, di aver bisogno di centinaia di personaggi o migliaia di particelle che si spostano per effetto di un'esplosione.

Pensate a come possiamo disegnare un personaggio specchiato senza dover caricare nuovi file di immagine e senza dover richiamare le trasformazioni con `drawImage` per ogni fotogramma.

## HTTP

*Il sogno dietro al Web è di uno spazio dati comune dove comuniciamo scambiandoci informazioni. La sua universalità è essenziale: il fatto che un link ipertestuale possa puntare a qualunque cosa, sia essa personale, locale o globale, in forma appena abbozzata o perfezionata al massimo.*

Tim Berners-Lee, *The World Wide Web: A very short personal history*

Il protocollo HTTP (*Hypertext Transfer Protocol*), già citato nel [Capitolo 12](#), è il meccanismo attraverso il quale si scambiano dati sul World Wide Web. Questo capitolo descrive il protocollo in dettaglio e spiega come il codice JavaScript nei browser vi accede.

### Il protocollo

Se scrivete [eloquentjavascript.net/17\\_http.html](http://eloquentjavascript.net/17_http.html) nella barra degli indirizzi del vostro browser, il browser cerca per prima cosa l'indirizzo del server associato con [eloquentjavascript.net](http://eloquentjavascript.net) e tenta di aprire, con quel server, una connessione TCP sulla porta 80, la porta predefinita per il traffico HTTP. Se il server esiste e accetta la connessione, il browser invia qualcosa di simile a questo:

---

```
GET /17_http.html HTTP/1.1
Host: eloquentjavascript.net
User-Agent: Your browser's name
```

---

Quindi il server risponde, sulla stessa connessione:

---

```
HTTP/1.1 200 OK
Content-Length: 65585
Content-Type: text/html
Last-Modified: Wed, 09 Apr 2014 10:48:09 GMT
<!doctype html>
... il resto del documento
```

---

A questo punto, il browser recupera la parte della risposta che viene dopo la riga vuota e la visualizza come documento HTML.

I dati inviati dal client si chiamano *richiesta* (request) e cominciano con questa riga:

---

GET /17\_http.html HTTP/1.1

---

La prima parola indica il *metodo* della richiesta. GET significa che vogliamo *recuperare* la risorsa specificata. Altri metodi di uso comune sono DELETE per eliminare una risorsa, PUT per sostituirla e POST per inviarle informazioni. Notate che il server non deve soddisfare tutte le richieste che riceve. Se provate a connettervi con un sito Web qualunque e dare al suo server l'istruzione DELETE per la pagina che vi propone, è quasi certo che riceverete una risposta negativa.

La parte che segue il nome del metodo è il percorso che indica dove si trova la risorsa richiesta. Nei casi più semplici, una risorsa è solo un file sul server, ma il protocollo non si limita a questo. Una risorsa può essere qualunque cosa che si possa trasmettere come file. Molti server generano la risposta al volo: per esempio, se andate su [twitter.com/marijnjh](https://twitter.com/marijnjh), il server cerca nel database un utente con nome *marijnjh* e, se lo trova, genera al volo la pagina del suo profilo.

Dopo il percorso, la prima riga della richiesta specifica HTTP/1.1 per indicare la versione del protocollo utilizzata.

Anche la risposta del server si apre con un numero di versione, seguito dallo stato della risposta, prima come codice a tre cifre e quindi come stringa leggibile:

---

HTTP/1.1 200 OK

---

I codici di stato che iniziano con 2 indicano che la richiesta ha avuto successo. I codici che iniziano con 4 indicano che qualcosa non ha funzionato. Il codice di stato HTTP forse più famoso è 404, che indica che il server non ha trovato la risorsa richiesta. I codici che iniziano con 5 indicano che si è verificato un errore sul server, ma non c'era nulla di sbagliato nella richiesta in sé.

Dopo la prima riga di una richiesta o di una risposta, si possono trovare altre righe, dette *header*. Ogni riga di header è nella forma "nome: valore" e riporta informazioni aggiuntive sulla richiesta o sulla risposta. Ecco le righe di header nella risposta di esempio:

---

Content-Length: 65585

Content-Type: text/html

Last-Modified: Wed, 09 Apr 2014 10:48:09 GMT

---

Le prime due righe riportano le dimensioni e il tipo del documento richiesto (e trovato). In questo caso, si tratta di un documento HTML di 65.585 byte. La terza riga specifica data e ora dell'ultima modifica del documento.

In generale, sta al server o al client stabilire quali header riportare in una richiesta o in una risposta; solo alcuni sono obbligatori. In una richiesta è sempre preferibile indicare l'header Host, che specifica il nome dell'host, perché lo stesso server può gestire diversi

nomi di host su un solo indirizzo IP, e senza quell'indicazione, il server non può sapere a quale host il client vuole parlare.

Dopo gli header, richieste e risposte devono inserire una riga bianca prima del *corpo*, che contiene i dati trasmessi. Le richieste GET e DELETE non trasmettono dati ma PUT e POST lo fanno. Anche alcuni tipi di risposta, come quelle di errore, non richiedono un corpo.

## I browser e HTTP

Come abbiamo visto nell'esempio, il browser manda una richiesta quando inseriamo un URL nella barra degli indirizzi. Quando la pagina HTML richiesta contiene riferimenti ad altri file, per esempio immagini o file di JavaScript, vengono recuperati anche quei file.

Un sito Web di media complessità può facilmente arrivare ad avere tra 10 e 200 risorse. Per poterle recuperare rapidamente, i browser inviano diverse richieste simultaneamente e non aspettano le risposte una per volta. Questi documenti vengono sempre recuperati con richieste GET.

Le pagine HTML possono contenere moduli (*form*), che permettono all'utente di inviare al server le informazioni in essi richieste. Questo è un esempio di modulo:

---

```
<form method="GET" action="example/message.html">
  <p>Name: <input type="text" name="name"></p>
  <p>Message:<br><textarea name="message"></textarea></p>
  <p><button type="submit">Send</button></p>
</form>
```

---

Il codice descrive un form con due campi: uno piccolo, che chiede di inserire un nome, e uno più grande dove scrivere un messaggio. Quando fate clic sul pulsante Send, i dati che avete inserito nei campi del modulo vengono codificati in una *stringa di ricerca* (query string). Quando l'attributo del metodo per l'elemento `<form>` è GET (e quando manca), la stringa di ricerca viene attaccata in coda all'URL specificato per l'azione e il browser trasmette una richiesta GET a quell'URL.

---

```
GET /example/message.html?name=Jean&message=Yes%3F HTTP/1.1
```

---

L'inizio di una stringa di ricerca è indicato da un punto di domanda, seguito da coppie nome-valore che corrispondono agli attributi name sugli elementi del campo e al rispettivo contenuto. Per separare le coppie, si usa un carattere &.

Il messaggio codificato nell'URL appena riportato è "Yes?", anche se al posto del punto di domanda trovate del codice strano. Questo succede perché bisogna usare delle sequenze di escape al posto di alcuni caratteri. Uno di questi è il punto di domanda, che si rappresenta come %3F. Sembra ci sia una regola non scritta, che dice che ogni formato ha bisogno delle sue sequenze di escape specifiche. Nella *codifica degli URL* si usa un segno

di percentuale seguito da due cifre esadecimali per il codice del carattere. In questo caso, 3F, che corrisponde a 63 nella notazione esadecimale, è il codice per il punto di domanda. JavaScript offre le funzioni `encodeURIComponent` e `decodeURIComponent`, rispettivamente per codificare e decifrare questo formato.

---

```
console.log(encodeURIComponent("Hello & goodbye"));
// → Hello%20%26%20goodbye
console.log(decodeURIComponent("Hello%20%26%20goodbye"));
// → Hello & goodbye
```

---

Diversamente, se modifichiamo in `POST` l'attributo di `method=` per il modulo HTML visto nell'esempio precedente, la richiesta HTTP, elaborata quando si trasmette il modulo, userà questo metodo e inserirà la stringa di richiesta nel corpo della richiesta stessa, invece di aggiungerla all'URL.

---

```
POST /example/message.html HTTP/1.1
Content-length: 24
Content-type: application/x-www-form-urlencoded
name=Jean&message=Yes%3F
```

---

Per convenzione, il metodo `GET` si usa per le richieste che non hanno effetti collaterali, come per esempio le ricerche testuali. Le richieste che modificano qualcosa sul server, come quando si crea un nuovo account o quando si trasmette un messaggio, richiedono di solito altri metodi, tra cui `POST`. Il software del client, di solito il browser, non effettua richieste `POST` a casaccio, ma può effettuare richieste `GET` implicite, per esempio per precaricare una risorsa di cui l'utente potrebbe aver bisogno in futuro.

Nel prossimo capitolo torneremo sui moduli e su come si usa JavaScript per gli script relativi.

## XMLHttpRequest

L'interfaccia attraverso la quale gli script JavaScript sul browser possono inviare richieste HTTP si chiama `XMLHttpRequest` (proprio con questa strana alternanza di maiuscole e minuscole). Fu progettata da Microsoft alla fine degli anni Novanta per il suo browser, Internet Explorer. A quell'epoca, il formato file XML era molto popolare nelle applicazioni per ufficio, un mondo dove Microsoft è sempre stata di casa. Era talmente popolare, che l'acronimo XLM finì persino davanti al nome di quell'interfaccia per il protocollo HTTP, che col formato XML non ha nulla a che fare.

Il nome `XMLHttpRequest` non è tuttavia del tutto insensato, in quanto l'interfaccia consente di tradurre in XML i documenti ottenuti in risposta. Peccato che fare tutt'uno di due concetti ben distinti (effettuare una richiesta e analizzare la risposta) sia una pessima scelta di impostazione!

Mettendo a disposizione l'interfaccia XMLHttpRequest con Internet Explorer, Microsoft permise ai programmati di svolgere operazioni mai fatte prima con JavaScript. Per esempio, divenne possibile proporre un elenco di suggerimenti quando l'utente cominciava a scrivere in un campo testo, perché lo script trasmetteva al server, via HTTP, i caratteri digitati. Il server poteva poi leggere un database per trovare i termini che corrispondevano all'input parziale e rimandarli all'utente. Questa funzionalità era spettacolare: allora, era normale dover attendere che venisse ricaricata tutta la pagina per ciascuna delle interazioni col sito.

L'altro browser allora più usato, Mozilla (poi Firefox), non poteva rimanere indietro. Per permettere comportamenti simili, gli sviluppatori di Mozilla copiarono l'interfaccia, con tanto di nome finto. Anche la generazione di browser successiva seguì l'esempio e oggi XMLHttpRequest è diventata un'interfaccia standard di fatto.

## Inviare una richiesta

Per effettuare una semplice richiesta, creiamo un oggetto request col costruttore XMLHttpRequest e richiamiamo i suoi metodi open e send:

```
var req = new XMLHttpRequest();
req.open("GET", "example/data.txt", false);
req.send(null);
console.log(req.responseText);
// → This is the content of data.txt [Questo è il contenuto di data.txt]
```

Il metodo open configura la richiesta. In questo caso, effettuiamo una richiesta GET per il file *example/data.txt*. Gli URL che non iniziano col nome di un protocollo (come *http:*) sono relativi, il che significa che vengono interpretati a partire dal documento corrente. Quando iniziano con una barra (/), sostituiscono il percorso corrente, che è la parte che viene dopo il nome del server. Quando non iniziano con una barra, la parte del percorso corrente fino a e compreso il suo ultimo carattere barra (/) viene anteposta all'URL relativo.

Una volta aperta la richiesta, possiamo trasmetterla col metodo send, col corpo della richiesta come argomento. Per le richieste GET possiamo trasmettere null. Se il terzo argomento di open fosse false, send restituirebbe solo dopo aver ricevuto la risposta alla nostra richiesta. Per recuperare il corpo della risposta, esaminiamo la proprietà *responseText* dell'oggetto *request*.

Possiamo estrarre da quest'oggetto anche le altre informazioni comprese nella risposta. Al codice di stato si accede attraverso la proprietà *status*, mentre il testo leggibile corrispondente si trova in *statusText*. Gli header si esaminano con *getResponseHeader*.

```
var req = new XMLHttpRequest();
req.open("GET", "example/data.txt", false);
```

```
req.send(null);
console.log(req.status, req.statusText);
// → 200 OK
console.log(req.getResponseHeader("content-type"));
// → text/plain
```

---

I nomi degli header non distinguono tra maiuscole e minuscole. Di solito, sono scritti con una lettera maiuscola all'inizio di ogni parola, come in “Content-Type”, ma anche “content-type” e “cOnTeNt-TyPe” si riferiscono allo stesso header.

Il browser aggiunge automaticamente alcuni header di richiesta, tra cui “Host” e quelli che gli servono per calcolare le dimensioni del corpo. Col metodo `setRequestHeader` potete poi specificare gli header che vi interessano: ciò, tuttavia, è necessario solo per esigenze speciali e richiede la cooperazione del server, in quanto i server possono sempre ignorare gli header che non sanno come trattare.

## Richieste asincrone

Negli esempi che abbiamo visto, la richiesta termina quando la chiamata a `send` restituisce. Il che è un bene, perché significa che sono immediatamente disponibili proprietà come `responseText`. Significa anche, però, che il programma resta in sospeso finché browser e server rimangono in comunicazione. Se la connessione è cattiva, il server è lento o il file molto pesante, questo può richiedere un bel po' di tempo. Ancor peggio, poiché non possono scattare gestori di eventi fintanto che il programma è in sospeso, tutto il documento cessa di rispondere.

Se passiamo `true` come terzo argomento a `open`, la richiesta è *asincrona*. Ciò significa che quando richiamiamo `send`, l'unica cosa che succede immediatamente è che la richiesta viene messa in coda per essere trasmessa. Il programma può continuare e il browser può trasmettere e ricevere i dati in background.

Non possiamo, però, accedere alla risposta finché la richiesta non sia stata completata: ci serve pertanto un meccanismo che ci avvisi quando i dati sono disponibili.

Per questo, dobbiamo rimanere in attesa di un evento “load” sull'oggetto `request`.

---

```
var req = new XMLHttpRequest();
req.open("GET", "example/data.txt", true);
req.addEventListener("load", function() {
  console.log("Done:", req.status);
});
req.send(null);
```

---

Proprio come con `requestAnimationFrame` del [Capitolo 15](#), dobbiamo ricorrere a uno stile di programmazione asincrono, avvolgendo in una funzione le operazioni che vanno

svolte dopo la richiesta e pianificandone la chiamata al momento opportuno. Ci torneremo sopra più avanti.

## Recuperare i dati XML

Quando la risorsa recuperata da un oggetto XMLHttpRequest è un documento XML, la proprietà responseXML dell'oggetto mantiene una rappresentazione di quel documento sotto forma di struttura dati. Quella rappresentazione funziona sostanzialmente come nel modello DOM descritto nel [Capitolo 13](#), con la differenza che non ha funzionalità specifiche per l'HTML, come le proprietà di stile. L'oggetto mantenuto da responseXML corrisponde all'oggetto document. La sua proprietà documentElement fa riferimento al tag esterno del documento XML. Nel documento seguente (*example/fruit.xml*), si tratta del tag <fruits>:

---

```
<fruits>
  <fruit name="banana" color="yellow"/>
  <fruit name="lemon" color="yellow"/>
  <fruit name="cherry" color="red"/>
</fruits>
```

---

Ecco come possiamo recuperare quel file:

```
var req = new XMLHttpRequest();
req.open("GET", "example/fruit.xml", false);
req.send(null);
console.log(req.responseXML.querySelectorAll("fruit").length);
// → 3
```

---

I documenti XML si usano per scambiare col server informazioni strutturate. La loro forma (tag annidati in altri tag) si presta molto bene, sicuramente meglio dei file di testo normali, a memorizzare moltissimi tipi di dati. Tuttavia, l'interfaccia del DOM non è molto comoda per estrarre dati e i documenti XML sono spesso verbosi. Pertanto, è spesso preferibile comunicare con dati JSON, che sono più semplici da leggere e da scrivere, sia per i programmi, sia per i programmatorei.

---

```
var req = new XMLHttpRequest();
req.open("GET", "example/fruit.json", false);
req.send(null);
console.log(JSON.parse(req.responseText));
// → {banana: "yellow", lemon: "yellow", cherry: "red"}
```

---

## Protezioni sandbox in HTTP

Come sempre, effettuare delle richieste HTTP negli script per pagine Web solleva il problema della sicurezza. La persona che controlla lo script potrebbe avere interessi diversi da quelli di chi vi accede. Più specificamente, se vado a vedere il sito (ipotetico) [bandadiladri.org](http://bandadiladri.org), non voglio che gli script su quel server trasmettano richieste a [lamiabanca.com](http://lamiabanca.com), usando i dati identificativi che trovano nel mio browser, magari per trasferire tutti i miei soldi su un conto corrente di un ladro a caso.

I siti Web possono difendersi da questo tipo di attacchi, ma ciò richiede del lavoro e molti siti non lo fanno. Per questo, i browser ci proteggono impedendo agli script di effettuare richieste HTTP ad altri *domini* (nomi come [bandadiladri.org](http://bandadiladri.org) e [lamiabanca.com](http://lamiabanca.com)).

Il che può essere un bel problema quando dobbiamo realizzare sistemi che hanno bisogno di accesso a diversi domini per ragioni legittime. Per fortuna, per indicare specificamente al browser che è ammesso ricevere la richiesta da altri domini, i server possono inserire nella risposta una riga header come la seguente

---

Access-Control-Allow-Origin: \*

---

## Astrazione delle richieste

Nel [Capitolo 10](#), per la realizzazione del sistema di moduli AMD, avevamo usato un’ipotetica funzione `backgroundReadFile`, che accettava un nome file e una funzione da richiamare col contenuto del file, appena finito di recuperarlo. Ecco come si può impostare semplicemente una funzione come quella:

---

```
function backgroundReadFile(url, callback) {  
    var req = new XMLHttpRequest();  
    req.open("GET", url, true);  
    req.addEventListener("load", function() {  
        if (req.status < 400)  
            callback(req.responseText);  
    });  
    req.send(null);  
}
```

---

Questa semplice astrazione semplifica l’uso di `XMLHttpRequest` per semplici richieste GET. Se dovete scrivere un programma che deve effettuare richieste HTTP, è una buona idea ricorrere a funzioni ausiliarie per non dover ripetere tutte le volte il brutto sistema di richieste di `XMLHttpRequest`.

Il nome dell’argomento della funzione, `callback`, viene spesso usato per descrivere

questo tipo di funzioni. Le *funzioni di callback*, o di richiamo, servono per permettere al codice di “richiamarci” più tardi.

Non è difficile scrivere una funzione di utility per HTTP specifica per le vostre esigenze. Quella appena riportata effettua solo richieste GET e non ci dà alcun controllo sugli header, né sul corpo della richiesta. Potete scriverne una simile per richieste POST o una più generica, che permetta diversi tipi di richiesta. Funzioni per avvolgere XMLHttpRequest si trovano anche in molte librerie JavaScript.

Il problema più grosso con la funzione wrapper di prima sta nel modo in cui gestisce gli errori: quando la richiesta restituisce un codice di stato che indica un errore (dal 400 in su), la funzione non fa nulla. In alcuni casi questo comportamento può essere appropriato, ma immaginate di avere un indicatore di attività sulla pagina, che indica che stiamo recuperando dei dati. Se la richiesta non ha esito perché il server si blocca o la connessione si interrompe, la pagina non cambia e sembra che continui a lavorare. L’utente aspetta per un po’, poi si scoccia e pensa che il sito sia inutile e mal fatto.

Dovremmo avere la possibilità di essere avvertiti quando la richiesta non ha esito, in modo da prendere le misure necessarie. Per esempio, potremmo far sparire il messaggio che indica che stiamo caricando e far sapere all’utente che qualcosa è andato storto.

La gestione degli errori nel codice asincrono è ancora più complicata che col codice sincrono. Poiché spesso abbiamo bisogno di rimandare parte del nostro lavoro e inserirlo in una funzione di callback, l’ambito dei blocchi try perde significato. Nel codice seguente, l’eccezione non sarà catturata perché la chiamata a backgroundReadFile restituisce immediatamente. Il flusso di controllo lascia quindi il blocco try e la funzione in esso specificata verrà richiamata solo più tardi.

---

```
try {
    backgroundReadFile("example/data.txt", function(text) {
        if (text != "expected")
            throw new Error("That was unexpected");
    });
} catch (e) {
    console.log("Hello from the catch block");
}
```

---

Per gestire le richieste che non hanno esito, dobbiamo passare alla funzione wrapper, una funzione aggiuntiva, da richiamare quando qualcosa va storto. Oppure, possiamo seguire la convenzione che se la richiesta non ha esito, viene passato un argomento che descrive il problema alla funzione callback. Ecco un esempio:

---

```
function getURL(url, callback) {
    var req = new XMLHttpRequest();
    req.open("GET", url, true);
```

```
req.addEventListener("load", function() {
  if (req.status < 400)
    callback(req.responseText);
  else
    callback(null, new Error("Request failed: " +
      req.statusText));
});

req.addEventListener("error", function() {
  callback(null, new Error("Network error"));
});

req.send(null);
}
```

---

Abbiamo aggiunto un gestore per l'evento "error", che ci sarà notificato quando la richiesta non ha esito. Quando la richiesta riceve in risposta un codice di stato che indica un errore, richiamiamo la funzione callback con un argomento `error`.

Il codice che fa uso di `getURL` dovrà poi verificare se c'è stato un errore e, se lo trova, gestirlo.

---

```
getURL("data/nonsense.txt", function(content, error) {
  if (error != null)
    console.log("Failed to fetch nonsense.txt: " + error);
  else
    console.log("nonsense.txt: " + content);
});
```

---

Questo codice però non aiuta quando si verificano delle eccezioni. Quando si collegano tra loro diverse azioni asincrone, un'eccezione in qualunque punto della catena risalirà fino al livello superiore e interromperà la catena di azioni che avete previsto, a meno che non avvolgiate ciascuna funzione di gestione nel suo blocco `try/catch`.

## Promesse

Nei progetti complessi, non è facile scrivere correttamente del codice asincrono in semplice stile callback. È facile dimenticare di verificare un errore o permettere a un'eccezione inaspettata di interrompere bruscamente il programma. Non solo, ma ci vuole molta, molta pazienza per impostare una corretta gestione degli errori quando ogni errore può attraversare più funzioni di callback e blocchi `catch`.

Ci sono stati molti tentativi di risolvere questi problemi con delle nuove astrazioni. Uno dei sistemi più efficaci è quello cosiddetto delle *promesse* (*promises*). Le promesse

avvolgono un'azione asincrona in un oggetto, che può essere passato ad altre parti del codice e istruito a svolgere determinate attività quando l'azione si completa o non ha esito. Quest'interfaccia farà parte della prossima versione del linguaggio JavaScript e può già essere usata come libreria.

L'interfaccia per le promesse non è sempre facile da capire, ma è molto potente. In questo capitolo la descrivo molto sommariamente. Potete trovare ulteriori informazioni su [www.promisejs.org](http://www.promisejs.org).

Per creare un oggetto `promise`, richiamiamo il costruttore `Promise` passandogli una funzione che inizializza l'azione asincrona. Il costruttore richama quella funzione con due argomenti, che sono a loro volta funzioni. La prima è da richiamare quando l'azione si completa con successo, la seconda quando l'azione non va a buon fine.

Ecco una nuova versione della funzione wrapper per richieste GET, che questa volta restituisce una promessa. Questa volta la chiamiamo solo `get`:

---

```
function get(url) {
  return new Promise(function(succeed, fail) {
    var req = new XMLHttpRequest();
    req.open("GET", url, true);
    req.addEventListener("load", function() {
      if (req.status < 400)
        succeed(req.responseText);
      else
        fail(new Error("Request failed: " + req.statusText));
    });
    req.addEventListener("error", function() {
      fail(new Error("Network error"));
    });
    req.send(null);
  });
}
```

---

Notate che l'interfaccia per la funzione è ora molto più semplice: riceve un URL e restituisce una promessa. La promessa funge da *handle* (riferimento astratto) per i risultati della richiesta. Ha un metodo `then` che potete richiamare con due funzioni: una per quando la chiamata ha successo e una per quando non ha esito.

---

```
get("example/data.txt").then(function(text) {
  console.log("data.txt: " + text);
}, function(error) {
  console.log("Failed to fetch data.txt: " + error);
```

```
});
```

---

Fin qui, questo è solo un altro modo per esprimere la stessa cosa che abbiamo già descritto. È solo quando avrete bisogno di impostare delle azioni a catena che le promesse faranno la differenza.

Il richiamo a `then` produce una nuova promessa, il cui risultato (il valore passato ai gestori del successo) dipende dal valore di restituzione della prima funzione passata a `then`. Questa funzione può restituire un'altra promessa per indicare che c'è dell'altro lavoro asincrono da svolgere. In questo caso, la promessa restituita da `then` rimane in attesa della promessa restituita dalla funzione di gestione e avrà successo o no con lo stesso valore quando si risolve. Quando la funzione di gestione restituisce un valore diverso da `promise`, la promessa restituita da `then` ha subito successo con quel valore come risultato.

Ciò significa che potete usare `then` per trasformare il risultato di una promessa. L'esempio seguente restituisce una promessa il cui risultato è il contenuto dell'URL dato, sotto forma di struttura dati JSON:

```
functiongetJSON(url) {  
    return get(url).then(JSON.parse);  
}
```

---

L'ultima chiamata a `then` non specifica un gestore in caso di fallimento: ciò è permesso. L'errore sarà passato alla promessa restituita da `then`, che è esattamente quel che vogliamo: `getJSON` non sa che cosa fare quando qualcosa non funziona, ma la funzione che la richiama lo sa.

Come esempio di dimostrazione del funzionamento delle promesse, impostiamo un programma che recupera una serie di file JSON dal server e, intanto che lavora, visualizza sullo schermo la parola *Loading...* (Sto caricando...). Il file JSON contiene dati su alcune persone, con dei link a file che rappresentano altre persone in proprietà come `father` (padre), `mother` (madre) o `spouse` (coniuge).

Vogliamo recuperare il nome della madre del coniuge di `example/bert.json`. Se qualcosa andasse storto, vogliamo eliminare il testo che indica che stiamo caricando qualcosa e mostrare al suo posto un messaggio di errore. Ecco come potremmo farlo con delle promesse:

```
<script>  
  
function showMessage(msg) {  
    var elt = document.createElement("div");  
    elt.textContent = msg;  
    return document.body.appendChild(elt);  
}  
  
var loading = showMessage("Loading...");
```

```

getJSON("example/bert.json").then(function(bert) {
    return getJSON(bert.spouse);
}).then(function(spouse) {
    return getJSON(spouse.mother);
}).then(function(mother) {
    showMessage("The name is " + mother.name);
}).catch(function(error) {
    showMessage(String(error));
}).then(function() {
    document.body.removeChild/loading);
});
</script>

```

---

Il programma risulta abbastanza compatto e facile da leggere. Il metodo `catch` è simile a `then`, salvo che si aspetta solo un gestore per il fallimento e lascerà passare il risultato senza modifiche in caso di successo. In modo analogo alla clausola `catch` per le dichiarazioni `try`, il controllo continua senza scossoni dopo che l'errore è stato catturato. Pertanto, viene sempre eseguita l'istruzione `then` finale, che elimina il messaggio “*Loading...*”, anche quando qualcosa va storto.

Potete pensare all’interfaccia delle promesse come a un linguaggio a se stante per il flusso di controllo asincrono. Le chiamate ai metodi e le espressioni di funzione necessarie per ottenere questo risultato producono del codice un po’ strano, anche se mai tanto strano quanto sarebbe se ci occupassimo direttamente della gestione di tutti gli errori.

## Il valore del protocollo HTTP

Per i sistemi che richiedono la comunicazione tra un programma JavaScript in esecuzione nel browser (lato client) e un programma sul server (lato server), si può scegliere tra diversi modelli di impostazione delle comunicazioni.

Uno dei più comuni è quello delle *chiamate di procedure remote*, rpc (remote procedure calls). In questo modello, le comunicazioni seguono la struttura di normali chiamate di funzione, salvo che la funzione viene eseguita sull’altra macchina. Richiamare una funzione effettua una richiesta al server, nella quale sono specificati nome e argomenti della funzione. La risposta a quella richiesta contiene il valore restituito.

Pensando in termini di chiamate di procedure remote, il protocollo HTTP è solo un veicolo per le comunicazioni e potete impostare un livello di astrazione che lo nasconde completamente.

Un’altra soluzione è di pensare alle comunicazioni in termini di risorse e metodi HTTP. Invece di una chiamata di procedura remota `addUser`, usate una richiesta PUT per `/users/larry`. Invece di codificare le proprietà di quell’utente in argomenti della

funzione, definite un formato documento o usate un formato esistente per rappresentare un utente. Il corpo della richiesta `PUT` per creare una nuova risorsa è dunque solo quel documento. La risorsa si recupera effettuando una richiesta `GET` al suo URL (per esempio, `/user/larry`), che restituisce il documento che rappresenta la risorsa.

Questa seconda soluzione fa un uso migliore di alcune delle funzionalità offerte da HTTP, qual è il supporto per mantenere le risorse in memoria di cache (salvandone una copia sul lato client). Può anche migliorare la coerenza della vostra interfaccia, in quanto pensare in termini di risorse è più semplice che far ordine tra funzioni.

## La sicurezza e HTTPS

I dati trasmessi su Internet seguono spesso dei percorsi lunghi e pericolosi. Per arrivare a destinazione, devono saltare da un nodo all'altro, passando da reti di tutti i tipi: da punti di accesso pubblici, a reti sulle quali vegliano organizzazioni commerciali e statali. In qualunque punto del percorso, i dati possono essere ispezionati o addirittura modificati.

Se è importante che qualcosa rimanga segreto, per esempio la password del vostro account di posta elettronica, o che arrivi a destinazione senza variazioni, come per il numero di conto sul quale trasferire soldi dallo sportello online della vostra banca, allora il semplice HTTP non è sufficiente.

Il protocollo sicuro HTTP, dove gli URL iniziano con `https://`, avvolge il traffico HTTP in modo che sia più difficile da leggere e da manipolare. Per prima cosa, il client verifica che il server sia effettivamente quel che dice di essere: per questo, richiede al server di mostrare un certificato crittografico rilasciato da un'autorità riconosciuta dal browser. Tutti i dati trasmessi sulla connessione vengono poi crittografati per non essere letti o modificati durante il transito.

Quando funziona correttamente, HTTPS blocca pertanto sia chi cerca di impersonare il server col quale volete comunicare, sia chi vuole ficcare il naso nei vostri affari. Non è perfetto e si sono verificati diversi incidenti, dove non è servito, per colpa di certificati falsificati o rubati, o di software che non funzionava. Ciononostante, aggirare maliziosamente il protocollo HTTP è un gioco da ragazzi, mentre forzare HTTPS richiede sforzi che solo i governi e le organizzazioni criminali più sofisticate possono sperare di compiere.

## Riepilogo

In questo capitolo, abbiamo visto che HTTP è un protocollo per accedere a risorse su Internet. Un *client* trasmette una richiesta, che contiene un metodo (di solito `GET`) e un percorso che identifica una risorsa. Il *server* decide che cosa fare con la richiesta e risponde con un codice di stato e un corpo della risposta. Sia la richiesta, sia la risposta possono contenere righe di header con altre informazioni.

I browser trasmettono richieste `GET` per recuperare le risorse necessarie per

visualizzare una pagina Web. Le pagine Web possono contenere moduli (form), che permettono di trasmettere i dati inseriti dall’utente insieme alla richiesta effettuata quando il modulo viene registrato. Di questo argomento ripareremo nel prossimo capitolo.

L’interfaccia attraverso la quale il codice JavaScript dei browser può effettuare richieste HTTP si chiama XMLHttpRequest. Potete ignorare il significato della sigla “XML”, purché non dimentichiate di scriverla nel nome dell’interfaccia. XMLHttpRequest si usa in due modi: sincrono, dove tutto si blocca finché la richiesta non è conclusa, e asincrono, che richiede un gestore di eventi per scoprire quando arriva la risposta. La soluzione asincrona è generalmente la migliore. Ecco come si può effettuare una richiesta:

---

```
var req = new XMLHttpRequest();
req.open("GET", "example/data.txt", true);
req.addEventListener("load", function() {
    console.log(req.status);
});
req.send(null);
```

---

La programmazione asincrona è piuttosto complessa. Quella delle *promesse* è un’interfaccia che la semplifica un pochino, incanalando le condizioni di errore e le eccezioni verso il gestore più appropriato e sostituendo con delle astrazioni alcuni degli elementi più ripetitivi e soggetti a errore di questo stile di programmazione.

## Esercizi

### Negoziazione dei contenuti

Una delle cose che HTTP può fare, ma di cui non abbiamo parlato in questo capitolo, si chiama *negoziazione dei contenuti*. Si usa l’header Accept di una richiesta per indicare al server il tipo di documento che il client vuole ottenere. Molti server ignorano questo header, ma alcuni sono in grado di riconoscere diversi modi per codificare una risorsa: pertanto, leggono l’header Accept e trasmettono la risorsa che il client preferisce.

L’URL [eloquentjavascript.net/author](http://eloquentjavascript.net/author) è configurato in modo da rispondere con testo semplice, HTML o JSON a seconda della richiesta ricevuta dal client. Questi formati sono identificati dai *tipi di media* standard: text/plain, text/html e application/json.

Trasmettete delle richieste per recuperare tutti e tre i formati di questa risorsa. Usate il metodo setRequestHeader dell’oggetto XMLHttpRequest per impostare la riga di header Accept per uno dei tipi di media riportati sopra. Ricordate di impostare l’header dopo open e prima di send.

Infine, provate a richiedere il tipo di media application/rainbows+unicorns e vedete che cosa succede.

## **Aspettare più promesse**

Il costruttore `Promise` ha un metodo `all` che, dato un array di promesse, ne restituisce una che rimane in attesa che tutte le altre terminino e, alla fine di tutte le operazioni, restituisce un array di valori. Se una delle promesse dell'array non ha esito, fallisce anche la promessa restituita, col valore corrispondente nella promessa che non ha avuto successo.

Provate a realizzare qualcosa di simile come funzione regolare `all`.

Notate che dopo che una promessa è risolta (ha avuto successo o non ha avuto esito), non può più fare nulla ed eventuali chiamate successive vengono ignorate. Questo può semplificare il modo in cui impostate il fallimento della promessa.

## MODULI E CAMPI DI MODULI

*Or bene, festeggisi oggi un sì bell'accordo, e, come tuo, io ti servirò di mia mano alla mensa. Ma, di grazia, un sol motto! — Dalla vita alla morte, non vorreste farmi una copia di righe?*

Mefistofele, nel *Faust* di Goethe  
[traduzione di Giovita Scalvini, 1835]

Abbiamo accennato nel capitolo precedente che i moduli permettono di trasmettere via HTTP dati forniti dall'utente. I relativi elementi HTML ebbero origine per il Web prima di JavaScript e davano per scontato che l'interazione col server portasse sempre a una nuova pagina Web.

Sebbene gli elementi dei moduli facciano parte del DOM come il resto della pagina, gli elementi del DOM che rappresentano i campi dei moduli offrono proprietà ed eventi che non esistono per altri elementi. Tra l'altro, questi permettono di ispezionare e controllare i campi di input con programmi JavaScript, aggiungere funzionalità al modulo o usare moduli e relativi campi come mattoni per costruire applicazioni in JavaScript.

### I campi

Un modulo per il Web consiste in un numero arbitrario di campi di input racchiusi in un tag `<form>`. La sintassi HTML prevede diversi tipi di campi, che vanno da semplici caselle on/off a menu a discesa e campi per l'inserimento di testo. In questo libro non approfondiamo il discorso sui tipi di campi, ma vi diamo una panoramica generale.

Alcuni tipi di campi usano il tag `<input>`, il cui attributo viene usato per selezionare lo stile del campo. Ecco alcuni tipi di `<input>` di uso comune:

text	Un campo di testo su una sola riga
password	Come text, ma con i caratteri nascosti (sostituiti da asterischi)
checkbox	Una casella acceso/spento
radio	Componente di un campo a più scelte
file	Permette all'utente di scegliere un file dal computer

I campi non devono necessariamente essere racchiusi in un tag `<form>` e potete metterli dove volete sulla pagina. I singoli campi non possono essere trasmessi (solo il modulo nel suo insieme può esserlo); del resto, quando li usiamo per rispondere all'input

con JavaScript, non sempre vogliamo che ciò accada.

---

```
<p><input type="text" value="abc"> (text)</p>
<p><input type="password" value="abc"> (password)</p>
<p><input type="checkbox" checked> (checkbox)</p>
<p><input type="radio" value="A" name="choice">
    <input type="radio" value="B" name="choice" checked>
    <input type="radio" value="C" name="choice"> (radio)</p>
<p><input type="file" checked> (file)</p>
```

---

I campi creati col codice HTML sopra riportato hanno questo aspetto:

The image shows a screenshot of a web browser displaying five different input types. 1. A text input field containing the text "abc" with the label "(text)" to its right. 2. A password input field containing three dots "..." with the label "(password)" to its right. 3. A checkbox input field with a checked checked state, accompanied by the label "(checkbox)". 4. Three radio button inputs, where the third one is checked, accompanied by the label "(radio)". 5. A file input field with a "Choose File" button and the file path "snippets.txt" to its right, with the label "(file)" to its right.

L’interfaccia JavaScript per questi elementi dipende dal loro tipo. Ne ripareremo in dettaglio più avanti in questo capitolo.

I campi di testo su più righe hanno un loro tag, `<textarea>`, sostanzialmente perché non ha molto senso usare un attributo per specificare un valore che può occupare più righe. Il tag `<textarea>` richiede un tag di chiusura specifico, `</textarea>`, e il suo valore è il testo inserito tra i due tag e non l’attributo `value`.

---

```
<textarea>
one
two
three
</textarea>
```

---

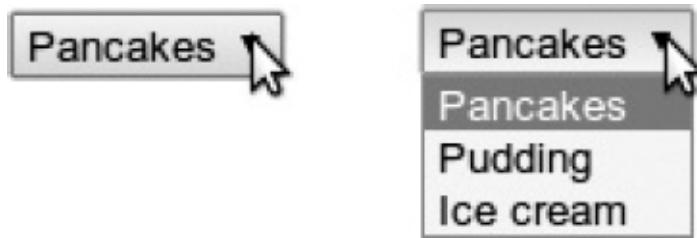
Infine, col tag `<select>` si imposta un campo che consente all’utente di scegliere una o più opzioni predefinite.

---

```
<select>
    <option>Pancakes</option>
    <option>Pudding</option>
    <option>Ice cream</option>
</select>
```

---

Il campo viene reso come segue:



Quando cambia il valore di un campo, scatta un evento "change".

## Campi attivi

Diversamente dagli altri elementi dei documenti HTML, i campi dei moduli possono *essere attivati* dalla tastiera: quando l'utente fa clic sul campo, o quando lo si attiva in altro modo, esso diventa l'elemento attivo e il destinatario principale dell'input da tastiera.

Se un documento ha un campo di tipo testo, i caratteri digitati vi finiscono solo se il campo è attivo. Altri campi rispondono in modo diverso agli eventi da tastiera. Per esempio, in un menu <select> il focus si sposta sull'opzione che contiene il testo digitato dall'utente e risponde ai tasti freccia spostandosi in su e in giù lungo l'elenco di opzioni.

Il focus si controlla in JavaScript coi metodi `focus` e `blur`. Il primo sposta il focus all'elemento sul quale lo richiamiamo, mentre il secondo lo disattiva. Il valore di `document.activeElement` corrisponde all'elemento che è attivo in questo momento.

---

```
<input type="text">
<script>
  document.querySelector("input").focus();
  console.log(document.activeElement.tagName);
  // → INPUT
  document.querySelector("input").blur();
  console.log(document.activeElement.tagName);
  // → BODY
</script>
```

In alcuni casi, si presume che l'utente voglia interagire immediatamente col modulo. JavaScript può attivare il campo quando si carica il documento; ma la sintassi HTML offre anche l'attributo `autofocus`, che produce lo stesso effetto indicando al browser quel che vogliamo ottenere. Ciò consente al browser di disattivare il comportamento predefinito quando ciò non fosse appropriato, come per esempio quando l'utente avesse scelto di attivare qualche altro elemento.

---

```
<input type="text" autofocus>
```

I browser tradizionalmente consentono all’utente anche di spostare il focus lungo il documento attraverso il tasto TAB. Con l’attributo tabindex, possiamo modificare l’ordine in cui gli elementi vengono attivati. Nel documento seguente, il focus passa dall’input di testo al pulsante OK invece di passare prima dal link (*help*).

---

```
<input type="text" tabindex=1> <a href="."> (help)</a>  
<button onclick="console.log('ok')" tabindex=2>OK</button>
```

---

Di default, gli elementi HTML diversi dai campi di un modulo non possono ricevere il focus: possiamo però aggiungere un attributo tabindex a qualunque elemento per renderlo attivabile.

## Campi non abilitati

Tutti i campi dei moduli possono essere *disattivati* con l’attributo disabled, che esiste anche come proprietà sul relativo oggetto DOM.

---

```
<button>I'm all right</button>  
<button disabled>I'm out</button>
```

---

I campi che non sono abilitati non possono ricevere il focus, né essere modificati, e si distinguono dai campi abilitati perché vengono resi dal browser in un grigio più pallido.



Quando in un programma avete bisogno di gestire un’azione, causata da un pulsante o da un altro componente grafico, che prevede comunicazioni col server e può pertanto richiedere tempo, potete disattivare l’elemento finché l’azione ha avuto termine. In questo modo, se l’utente si stanca di aspettare e preme di nuovo il pulsante, non fa scattare un’altra volta la stessa azione.

## Il modulo nel suo insieme

Quando un campo è racchiuso in un elemento `<form>`, l’elemento DOM corrispondente ha una proprietà `form` collegata all’elemento `form` del DOM. A sua volta, l’elemento `<form>` ha una proprietà `elements` che contiene una collezione simile a un array dei campi in esso contenuti.

L’attributo name del campo determina come sarà identificato il suo valore quando si invia il modulo per l’elaborazione. Può essere usato anche come nome di proprietà quando si accede alle proprietà dell’elemento, operando sia come oggetto simile a un array (accessibile per numero), sia come mappa (accessibile per nome).

---

```
<form action="example/submit.html">
```

```
Name: <input type="text" name="name"><br>
Password: <input type="password" name="password"><br>
<button type="submit">Log in</button>
</form>
<script>
  var form = document.querySelector("form");
  console.log(form.elements[1].type);
  // → password
  console.log(form.elements.password.type);
  // → password
  console.log(form.elements.name.form == form);
  // → true
</script>
```

---

Quando si preme un pulsante con un attributo `type="submit"`, il modulo viene inviato. Lo stesso effetto si ottiene premendo INVIO quando un campo del form ha il focus.

Inviando un modulo normalmente, il browser naviga fino alla pagina indicata dall'attributo `action` del modulo attraverso una richiesta GET o POST. Ma prima di quell'azione, scatta un evento "submit", che può essere gestito da JavaScript anche con gestori che impediscano il comportamento predefinito richiamando `preventDefault` sull'oggetto evento.

---

```
<form action="example/submit.html">
  Value: <input type="text" name="value">
  <button type="submit">Save</button>
</form>
<script>
  var form = document.querySelector("form");
  form.addEventListener("submit", function(event) {
    console.log("Saving value", form.elements.value.value);
    event.preventDefault();
  });
</script>
```

---

L'intercettazione di eventi "submit" ha diversi usi in JavaScript. Possiamo scrivere del codice che verifica che l'utente abbia inserito dei dati sensati e mostrare subito un messaggio di errore invece di inviare il modulo. Oppure, possiamo disattivare completamente il solito modo di invio del modulo, come nell'esempio precedente, e lasciare che sia il programma a gestire l'input, magari usando XMLHttpRequest per trasmetterlo a un server senza dover ricaricare la pagina.

## Campi di testo

I campi creati dai tag `<input>` con `type="text"` o `type="password"`, nonché i tag `<textarea>`, condividono la stessa interfaccia. I rispettivi elementi DOM hanno una proprietà `value` che mantiene il contenuto corrente come valore stringa. Impostare questa proprietà su un'altra stringa modifica il contenuto del campo.

Le proprietà `selectionStart` e `selectionEnd` dei campi di testo ci danno informazioni sul cursore e sulle selezioni nel testo. Quando nessuna parte del testo è selezionata, queste due proprietà mantengono lo stesso numero, che indica la posizione del cursore. Per esempio, 0 indica l'inizio del testo e 10 indica che il cursore si trova dopo il decimo carattere. Quando è selezionata una parte del testo, le due proprietà non coincidono e ci danno l'inizio e la fine della parte selezionata. Anche queste proprietà, come `value`, possono essere sovrascritte.

Come esempio, immaginate di dover scrivere un articolo su Khasekhemwy, ma di non essere sicuri dell'ortografia di quel nome. Il codice seguente imposta un tag `<textarea>` con un gestore di eventi che inserisce automaticamente la stringa "Khasekhemwy" quando premete il tasto F2.

---

```
<textarea></textarea>
<script>
  var textarea = document.querySelector("textarea");
  textarea.addEventListener("keydown", function(event) {
    // The key code for F2 happens to be 113
    if (event.keyCode == 113) {
      replaceSelection(textarea, "Khasekhemwy");
      event.preventDefault();
    }
  });
  function replaceSelection(field, word) {
    var from = field.selectionStart, to = field.selectionEnd;
    field.value = field.value.slice(0, from) + word +
      field.value.slice(to);
    // Put the cursor after the word
    field.selectionStart = field.selectionEnd =
      from + word.length;
  }
</script>
```

---

La funzione `replaceSelection` sostituisce la parte di testo correntemente selezionata con la parola data e quindi sposta il cursore alla fine, per permettervi di continuare a

scrivere.

L'evento "change" per un campo testo non scatta ogni volta che si digita qualcosa. Scatta invece quando, dopo averne modificato il contenuto, il campo si disattiva. Per rispondere immediatamente alle modifiche in un campo di testo, dovete registrare un gestore per l'evento "input", che scatti ogni volta che l'utente digita un carattere, cancella del testo o manipola in altro modo il contenuto del campo.

L'esempio seguente mostra un campo di testo e un contatore che riporta la lunghezza del testo digitato finora:

---

```
<input type="text"> length: <span id="length">0</span>
<script>
  var text = document.querySelector("input");
  var output = document.querySelector("#length");
  text.addEventListener("input", function() {
    output.textContent = text.value.length;
  });
</script>
```

---

## Caselle di spunta e pulsanti di opzione

Una casella di spunta (checkbox) è un semplice interruttore binario. Il suo valore può essere estratto o modificato attraverso la sua proprietà checked, che mantiene un valore booleano.

---

```
<input type="checkbox" id="purple">
<label for="purple">Make this page purple</label>
<script>
  var checkbox = document.querySelector("#purple");
  checkbox.addEventListener("change", function() {
    document.body.style.background =
      checkbox.checked ? "mediumpurple" : "";
  });
</script>
```

---

Il tag `<label>` serve per associare del testo a un campo di input. Il suo attributo `for` fa riferimento all'attributo `id` del campo. Facendo clic su `<label>` si attiva il campo, che cambia il valore (da selezionato a non selezionato o viceversa) quando si tratta di una casella di spunta o di un pulsante di opzione.

Il pulsante di opzione è simile alla casella di spunta, ma è implicitamente collegato ad

altri pulsanti dello stesso tipo e con lo stesso attributo name, in modo che possa essere attivo solo uno per volta.

---

Color:

```
<input type="radio" name="color" value="mediumpurple"> Purple
<input type="radio" name="color" value="lightgreen"> Green
<input type="radio" name="color" value="lightblue"> Blue
<script>
    var buttons = document.getElementsByName("color");
    function setColor(event) {
        document.body.style.background = event.target.value;
    }
    for (var i = 0; i < buttons.length; i++)
        buttons[i].addEventListener("change", setColor);
</script>
```

---

Il metodo `document.getElementsByName` recupera tutti gli elementi con l'attributo name dato. Nell'esempio, si passa in ciclo su tutti gli elementi (con un ciclo `for` normale, non `forEach`, perché la collezione restituita non è un vero array) e si registra un gestore per ciascuno di essi. Ricorderete che gli oggetti evento hanno una proprietà `target` che fa riferimento all'elemento che ha fatto scattare l'evento. Ciò può essere utile in gestori di eventi come questo, che può essere richiamato su diversi elementi e deve poter accedere all'elemento `target` corrente.

## Campi selezionabili

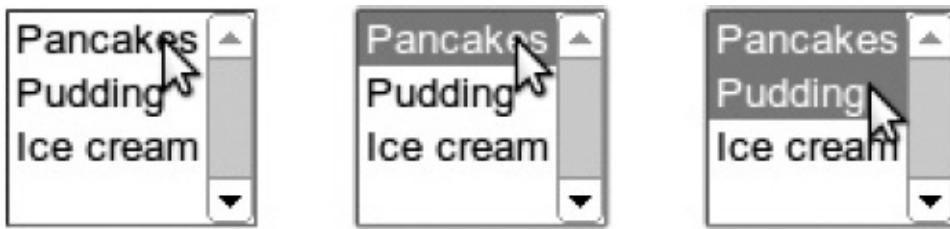
I campi selezionabili sono concettualmente simili ai pulsanti di opzione, in quanto consentono di scegliere un'opzione da una serie data. Mentre il pulsante di opzione lascia al programmatore la scelta del layout per le opzioni, l'aspetto del tag `<select>` dipende dal browser.

I campi selezionabili hanno una variante, più simile a un elenco di caselle di spunta che a pulsanti di opzione. Con l'attributo `multiple`, infatti, il tag `<select>` permette di scegliere qualunque numero di opzioni tra quelle date, invece di una sola.

---

```
<select multiple>
    <option>Pancakes</option>
    <option>Pudding</option>
    <option>Ice cream</option>
</select>
```

---



In genere, i browser visualizzano questo tipo di tag `<select>` in maniera diversa da quello tradizionale a scelta unica, che viene rappresentato come una *lista a cascata*, che mostra le opzioni disponibili solo quando lo aprirete.

L'attributo `size` del tag `<select>` permette di impostare il numero di opzioni visibili contemporaneamente, dandovi così il controllo esplicito sull'aspetto della lista. Per esempio, impostare l'attributo `size` su “3” mostra tre righe, non importa se l'opzione `multiple` sia impostata o meno.

Ciascun tag `<option>` ha un valore, che si può definire con un attributo `value`. Quando il valore non è definito, viene preso il testo contenuto nel tag come valore di scelta. La proprietà `value` di un elemento `<select>` riflette la scelta selezionata. Quando il campo consente più scelte, questa proprietà perde significato, in quanto riporterà solo una delle opzioni selezionate.

Ai tag `<option>` di un campo `<select>` si accede come a un oggetto simile a un array attraverso la proprietà `options` del campo. Ciascuna opzione ha una proprietà `selected`, che indica se quell'opzione è correntemente selezionata. La proprietà può essere sovrascritta per selezionare o deselectare un'opzione.

Nell'esempio che segue, estraiamo i valori selezionati da un campo selezionabile `multiple` e li usiamo per comporre un numero binario da singoli bit. Tenete premuto il tasto `CTRL` (o `COMMAND` sul Mac) per selezionare più opzioni.

---

```

<select multiple>
  <option value="1">0001</option>
  <option value="2">0010</option>
  <option value="4">0100</option>
  <option value="8">1000</option>
</select> = <span id="output">0</span>
<script>
  var select = document.querySelector("select");
  var output = document.querySelector("#output");
  select.addEventListener("change", function() {
    var number = 0;
    for (var i = 0; i < select.options.length; i++) {
      var option = select.options[i];
      if (option.selected)
        number += Number(option.value);
    }
    output.innerHTML = number;
  });
</script>

```

```
}

output.textContent = number;

});

</script>
```

---

## Campi file

I campi file furono pensati inizialmente come soluzione per caricare dei file dalla macchina del browser attraverso un modulo. Nei browser moderni, offrono anche la possibilità di leggere il contenuto dei file attraverso programmi JavaScript. Il campo funge un po' da sentinella, nel senso che lo script non può leggere direttamente i file che trova sul computer, ma se l'utente seleziona un file in quel campo, il browser interpreta quell'azione come autorizzazione per lo script a leggere il file.

I campi file sono di solito visualizzati come un pulsante, con un'etichetta che dice "scegli file", "sfoglia", o diciture simili, seguita dalle informazioni sul file.

---

```
<input type="file">

<script>

var input = document.querySelector("input");
input.addEventListener("change", function() {
  if (input.files.length > 0) {
    var file = input.files[0];
    console.log("You chose", file.name);
    if (file.type)
      console.log("It has type", file.type);
  }
});
</script>
```

---

La proprietà `files` di un campo di tipo file è un oggetto simile a un array (anch'esso non un vero array) che contiene i file scelti. All'inizio è vuoto. La ragione per cui non esiste una semplice proprietà `file` è che anche i campi file offrono un attributo `multiple`, che consente di scegliere più file contemporaneamente.

Gli oggetti della proprietà `files` hanno proprietà come `name` (il nome del file), `size` (le dimensioni in byte) e `type` (il tipo di media del file, come `text/ plain` o `image/jpeg`).

Non hanno però una proprietà per il contenuto del file, per arrivare al quale bisogna fare dell'altro lavoro. Poiché leggere un file dal disco può richiedere tempo, bisogna che l'interfaccia sia asincrona per evitare che il documento si blocchi. Potete pensare al

costruttore `FileReader` come a una cosa simile a `XMLHttpRequest`, ma specifica per i file.

---

```
<input type="file" multiple>
<script>
  var input = document.querySelector("input");
  input.addEventListener("change", function() {
    Array.prototype.forEach.call(input.files, function(file) {
      var reader = new FileReader();
      reader.addEventListener("load", function() {
        console.log("File", file.name, "starts with",
                    reader.result.slice(0, 20));
      });
      reader.readAsText(file);
    });
  });
</script>
```

---

Per leggere un file, si crea un oggetto `FileReader`, si registra un suo gestore di eventi "load" e si richiama il suo metodo `readAsText` passandogli il file che vogliamo leggere. Quando il file è stato caricato, la proprietà `result` contiene il contenuto del file.

Nell'esempio, usiamo `Array.prototype.forEach` per iterare sull'array, in quanto sarebbe troppo complicato recuperare i giusti oggetti `file` e `reader` dal gestore di eventi con un ciclo normale.

`FileReader` fa scattare un evento "error" quando la lettura del file non si conclude, non importa per quale motivo. L'oggetto `error` finisce nella proprietà `error` di `FileReader`. Se non volete dover ricordare i particolari di un'altra interfaccia asincrona, potete avvolgere la funzione in un oggetto `Promise` ([Capitolo 17](#)) come il seguente:

---

```
function readFile(file) {
  return new Promise(function(succeed, fail) {
    var reader = new FileReader();
    reader.addEventListener("load", function() {
      succeed(reader.result);
    });
    reader.addEventListener("error", function() {
      fail(reader.error);
    });
    reader.readAsText(file);
  });
}
```

---

Richiamando il metodo `slice` sul file e passando il risultato (detto oggetto `blob`) al lettore di file, potete leggere solo una parte del file.

## Registrare i dati sul lato client

Delle semplici pagine HTML con un po' di JavaScript possono essere un ottimo sistema per "mini applicazioni", piccoli programmi per automatizzare cose di uso quotidiano. Collegando campi di modulo con gestori di eventi, potete fare di tutto: dalle conversioni tra gradi Celsius e gradi Fahrenheit al calcolo delle password partendo da una master password e dal nome di un sito Web.

Quando applicazioni come queste devono tenere in memoria qualcosa tra una sessione e l'altra, non potete usare variabili JavaScript, che vengono eliminate appena si chiude la pagina. Potreste impostare un server, collegarlo a Internet e usarlo per tenere in memoria i dati delle vostre applicazioni: vedremo come farlo nel [Capitolo 20](#). Per questo, però, ci vuole un sacco di lavoro complicato. A volte basterebbe tenere i dati nel browser. Ma come si fa?

Potete memorizzare dati stringa in modo che sopravvivano alla pagina Web inserendoli in un oggetto `localStorage`. Quest'oggetto vi permette di archiviare valori stringa per nome (o per stringa), come nell'esempio seguente:

---

```
localStorage.setItem("username", "marijn");
console.log(localStorage.getItem("username"));
// → marijn
localStorage.removeItem("username");
```

---

Un valore in un oggetto `localStorage` rimane disponibile finché non viene sovrascritto, o eliminato con `removeItem`, oppure quando l'utente ripulisce i dati locali.

I siti in diversi domini ricevono aree di memorizzazione diverse. Ciò significa che i dati memorizzati in `localStorage` da un certo sito Web possono, in linea di massima, solo essere scritti (e sovrascritti) da script che si trovano sullo stesso sito.

Inoltre, i browser applicano un limite alle dimensioni dei dati memorizzabili in `localStorage`, che di solito non supera qualche megabyte. Queste restrizioni, insieme al fatto che non conviene mai riempire i dischi rigidi con robe inutili, impediscono a questa funzionalità di consumare troppo spazio.

Il codice che segue produce una piccola applicazione per prendere appunti. Mantiene gli appunti dell'utente come un oggetto, associando ai titoli le rispettive stringhe di contenuto. L'oggetto è codificato come JSON e memorizzato in `localStorage`. L'utente può selezionare una nota dal campo `<select>` e modificarne il testo nel rispettivo campo `<textarea>`. Per aggiungere un appunto, si fa clic su un pulsante.

---

Notes: <select id="list"></select>

```
<button onclick="addNote()">new</button><br>
<textarea id="currentnote" style="width: 100%; height: 10em">
</textarea>
<script>
  var list = document.querySelector("#list");
  function addToList(name) {
    var option = document.createElement("option");
    option.textContent = name;
    list.appendChild(option);
  }
  // Initialize the list from localStorage
  var notes = JSON.parse(localStorage.getItem("notes")) ||
    {"shopping list": ""};
  for (var name in notes)
    if (notes.hasOwnProperty(name))
      addToList(name);
  function saveToStorage() {
    localStorage.setItem("notes", JSON.stringify(notes));
  }
  var current = document.querySelector("#currentnote");
  current.value = notes[list.value];
  list.addEventListener("change", function() {
    current.value = notes[list.value];
  });
  current.addEventListener("change", function() {
    notes[list.value] = current.value;
    saveToStorage();
  });
  function addNote() {
    var name = prompt("Note name", "");
    if (!name) return;
    if (!notes.hasOwnProperty(name)) {
      notes[name] = "";
      addToList(name);
      saveToStorage();
    }
    list.value = name;
  }
</script>
```

```
    current.value = notes[name];  
}  
</script>
```

---

Lo script inizializza la variabile `notes` sul valore memorizzato in `localStorage` oppure, se quel valore manca, su un semplice oggetto contenente solo una nota vuota dal titolo "shopping list" (lista della spesa). Leggere un campo che non esiste in `localStorage` restituisce un valore di `null`. Passando `null` a `JSON.parse`, l'interprete analizza la stringa "`null`" e restituisce `null`. In queste situazioni, si preferisce pertanto passare un valore di default con l'operatore `||`.

Quando cambiano i dati degli appunti, perché se ne aggiunge uno nuovo o se ne modifica uno esistente, si richiama la funzione `saveLocalStorage` per aggiornare il campo. Se l'applicazione dovesse gestire migliaia di appunti, invece di una mezza dozzina, questo procedimento sarebbe troppo dispendioso e dovremmo trovare un modo migliore per memorizzare gli appunti; per esempio, assegnando un campo di memorizzazione per ciascuno di essi.

Quando l'utente aggiunge un nuovo appunto, il codice deve aggiornare esplicitamente il campo `testo`, anche se il campo `<select>` dispone di un gestore "change" che svolge lo stesso lavoro: tuttavia, gli eventi "change" scattano solo quando l'utente modifica il valore del campo, non quando a far ciò è uno script.

Esiste un altro oggetto simile a `localStorage`, che si chiama `sessionStorage`. La differenza tra i due è che il contenuto di `sessionStorage` viene eliminato alla fine di ogni sessione, che in quasi tutti i browser coincide con la chiusura del browser.

## Riepilogo

In HTML si possono definire diversi tipi di campi per i moduli, tra cui campi di testo, caselle di spunta, campi per scelte multiple e per scegliere file.

Questi campi si possono ispezionare e manipolare con JavaScript. Fanno scattare eventi "change" in caso di modifica, eventi "input" quando ci si scrive dentro e vari eventi da tastiera. Questi eventi ci permettono di rilevare quando l'utente sta interagendo con i campi. Per leggere o impostare il contenuto dei campi, si usano proprietà come `value` (per i campi di testo e selezionabili) o `checked` (per le caselle di scelta e i pulsanti di opzione).

Quando si preme un pulsante definito con `<button type="submit">`, scatta un evento "submit", sul quale gli script JavaScript possono richiamare `preventDefault` per impedire che il modulo venga inviato. Gli elementi per i campi del modulo non devono per forza essere racchiusi tra tag `<form>`.

Quando l'utente seleziona un file dal proprio sistema in un campo file, si può usare l'interfaccia `FileReader` per leggere il contenuto del file da un programma JavaScript.

Gli oggetti `localStorage` e `sessionStorage` si possono usare per salvare dei dati in

modo che non spariscano quando si ricarica la pagina. Il primo salva i dati per sempre (o finché non li elimina l’utente), il secondo li salva solo finché rimane aperto il browser.

## Esercizi

### ***Un laboratorio per JavaScript***

Realizzate un’interfaccia che permetta di scrivere ed eseguire brani di JavaScript.

Mettete un pulsante vicino a un campo <textarea>, che quando viene premuto usi il costruttore Function descritto nel [Capitolo 10](#) per avvolgere il testo in una funzione e richiamarla. Convertite in stringa il valore restituito dalla funzione, o eventuali errori essa sollevasse, e visualizzate il risultato dopo il campo testo.

### ***Autocompletamento***

Estendete un campo testo in modo che, quando l’utente vi scrive dentro, sotto di esso compaia un elenco di valori suggeriti. Avete a disposizione un array di valori possibili e dovrete mostrare quelli che iniziano con i caratteri digitati. Quando l’utente fa clic su un suggerimento, sostituite con questo il contenuto corrente del campo testo.

### ***Il Gioco della vita di Conway***

Il Gioco della vita di Conway è una semplice simulazione che crea “vita” artificiale su una griglia, dove alcune celle sono vive e le altre non lo sono. Per ogni generazione (turno di gioco), si applicano le seguenti regole:

- Le celle vive, che hanno meno di due o più di tre vicini vivi, muoiono.
- Le celle vive, che hanno due o tre vicini vivi, sopravvivono fino alla prossima generazione.
- Le celle morte, che hanno esattamente tre vicini vivi, diventano vive.

Il vicino (neighbor nel codice) è definito come una qualunque delle celle adiacenti, comprese quelle diagonali.

Notate che queste regole si applicano contemporaneamente a tutta la griglia e non a una cella per volta. Ciò significa che il conteggio dei vicini si basa sulla situazione all’inizio della generazione e i cambiamenti che avvengono nelle celle vicine durante questa generazione non devono influenzare il nuovo stato della cella data.

Realizzate il gioco con la struttura dati che ritenete più appropriata. Usate `Math.random` per popolare la griglia a caso all’inizio del gioco. Visualizzate il gioco come una griglia di caselle di spunta, con un solo pulsante per passare alla generazione successiva. Quando l’utente seleziona o deselectiona le caselle, i loro cambiamenti vanno considerati nel calcolo della generazione successiva.

## PROGETTO: UN PROGRAMMA PER DIPINGERE

*Guardo tutti i colori che ho davanti a me. Guardo la tela vuota. Poi, cerco di applicare i colori come parole che plasmano poemi, come note che plasmano musica.*

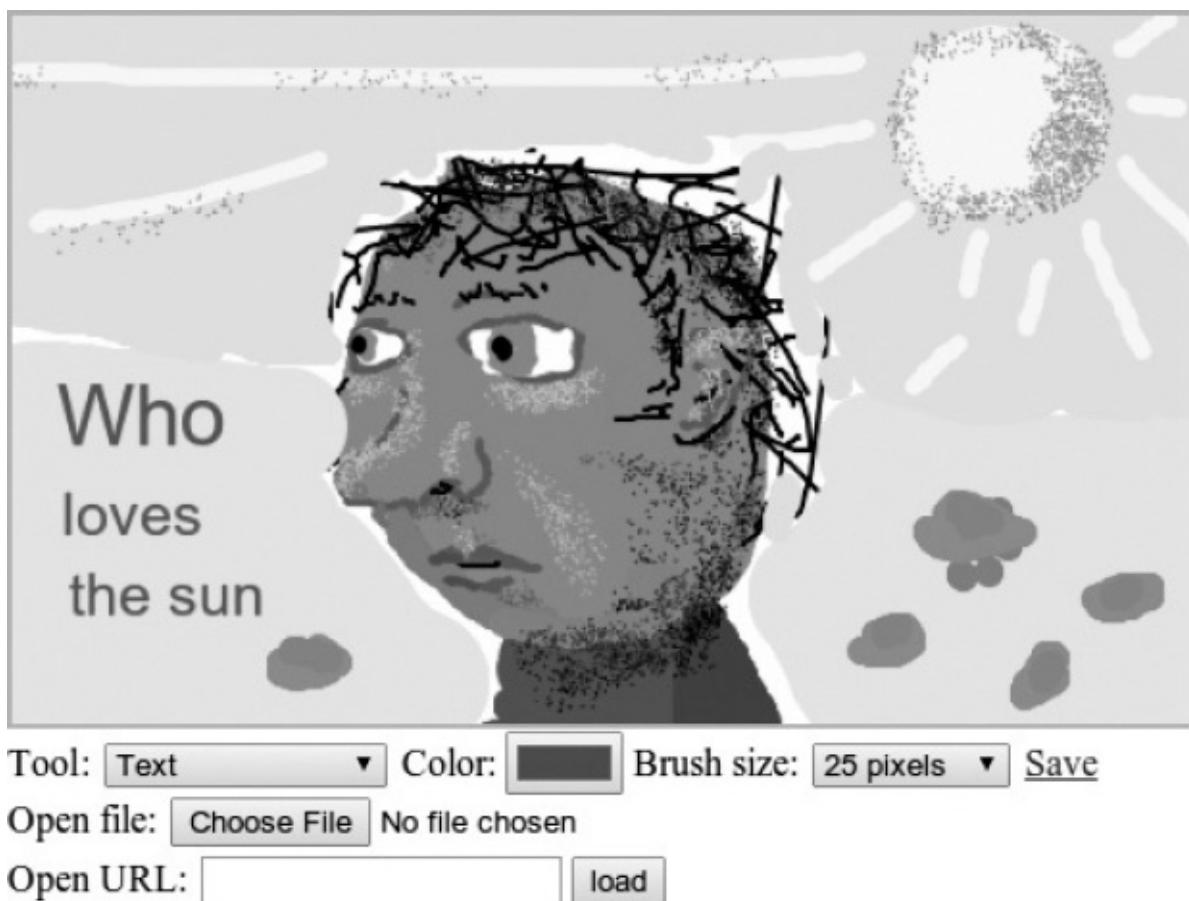
Joan Miró

Il materiale dei capitoli precedenti vi dà tutti gli elementi che servono per costruire una semplice applicazione Web. Cosa che faremo in questo capitolo.

La nostra applicazione sarà un programma di disegno simile a Microsoft Paint, ma basato sul Web. Potrete usarlo per aprire file di immagini, disegnare e scribacchiarci sopra col mouse e salvarli.

Dipingere su un computer è fantastico. Non dovete preoccuparvi dei materiali, né di abilità o talento. Dovete solo cominciare a pasticciare con i colori.

Ecco un esempio:



## Realizzazione

L'interfaccia per questo programma mostra un elemento <canvas> di grandi dimensioni nella parte alta dello schermo, con una serie di campi di modulo nella parte inferiore. L'utente disegna sull'immagine selezionando uno strumento da un campo <select> e quindi facendo clic e trascinando il mouse sull'area di disegno. Ci sono strumenti per disegnare linee, cancellare parti dell'immagine, aggiungere del testo e così via.

Fare clic sulla tela virtuale fa scattare l'evento "mousedown" per lo strumento selezionato, che può gestirlo come vuole. Per esempio, lo strumento per tracciare righe rimane in attesa di eventi "mousemove" finché il pulsante viene rilasciato e traccia linee lungo il percorso del mouse col colore e le dimensioni del pennello selezionati.

Colore e dimensioni del pennello si selezionano con altri campi del modulo, che sono impostati per aggiornare, su richiesta dell'utente, i metodi `fillStyle`, `strokeStyle` e `LineWidth` del contesto di disegno.

Ci sono due modi per caricare un'immagine nel programma. Nel primo, usate un campo file per selezionare un'immagine dal sistema di file locale. Il secondo chiede un URL per caricare un'immagine dal Web.

Le immagini sono salvate in modo abbastanza atipico. Il link *Save*, sulla destra del programma, punta all'immagine corrente e può essere seguito, condiviso o salvato. Spiegherò tra poco come funziona.

## Costruire la struttura DOM

L'interfaccia del programma richiede più di 30 elementi DOM, che dobbiamo costruire in qualche modo.

Il formato HTML è quello che si presta meglio a definire strutture DOM complesse. Separare il programma in due, però, da una parte il codice HTML e dall'altra lo script, è complicato dal fatto che molti degli elementi DOM devono avere gestori di eventi o devono rispondere in qualche altro modo allo script. Pertanto, lo script avrebbe bisogno di un sacco di chiamate a `querySelector` (o simili) per trovare gli elementi del DOM sui quali intervenire.

Sarebbe meglio se la struttura DOM per le varie parti dell'interfaccia fosse definita in modo più vicino al codice JavaScript che fa girare il programma. Pertanto, ho scelto di creare in JavaScript tutti gli elementi come nodi DOM. Come abbiamo visto nel [Capitolo 13](#), l'interfaccia predefinita per realizzare una struttura DOM è tremendamente verbosa. Per realizzare la costruzione della struttura DOM, ci serve pertanto una funzione ausiliaria.

Questa funzione ausiliaria è una versione estesa della funzione `elt` del [Capitolo 13](#), che crea un elemento con il nome e gli attributi dati e lo collega a tutti gli altri argomenti che riceve come nodi figli, convertendo automaticamente delle stringhe in nodi di testo.

---

```
function elt(name, attributes) {
    var node = document.createElement(name);
    if (attributes) {
        for (var attr in attributes)
            if (attributes.hasOwnProperty(attr))
                node.setAttribute(attr, attributes[attr]);
    }
    for (var i = 2; i < arguments.length; i++) {
        var child = arguments[i];
        if (typeof child == "string")
            child = document.createTextNode(child);
        node.appendChild(child);
    }
    return node;
}
```

---

Ciò semplifica la creazione degli elementi, senza che il codice debba essere troppo lungo e complicato.

## Le fondamenta

Il nucleo del nostro programma è la funzione `createPaint`, che collega l'interfaccia di disegno all'elemento DOM che riceve come argomento. Poiché vogliamo costruire il programma un pezzo per volta, definiamo un oggetto, `controls`, che manterrà le funzioni di inizializzazione per i controlli sotto l'immagine.

---

```
var controls = Object.create(null);
function createPaint(parent) {
    var canvas = elt("canvas", {width: 500, height: 300});
    var cx = canvas.getContext("2d");
    var toolbar = elt("div", {class: "toolbar"});
    for (var name in controls)
        toolbar.appendChild(controls[name](cx));
    var panel = elt("div", {class: "picturepanel"}, canvas);
    parent.appendChild(elt("div", null, panel, toolbar));
}
```

---

Ciascun controllo ha accesso al contesto di disegno e, attraverso la proprietà `canvas` del contesto, all'elemento `<canvas>`. Gran parte del programma vive proprio qui: la tela

contiene l'immagine corrente, il colore (nella proprietà `fillStyle`) e le dimensioni del pennello (nella proprietà `linewidth`) selezionati.

Racchiudiamo l'area di disegno e i controlli in elementi `<div>` con attributi `class` per aggiungere degli stili, come per esempio il bordo grigio intorno all'immagine.

## Selezione degli strumenti

Il primo controllo che aggiungiamo è l'elemento `<select>` che consente di scegliere uno strumento per disegnare. Come con i controlli, useremo un oggetto per raccogliere gli strumenti a disposizione, per non doverli codificare individualmente in un solo posto e poter aggiungere senza difficoltà nuovi strumenti in un secondo tempo. Quest'oggetto associa i nomi degli strumenti con la funzione da richiamare quando vengono selezionati e l'utente fa clic sulla tela virtuale.

---

```
var tools = Object.create(null);
controls.tool = function(cx) {
    var select = elt("select");
    for (var name in tools)
        select.appendChild(elt("option", null, name));
    cx.canvas.addEventListener("mousedown", function(event) {
        if (event.which == 1) {
            tools[select.value](event, cx);
            event.preventDefault();
        }
    });
    return elt("span", null, "Tool: ", select);
};
```

Il campo per gli strumenti è popolato da elementi `<option>` per ciascuno degli strumenti definiti, mentre un gestore di eventi "mousedown" sulla tela ha il compito di richiamare la funzione per lo strumento corrente passando come argomenti sia l'oggetto evento, sia il contesto di disegno. Il gestore richama inoltre `preventDefault`, in modo che, tenendo premuto il pulsante del mouse mentre lo si trascina, impedisca il comportamento normale di selezionare parte della pagina.

Lo strumento più semplice è quello che permette di tracciare linee col mouse. Per posizionare correttamente le estremità della linea, dobbiamo trovare sull'area di disegno le coordinate relative a un dato evento del mouse. Qui viene utile il metodo `getBoundingClientRect`, al quale avevo accennato nel [Capitolo 13](#), che ci dà la posizione di visualizzazione di un elemento relativa all'angolo superiore sinistro dello schermo. Poiché anche le proprietà `clientX` e `clientY` degli eventi del mouse sono relative allo stesso angolo, basta sottrarre da queste l'angolo superiore sinistro della tela per trovare la

posizione relativa a quell'angolo.

---

```
function relativePos(event, element) {  
    var rect = element.getBoundingClientRect();  
    return {x: Math.floor(event.clientX - rect.left),  
            y: Math.floor(event.clientY - rect.top)};  
}
```

---

Altri strumenti di disegno devono rimanere in attesa di eventi "mousemove" fintanto che il pulsante del mouse rimane premuto. La funzione trackDrag si occupa di registrare e deregistrare gli eventi in queste situazioni.

---

```
function trackDrag(onMove, onEnd) {  
    function end(event) {  
        removeEventListener("mousemove", onMouse);  
        removeEventListener("mouseup", end);  
        if (onEnd)  
            onEnd(event);  
    }  
    addEventListener("mousemove", onMouse);  
    addEventListener("mouseup", end);  
}
```

---

Questa funzione accetta due argomenti: uno è una funzione da richiamare per ciascun evento "mousemove", l'altra è per quando viene rilasciato il pulsante del mouse. Uno o l'altro argomento possono essere omessi se non sono necessari.

Lo strumento linea (line) usa due funzioni ausiliarie per effettuare il disegno:

---

```
tools.Line = function(event, cx, onEnd) {  
    cx.lineCap = "round";  
    var pos = relativePos(event, cx.canvas);  
    trackDrag(function(event) {  
        cx.beginPath();  
        cx.moveTo(pos.x, pos.y);  
        pos = relativePos(event, cx.canvas);  
        cx.lineTo(pos.x, pos.y);  
        cx.stroke();  
    }, onEnd);  
};
```

---

La funzione comincia con l'impostare la proprietà `lineCap` del contesto su "round", che arrotonda le estremità di un tracciato invece di lasciarle squadrate. Questo è un trucco per fare in modo che più linee, tracciate in risposta a diversi eventi, sembrino un'unica linea ininterrotta. Aumentando lo spessore della linea, se usate le estremità squadrate di default, vedrete degli spazi vuoti negli angoli.

Quindi, per ogni evento "mousemove" che scatta finché il pulsante del mouse è premuto, si traccia un semplice segmento di retta tra la posizione del mouse precedente e quella corrente, usando lo stile `strokeStyle` e lo spessore `lineWidth` che risultano impostati.

L'argomento `onEnd` viene passato a `tools.Line` attraverso `trackDrag`. Normalmente, non si passa un terzo argomento agli strumenti; quando si usa lo strumento linea, pertanto, il terzo argomento sarà `undefined` e non succede nulla alla fine del trascinamento del mouse. Il terzo argomento `undefined` va previsto per permetterci di aggiungere uno strumento gomma (`erase`) con poco codice in più.

---

```
tools.Erase = function(event, cx) {
  cx.globalCompositeOperation = "destination-out";
  tools.Line(event, cx, function() {
    cx.globalCompositeOperation = "source-over";
  });
};
```

---

La proprietà `globalCompositeOperation` influenza il modo in cui le operazioni sulla tela cambiano il colore dei pixel che toccano. Nell'impostazione predefinita, il valore della proprietà è "source-over", che significa che il colore di disegno viene aggiunto al colore esistente. Se si tratta di un colore opaco, sostituisce il colore precedente, ma se è in parte trasparente i due colori si mischiano.

Lo strumento gomma imposta `globalCompositeOperation` su "destinationout", che ha l'effetto di cancellare i pixel toccati, rendendoli nuovamente trasparenti.

Con questo, abbiamo due strumenti nel programma. Possiamo tracciare linee larghe un solo pixel (come valore predefinito per `strokeStyle` e `lineWidth`) e cancellarle. Il programma funziona, anche se è ancora molto limitato.

## Colore e dimensioni del pennello

Poiché gli utenti vorranno disegnare a colori, oltre che in nero, e scegliere spessori diversi per il pennello, aggiungiamo i controlli per queste due impostazioni.

Nel [Capitolo 18](#), ho parlato di diversi campi per i moduli, tra cui i campi per i colori. Tradizionalmente, i browser non offrono supporto per le tavolozze di scelta dei colori, ma negli ultimi anni, diversi nuovi tipi di campo sono diventati standard. Uno di questi è `<input type="color">`. Altri comprendono "date", "email", "url" e "number". Non

tutti i browser riconoscono questi tag e, nel momento in cui scriviamo, nessuna versione di Internet Explorer dispone di campi colore. Poiché il tipo predefinito per i tag <input> è "text", quando il tipo specificato non è disponibile, i browser lo trattano come testo. Ciò significa che gli utenti di Internet Explorer che vogliono usare il nostro programma dovranno digitare il nome del colore che vogliono, invece di selezionarlo da un comodo widget.

Ecco come può essere visualizzata una tavolozza di scelta dei colori:



---

```
controls.color = function(cx) {  
  var input = elt("input", {type: "color"});  
  input.addEventListener("change", function() {  
    cx.fillStyle = input.value;  
    cx.strokeStyle = input.value;  
  });  
  return elt("span", null, "Color: ", input);  
};
```

---

Quando cambia il valore del campo colore, `fillStyle` e `strokeStyle` vengono aggiornati per mantenere il nuovo valore.

Il campo per configurare le dimensioni del pennello funziona in modo simile.

---

```
controls.brushSize = function(cx) {  
  var select = elt("select");  
  var sizes = [1, 2, 3, 5, 8, 12, 25, 35, 50, 75, 100];  
  sizes.forEach(function(size) {  
    select.appendChild(elt("option", {value: size},
```

```

        size + " pixels"));
});

select.addEventListener("change", function() {
    cx.lineWidth = select.value;
});

return elt("span", null, "Brush size: ", select);
};

```

---

Il codice produce le opzioni di scelta da un array di dimensioni per il pennello e verifica che venga aggiornata la proprietà `lineWidth` quando si sceglie lo spessore del pennello.

## Salvare

Per spiegare come funziona il link *Save*, devo prima parlare di *URL di dati*. Un URL di dati porta a dei dati e ha *data:* come protocollo. Diversamente dagli URL che iniziano con *http:* e *https:*, gli URL di dati puntano a una risorsa, che contengono completamente. Ecco un esempio di URL di dati che contiene un semplice documento HTML:

---

```
data:text/html,<h1 style="color:red">Hello!</h1>
```

---

Gli URL di dati si usano per diverse operazioni, tra cui includere piccole immagini direttamente in un file con un foglio stile. Permettono inoltre di collegarci ai file che abbiamo creato sul lato client, nel browser, senza doverli prima spostare su qualche server.

I fogli hanno un comodo metodo `toDataURL`, che restituisce un URL di dati che racchiude la figura sul foglio come file immagine. Non vogliamo, però, salvare l'immagine ogni volta che la cambiamo: specialmente quando è di grandi dimensioni, ciò richiederebbe spostare un sacco di dati nel link e rallenterebbe il programma. Invece, impostiamo il link *Save* in modo che salvi il suo attributo `href` quando è attivato dalla tastiera o quando il mouse si sposta sopra di esso.

---

```

controls.save = function(cx) {
    var link = elt("a", {href: "/"}, "Save");
    function update() {
        try {
            link.href = cx.canvas.toDataURL();
        } catch (e) {
            if (e instanceof SecurityError)
                link.href = "javascript:alert(" +
                    JSON.stringify("Can't save: " + e.toString()) + ")";
        }
    }
    update();
    link.addEventListener("click", function() {
        cx.canvas.toDataURL();
    });
    link.addEventListener("mouseover", update);
    link.addEventListener("mouseout", update);
};

```

```
        throw e;
    }
}

link.addEventListener("mouseover", update);
link.addEventListener("focus", update);
return link;
};
```

---

In questo modo, il link rimane inattivo e punta all'oggetto sbagliato, ma quando l'utente vi si avvicina, si aggiorna magicamente e punta all'immagine corrente.

Caricando un'immagine molto grande, alcuni browser si bloccano sull'enorme URL di dati che ciò produce. Per le immagini più piccole, questa soluzione funziona senza problemi.

Qui, però, torniamo ancora una volta alle complicazioni della sandbox del browser. Quando si carica un'immagine da un URL su un server remoto, se la risposta del server non riporta un header con l'indicazione che la risorsa può essere utilizzata su altri domini ([Capitolo 17](#)), l'immagine può essere visualizzata dall'utente, ma non utilizzata dallo script.

Questo perché potremmo aver richiesto un'immagine che contiene informazioni riservate (per esempio, una rappresentazione grafica del saldo in banca dell'utente) accessibili solo per la sessione dell'utente. Se gli script avessero accesso alle informazioni ricavabili da quell'immagine, la riservatezza dei dati dell'utente sarebbe a rischio.

Per evitare che trapelino informazioni riservate, i browser *macchiano* la tela (*tainted canvas*) quando su di essa compaiono immagini a cui il browser non ha il permesso di accedere. In queste condizioni, non si possono estrarre dati sui pixel, né tantomeno URL di dati. Sulla tela potete sempre scrivere, ma non potete più leggere quel che vi si trova.

Ecco perché nella funzione di aggiornamento per il link *Save* abbiamo bisogno di un blocco *try/catch*. Quando l'area di disegno si offusca, richiamare *toDataURL* solleva un'eccezione che è un'istanza di *SecurityError*. A quel punto, impostiamo il link in modo che punti a un altro tipo di URL e usiamo il protocollo *javascript*: per eseguire lo script specificato dopo il segno “:”. Facendo clic sul link, si aprirà una finestra, con un messaggio che avverte l'utente del problema.

## Caricare file di immagini

Gli ultimi due controlli sono quelli per caricare le immagini da file locali e da URL. Avremo bisogno della funzione ausiliaria seguente, che richiede un'immagine dall'URL dato e la sostituisce al contenuto dell'area di disegno:

```
function loadImageURL(cx, url) {
    var image = document.createElement("img");
```

```

image.addEventListener("load", function() {
    var color = cx.fillStyle, size = cx.lineWidth;
    cx.canvas.width = image.width;
    cx.canvas.height = image.height;
    cx.drawImage(image, 0, 0);
    cx.fillStyle = color;
    cx.strokeStyle = color;
    cx.lineWidth = size;
});
image.src = url;
}

```

---

Vogliamo modificare le dimensioni della tela virtuale in modo che si adattino perfettamente all'immagine. Per qualche strana ragione, cambiare le dimensioni dell'area fa dimenticare al suo contesto di disegno alcune proprietà di configurazione, come `fillStyle` e `lineWidth`; la funzione deve pertanto salvarle e memorizzarle dopo aver aggiornato le dimensioni dell'elemento.

Il controllo per caricare un file locale usa la tecnica `FileReader` del [Capitolo 18](#). Oltre al metodo `readAsText`, che avevamo visto in quel capitolo, oggetti di questo tipo hanno un metodo `readAsDataURL`, che è esattamente quel che vogliamo. Carichiamo il file scelto dall'utente come URL di dati e lo passiamo a `loadImageURL` per inserirlo nell'area di disegno:

```

controls.openFile = function(cx) {
    var input = elt("input", {type: "file"});
    input.addEventListener("change", function() {
        if (input.files.length == 0) return;
        var reader = new FileReader();
        reader.addEventListener("load", function() {
            loadImageURL(cx, reader.result);
        });
        reader.readAsDataURL(input.files[0]);
    });
    return elt("div", null, "Open file: ", input);
};

```

---

Caricare un file da un URL è ancora più semplice. Con un campo testo, poiché non sappiamo con certezza quando l'utente ha finito di digitare l'URL, non possiamo limitarci a rimanere in attesa di eventi "change". Dobbiamo pertanto racchiudere il campo in un modulo e rispondere quando viene inviato il modulo, o perché l'utente ha premuto ENTER,

o perché ha fatto clic sul pulsante load.

---

```
controls.openURL = function(cx) {
    var input = elt("input", {type: "text"});
    var form = elt("form", null,
        "Open URL: ", input,
        elt("button", {type: "submit"}, "load"));
    form.addEventListener("submit", function(event) {
        event.preventDefault();
        loadImageURL(cx, input.value);
    });
    return form;
};
```

---

Abbiamo così definito tutti i controlli necessari per il nostro semplice programma, al quale tuttavia non farebbe male qualche strumento in più.

## Rifiniture

Possiamo intanto aggiungere uno strumento per il testo, che usa la seguente funzione per chiedere all'utente che cosa vuole scrivere:

---

```
tools.Text = function(event, cx) {
    var text = prompt("Text:", "");
    if (text) {
        var pos = relativePos(event, cx.canvas);
        cx.font = Math.max(7, cx.lineWidth) + "px sans-serif";
        cx.fillText(text, pos.x, pos.y);
    }
};
```

---

Potreste aggiungere altri campi per tipo e dimensioni del carattere, ma per mantenere le cose semplici ci limitiamo a un tipo di carattere sans-serif delle dimensioni correnti del pennello. Le dimensioni minime sono 7 pixel, perché corpi più piccoli risultano illeggibili.

Un altro strumento indispensabile per disegnare (da grafici dilettanti) è lo strumento spray (aerografo). Questo spruzza punti in posizioni casuali sotto il pennello fintanto che il pulsante del mouse rimane premuto, coprendo l'area più o meno densamente, a seconda della velocità del mouse.

---

```
tools.Spray = function(event, cx) {
    var radius = cx.lineWidth / 2;
```

```

var area = radius * radius * Math.PI;
var dotsPerTick = Math.ceil(area / 30);
var currentPos = relativePos(event, cx.canvas);
var spray = setInterval(function() {
  for (var i = 0; i < dotsPerTick; i++) {
    var offset = randomPointInRadius(radius);
    cx.fillRect(currentPos.x + offset.x,
                currentPos.y + offset.y, 1, 1);
  }
}, 25);
trackDrag(function(event) {
  currentPos = relativePos(event, cx.canvas);
}, function() {
  clearInterval(spray);
});
};

```

---

Lo strumento aerografo usa `setInterval` per generare puntini colorati ogni 25 millisecondi, fintanto che il pulsante del mouse rimane premuto. Attraverso la funzione `trackDrag`, mantiene `currentPos` nella posizione corrente del mouse e interrompe l'intervallo quando viene rilasciato il pulsante del mouse.

Per stabilire quanti puntini disegnare ogni volta che scatta l'intervallo, la funzione calcola la superficie del pennello corrente e la divide per 30. Per trovare una posizione a caso sotto il pennello, si usa la funzione `randomPointInRadius`.

---

```

function randomPointInRadius(radius) {
  for (;;) {
    var x = Math.random() * 2 - 1;
    var y = Math.random() * 2 - 1;
    if (x * x + y * y <= 1)
      return {x: x * radius, y: y * radius};
  }
}

```

---

La funzione genera dei punti nel riquadro compreso tra (-1,-1) e (1,1). Usando il teorema di Pitagora, verifica se il punto generato giace all'interno di un cerchio di raggio 1. Appena lo trova, la funzione restituisce il punto moltiplicato per l'argomento `radius` (raggio).

Il ciclo è necessario per distribuire uniformemente i punti. Il sistema più semplice per generare punti casuali all'interno di un cerchio è di usare un angolo e una distanza casuali

e richiamare `Math.sin` e `Math.cos` per trovare il punto corrispondente. Con questo sistema, però, è più probabile che i punti si trovino tutti vicino al centro del cerchio. Ci sono altre soluzioni per quel problema, ma sono tutte più complicate del ciclo.

Abbiamo ora un programma di disegno che funziona.

## Esercizi

Il programma ha ancora un sacco di spazio per miglioramenti. Aggiungiamo altre funzionalità sotto forma di esercizi.

### Rettangoli

Definite uno strumento, `Rectangle`, che riempie un rettangolo (col metodo `fillRect` del Capitolo 16) col colore corrente. Il rettangolo deve estendersi dal punto dove l'utente ha premuto il tasto del mouse al punto dove l'ha rilasciato. Notate che quest'ultimo può trovarsi sopra o a sinistra del primo.

Quando funziona, noterete che dà un po' fastidio non vedere il rettangolo che state tracciando mentre trascinate il mouse per definirne le dimensioni. Riuscite a trovare la maniera di visualizzare un rettangolo durante il trascinamento, ma senza lasciare un disegno sulla tela finché non avete lasciato andare il pulsante del mouse?

Se non riuscite a trovare un'idea, tornate alla discussione sull'attributo di stile `position:absolute` del Capitolo 13, dove dicevo che lo si può usare per aggiungere un nodo al di sopra del documento. Potete usare le proprietà `pageX` e `pageY` di un evento mouse per posizionare con precisione un elemento sotto il mouse, impostando gli stili per `left`, `top`, `width` e `height` sui valori corretti in pixel.

### Tavolozza per i colori

Un altro strumento comune nei programmi di grafica è la tavolozza per scegliere i colori (color picker), dove l'utente fa clic in un'immagine per scegliere il colore desiderato col puntatore del mouse. Realizzatela.

Per questo strumento, abbiamo bisogno di accedere ai contenuti dell'area di disegno. Il metodo `toDataURL` lo faceva (più o meno), ma ricavare con quel metodo informazioni sui pixel di un URL di dati è complicato. Al suo posto, usiamo il metodo `getImageData` sul contesto del disegno, che restituisce una porzione rettangolare dell'immagine come oggetto con proprietà `width`, `height` e `data`. La proprietà `data` mantiene un array di quattro numeri da 0 a 255, che rappresentano i valori dei componenti rosso, verde, blu e alfa (opacità) per ciascun pixel.

L'esempio che segue recupera i valori per un pixel dell'area di disegno, prima quando questa è vuota (tutti i pixel sono nero trasparente) e poi quando il pixel è stato colorato di rosso.

---

```
function pixelAt(cx, x, y) {
```

```
var data = cx.getImageData(x, y, 1, 1);
console.log(data.data);
}

var canvas = document.createElement("canvas");
var cx = canvas.getContext("2d");
pixelAt(cx, 10, 10);
// → [0, 0, 0, 0]
cx.fillStyle = "red";
cx.fillRect(10, 10, 1, 1);
pixelAt(cx, 10, 10);
// → [255, 0, 0, 255]
```

---

Gli argomenti di `getImageData` indicano le coordinate x e y di partenza del rettangolo che vogliamo recuperare, seguite da larghezza e altezza.

Per quest'esercizio, potete ignorare la trasparenza ed esaminare solo i primi tre valori per il pixel dato. Non preoccupatevi di aggiornare il campo `color` quando l'utente sceglie un colore: è sufficiente che le proprietà `fillStyle` e `strokeStyle` del contesto di disegno siano impostate sul colore indicato dal cursore del mouse.

Ricordate che queste proprietà accettano qualunque colore valido in CSS, compresi quelli in notazione `rgb(R, G, B)` del [Capitolo 15](#).

Il metodo `getImageData` è soggetto alle stesse restrizioni di `toDataURL` e solleverà un'eccezione quando l'area di disegno contiene pixel che si trovano in altri domini. Usate un blocco `try/catch` per riportare questi errori su una finestra di dialogo.

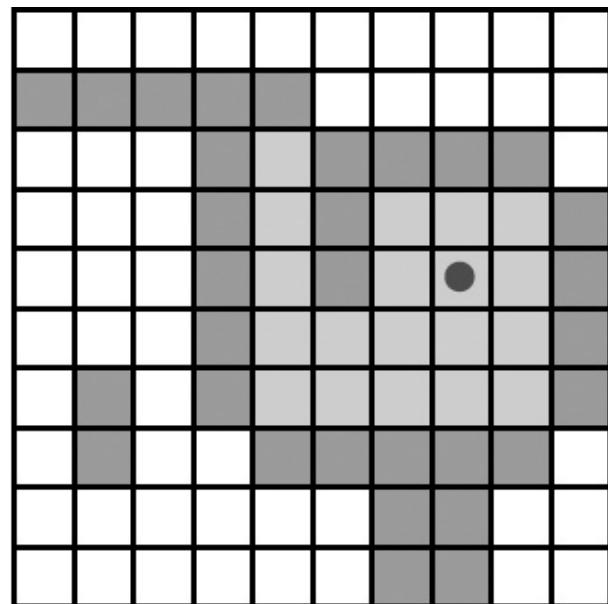
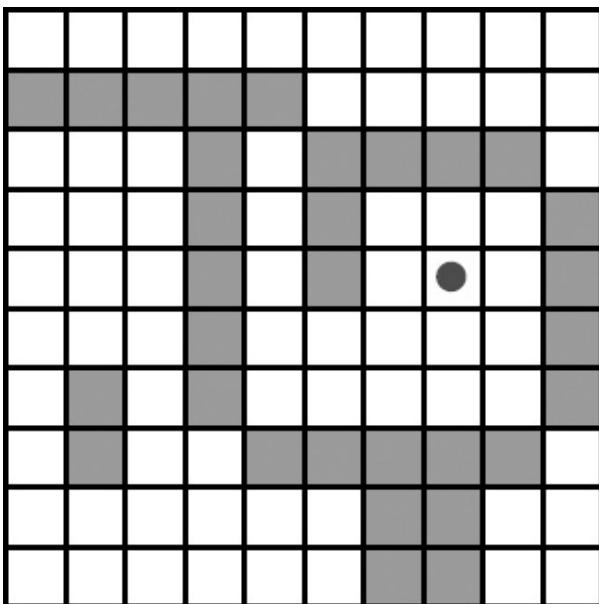
## Secchiello e riempimento

Questo è un esercizio più difficile dei due precedenti e richiede che progettiate una soluzione non triviale a un problema complesso. Scegliete un giorno di calma, prendetevi tutto il tempo che ci vuole per lavorarci sopra e non lasciatevi scoraggiare da eventuali fallimenti.

Lo strumento secchiello (flood fill) colora il pixel sotto il mouse e quelli che lo circondano e sono dello stesso colore. Ai fini di quest'esercizio, prenderemo in considerazione il gruppo di pixel che comprende quelli raggiungibili dal pixel iniziale spostandoci in progressione orizzontale e verticale (non diagonale), senza incontrare un pixel di colore diverso.

L'immagine alla pagina seguente illustra l'insieme di pixel che vengono colorati quando si usa lo strumento secchiello sul pixel contrassegnato.

Lo strumento non si estende diagonalmente e non tocca i pixel che non sono raggiungibili, nemmeno se sono dello stesso colore del pixel indicato.



Avrete bisogno anche qui di `getImageData` per trovare il colore di ciascun pixel. Può essere una buona idea recuperare tutta l'immagine in blocco e andare a cercare i dati sui singoli pixel nell'array risultante. In questo array, i pixel sono organizzati in modo simile agli elementi della griglia del [Capitolo 7](#), una riga per volta, con la differenza che ogni pixel è rappresentato da quattro valori. Il primo valore per il pixel in posizione  $(x,y)$  si trova nella posizione (di array)  $(x + y \times \text{larghezza}) \times 4$ .

Questa volta dovete calcolare anche il quarto valore (alfa), perché dobbiamo distinguere tra pixel vuoti e pixel neri.

Per trovare tutti i pixel adiacenti con lo stesso colore, dovete “camminare” sulla superficie, spostandovi di un pixel per volta su, giù, a sinistra o a destra, finché troverete nuovi pixel dello stesso colore di quello di partenza. Non troverete tutti i pixel al primo passaggio, però: dovete invece seguire un sistema simile a quello dell'espressione regolare di backtracking, descritta nel [Capitolo 9](#). Ogni volta che trovate più di una direzione possibile, dovete memorizzare tutte quelle che non seguite subito per riesaminarle alla fine dello spostamento corrente.

In un'immagine di dimensioni normali, ci sono un sacco di pixel. Pertanto, dovete sforzarvi di trovare una soluzione che comporti il minimo lavoro o il programma avrà tempi di esecuzione lunghissimi. Per esempio, ogni percorso dovrà ignorare i pixel incontrati nei passaggi precedenti, per non ripetere inutilmente del lavoro.

Vi suggerisco di richiamare `fillRect` per ogni pixel, quando ne incontrate uno che va colorato, e mantenere una qualche struttura dati che indichi quali pixel sono già stati esaminati.

# **Parte III**

**OLTRE**

## NODE.JS

*Uno studente chiese: “I programmatore del passato usavano solo macchine semplici e nessun linguaggio di programmazione, eppure producevano programmi bellissimi. Perché noi dobbiamo usare macchine complicate e linguaggi di programmazione?” Rispose Fu-Tzu: “I costruttori del passato usavano solo stecchi e argilla, eppure costruivano bellissime capanne.”*

Master Yuan-Ma, *The Book of Programming*

Finora, avete imparato il linguaggio JavaScript e lo avete usato in un solo ambiente: il browser. Questo capitolo e il prossimo presentano in breve Node.js, un programma che vi permette di applicare le vostre conoscenze di JavaScript al di fuori del browser. Con esso, potete realizzare qualunque cosa, da semplici strumenti a riga di comando, a server HTTP dinamici.

Questi capitoli puntano a insegnarvi i concetti fondamentali su cui si basa Node.js e a darvi abbastanza informazioni per poter scrivere alcuni utili programmi per Node, senza entrare troppo in dettaglio, né darvi su di esso informazioni complete.

Se volete eseguire il codice di questo capitolo, andate su [nodejs.org](http://nodejs.org) e seguite le istruzioni d'installazione per il vostro sistema operativo. Fate riferimento a quel sito Web anche per la documentazione completa su Node e sui suoi moduli integrati.

### Un po' di storia

Uno dei problemi più difficili nello scrivere sistemi di comunicazione in rete è la gestione di input e output: ossia, il leggere e lo scrivere dati da e sulla rete, sul disco rigido e su altri supporti o periferiche. Spostare dati richiede tempo e pianificarne gli spostamenti in maniera intelligente può fare una grossa differenza sui tempi di risposta del sistema alle richieste di rete o dell'utente.

Il modo tradizionale per gestire input e output è di fare in modo che una funzione, per esempio `readFile`, inizi a leggere un file e restituisca solo quando il file è stato letto completamente: di questo si parla come di *I/O sincrono*, dove I/O sta per input/output.

All'inizio, Node nacque con l'idea di semplificare le operazioni di I/O *asincrone*. Abbiamo già parlato di interfacce asincrone, come l'oggetto `XMLHttpRequest` del browser, discusso nel [Capitolo 17](#). Un'interfaccia asincrona fa in modo che l'esecuzione dello script continui per tutta la durata del lavoro e richiami una funzione di callback al termine. Ecco

come Node svolge tutte le operazioni di I/O.

JavaScript si presta molto bene a sistemi come Node. È uno dei pochi linguaggi di programmazione che non ha un suo sistema I/O. Pertanto, JavaScript può adattarsi senza rischi all'impostazione piuttosto eccentrica che Node dà alle interfacce di I/O. Nel 2009, quando nasceva Node, c'era già chi affrontava questioni di I/O basandosi sulle funzioni callback nel browser; la comunità che si raccoglieva intorno al linguaggio (JavaScript) era pertanto già abituata a uno stile di programmazione asincrono.

## Asincronicità

Proverò a illustrare le differenze tra I/O sincrono e I/O asincrono con l'esempio di un programma, che deve recuperare due risorse da Internet e svolgerci poi qualche operazione.

In un ambiente sincrono, il modo più ovvio per svolgere quel lavoro è di effettuare le richieste una dopo l'altra. Questo sistema ha lo svantaggio che la seconda richiesta avrà inizio solo al termine della prima. Il tempo complessivo necessario sarà pertanto almeno la somma dei due tempi di risposta. Questo non è un buon impiego della macchina, che sarà per lo più inattiva durante lo scambio di dati sulla rete.

La soluzione in un ambiente asincrono comincia con l'aggiunta di thread di controllo separati (nel [Capitolo 14](#) abbiamo parlato di thread). Un secondo thread può iniziare la seconda richiesta. I due thread aspettano ciascuno la propria risposta e a quel punto si risincronizzano per combinare i risultati.

Nel seguente diagramma, le linee più spesse rappresentano il tempo impiegato dalla macchina per l'esecuzione normale del programma, mentre quelle sottili rappresentano i tempi di attesa di I/O. Nel modello sincrono, il tempo richiesto per le operazioni di I/O fa parte del tempo richiesto per un dato thread di controllo. Nel modello asincrono, avviare un'operazione di I/O crea una *frattura* concettuale nel flusso del tempo. Il thread che ha iniziato la richiesta di I/O continua l'esecuzione e l'operazione di I/O viene svolta in parallelo, fin quando termina e richiama una funzione di callback.

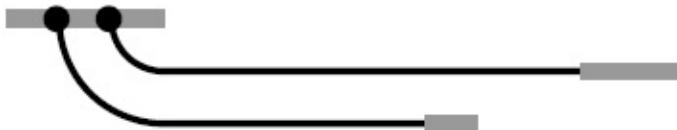
## Sincrono: un solo thread di controllo



## Sincrono: due thread di controllo



## Asincrono



Un altro modo per esprimere questa differenza, è che l'attesa è *implicita* nel modello sincrono, mentre è *esplicita* e direttamente sotto il nostro controllo in quello asincrono. Anche l'asincronicità, però, ha il suo rovescio: semplifica l'espressione di programmi che non si adattano facilmente al modello di controllo lineare, ma complica quella dei programmi lineari.

Nel [Capitolo 17](#), avevo già accennato al fatto che le funzioni di callback aggiungono confusione ai programmi. Possiamo discutere se questo stile di asincronicità sia meritevole o meno: di sicuro, bisogna farci l'abitudine.

In un sistema basato su JavaScript, comunque, sono dell'idea che l'asincronicità basata sulle funzioni di callback sia una scelta ragionevole. Uno dei punti di forza di JavaScript è la sua semplicità, che andrebbe persa se volessimo aggiungere più thread di controllo. Sebbene le funzioni di callback non producano codice semplice, sono concettualmente abbastanza semplici e potenti da permetterci di scrivere sistemi di server Web ad alte prestazioni.

## Il comando node

Installando Node.js sul vostro sistema, avrete a disposizione un programma, `node`, che si usa per eseguire file di JavaScript. Immaginate di avere un file, `hello.js`, che contenga questo codice:

```
var message = "Hello world";
console.log(message);
```

Per eseguire il programma, digitate `node` dalla riga di comando, come segue:

```
$ node hello.js
```

Il metodo `console.log` di Node si comporta in modo simile a quel che fa nel browser: stampa un brano di testo. In Node, però, il testo va al flusso di output standard del processo invece che a una console JavaScript nel browser.

Se eseguite il comando `node` senza specificare un file, vi viene proposto un prompt dove potete digitare del codice JavaScript e vederne immediatamente i risultati:

```
$ node
> 1 + 1
2
> [-1, -2, -3].map(Math.abs)
[1, 2, 3]
> process.exit(0)
$
```

---

La variabile `process`, proprio come `console`, è disponibile globalmente in Node e offre diversi modi per ispezionare e manipolare il programma corrente. Il metodo `exit` termina il processo e può ricevere un codice di stato che indica al programma che ha avviato `node` (in questo caso, l'interfaccia a riga di comando) se il programma ha avuto successo (codice di stato zero) o se ha incontrato un errore (qualunque altro codice di stato).

Per trovare gli argomenti di riga di comando assegnati ai vostri script, potete leggere `process.argv`, che è un array di stringhe. Notate che riporta anche il nome dei comandi `node` e il nome del vostro script, per cui gli argomenti iniziano in posizione di indice 2. Se `showarg.js` contiene solo l'istruzione `console.log(process.argv)`, potete digitare quanto segue:

```
$ node showargv.js one --and two
["node", "/home/marijn/showargv.js", "one", "--and", "two"]
```

---

Tutte le variabili globali standard di JavaScript, come `Array`, `Math` e `JSON`, sono presenti anche nell'ambiente di Node. È però assente la funzionalità relativa al browser, come `document` e `alert`.

L'oggetto ambito globale, che nel browser si chiama `window`, in Node ha il nome più corretto, `global`.

## Moduli

Oltre alle variabili che ho già citato, come `console` e `process`, Node prevede poca funzionalità nell'ambito globale. Se volete accedere ad altre funzionalità integrate, dovete cercarle nel sistema di moduli.

Del sistema di moduli CommonJS, che si basa sulla funzione `require`, abbiamo parlato nel [Capitolo 10](#). Questo sistema fa parte integrante di Node e si usa per caricare qualunque cosa, da moduli integrati a librerie scaricabili, a file che fanno parte del vostro programma.

Quando richiamate `require`, Node deve risolvere la stringa data e convertirla in un file da caricare. I nomi di percorso che iniziano con `"/"`, `"./"`, o `"../"` vengono risolti relativamente al percorso corrente del modulo, dove `"/"` indica la directory corrente, `".."` torna indietro di una directory e `"/"` indica la radice del sistema di file. Pertanto, se dal file `/home/marijn/elife/ run.js` chiedete `"/world/world"`, Node va a cercare il file `/home/marijn/elife/ world/world.js`. L'estensione `.js` può essere omessa.

Quando si passa a `require` una stringa che non sembra un percorso relativo o assoluto, il programma presume si tratti di un riferimento o a un modulo integrato, o a uno installato nella directory `node_modules`. Per esempio, `require("fs")` dà come risultato il modulo col sistema di file integrato di Node, mentre `require("elife")` tenta di caricare la libreria che si trova in `node_modules/elife/`. Un sistema comune per installare queste librerie si serve di NPM, di cui parlerò tra un momento.

Per illustrare l'uso di `require`, impostiamo un semplice progetto che consiste di due file. Il primo si chiama `main.js` e definisce uno script che si può richiamare dalla riga di comando per scombinare una stringa.

---

```
var garble = require("./garble");
// Index 2 holds the first actual command-line argument [La posizione 2 riporta il primo argomento di riga di comando] var argument =
process.argv[2];
console.log(garble(argument));
```

---

Il file `garble.js` definisce una libreria per distorcere delle stringhe, che si può usare sia dallo strumento riga di comando definito in precedenza, sia da altri script che hanno bisogno di accedere direttamente a una funzione di questo tipo.

---

```
module.exports = function(string) {
  return string.split("").map(function(ch) {
    return String.fromCharCode(ch.charCodeAt(0) + 5);
  }).join("");
};
```

---

Ricordate che sostituendo `module.exports`, invece di aggiungervi proprietà, possiamo esportare un valore specifico da un modulo. In questo caso, il risultato della richiesta del file `garble.js` è la funzione di manipolazione stessa.

La funzione divide la stringa che le viene passata in caratteri singoli, usando la striga vuota come divisore, e quindi sostituisce ognuno di essi col carattere con indice di cinque punti superiore. Infine, ricomponne il risultato in una stringa.

Possiamo ora richiamare il nostro strumento in questo modo:

```
$ node main.js JavaScript  
of{fxhwnuy
```

## Installazione con NPM

NPM, di cui ho parlato brevemente nel [Capitolo 10](#), è una raccolta online di moduli JavaScript, molti dei quali scritti specificamente per Node. Quando installate Node sul vostro computer, installate anche un programmino, `npm`, che offre una comoda interfaccia per quella raccolta.

Per esempio, un modulo che trovate su NPM è `figlet`, che converte il testo in *arte ASCII*, ossia in disegni tracciati con i caratteri. La seguente trascrizione mostra come installare e usare il programma:

```
$ npm install figlet
npm GET https://registry.npmjs.org/figlet
npm 200 https://registry.npmjs.org/figlet
npm GET https://registry.npmjs.org/figlet/-/figlet-1.0.9.tgz
npm 200 https://registry.npmjs.org/figlet/-/figlet-1.0.9.tgz
figlet@1.0.9 node_modules/figlet
$ node
> var figlet = require("figlet");
> figlet.text("Hello world!", function(error, data) {
    if (error)
        console.error(error);
    else
        console.log(data);
});
```



Al termine dell'installazione, troverete che NPM ha creato una directory di nome node\_modules. All'interno di quella directory troverete una directory figlet, che contiene la libreria. Quando eseguiamo node e richiamiamo require("figlet"), viene caricata questa libreria e possiamo richiamare il suo metodo text per disegnare il testo come nell'esempio.

Va detto che, invece di restituire semplicemente la stringa che costruisce le letterone, `figlet.text` accetta una funzione di callback alla quale passare il risultato, insieme a un argomento `error` che mantiene un oggetto errore se qualcosa non funziona, oppure `null` se tutto va bene.

Questo è abbastanza comune nel codice di Node. Per produrre il suo risultato, `figlet` richiede che la libreria legga un file contenente le forme dei caratteri. Poiché in Node recuperare quel file dal disco è un'operazione asincrona, `figlet.text` non può restituire immediatamente un risultato. In un certo senso, l'asincronicità è contagiosa: tutte le funzioni che richiamano una funzione asincrona devono essere a loro volta asincrone.

NPM offre molto di più di `npm install`. Può leggere file di pacchetti JSON (`package.json`), che contengono informazioni in formato JSON su programmi o librerie, quali quelle su cui dipende NPM. Installare `npm` in una directory che ne contenga installa automaticamente tutte le dipendenze e le dipendenze a esse relative. Lo strumento `npm` si usa anche per pubblicare librerie nella raccolta online e renderle disponibili perché altri le trovino, le scarichino e le usino.

Non entrerò ulteriormente nei particolari dell'uso di NPM. Per la documentazione completa e per un semplice sistema per trovare altre librerie, potete far riferimento direttamente a [npmjs.org](https://npmjs.org).

## Il modulo `filesystem`

Uno dei moduli integrati di uso più comune in Node è il modulo "`fs`", che sta per *filesystem* (sistema di file). Questo modulo offre funzioni per lavorare con file e directory.

Per esempio, esiste una funzione, `readFile`, che legge un file e richiama una funzione callback coi contenuti del file.

---

```
var fs = require("fs");
fs.readFile("file.txt", "utf8", function(error, text) {
  if (error)
    throw error;
  console.log("The file contained:", text);
});
```

---

Il secondo argomento di `readFile` indica la *codifica dei caratteri* usata per convertire il file in una stringa. Le codifiche per convertire il testo in dati binari sono parecchi; poiché nei sistemi più moderni si usa la codifica UTF-8, passare "`utf8`" quando si legge un file di testo è in genere appropriato, a meno che non abbiate motivo di pensare che sia stata usata una diversa codifica. Se non specificate la codifica, Node presume che vi interessino i dati binari e restituirà un oggetto `Buffer` invece di una stringa. Si tratta di un oggetto simile a un array che contiene dei numeri che rappresentano i byte del file.

---

```
var fs = require("fs");
```

```
fs.readFile("file.txt", function(error, buffer) {
  if (error)
    throw error;
  console.log("The file contained", buffer.length, "bytes.",
              "The first byte is:", buffer[0]);
});
```

---

Per scrivere un file sul disco si usa una funzione simile, `writeFile`.

```
var fs = require("fs");
fs.writeFile("graffiti.txt", "Node was here", function(err) {
  if (err)
    console.log("Failed to write file:", err);
  else
    console.log("File written.");
});
```

---

In questo caso, non c'è bisogno di specificare la codifica: se riceve una stringa invece di un oggetto Buffer, `writeFile` presume infatti di doverla scrivere come testo nella sua codifica di caratteri predefinita, che è UTF-8.

Il modulo "fs" contiene molte altre funzioni utili: `readdir` restituisce i file di una directory come array di stringhe, `stat` recupera informazioni su un file, `rename` rinomina un file, `unlink` ne elimina uno e così via. Trovate i particolari su [nodejs.org](http://nodejs.org).

Molte delle funzioni di "fs" hanno due varianti, una sincrona e una asincrona. Per esempio, esiste una versione sincrona di `readFile`, che si chiama `readFileSync`.

```
var fs = require("fs");
console.log(fs.readFileSync("file.txt", "utf8"));
```

---

Le funzioni sincrone sono meno ceremoniose da usare e possono essere utili in semplici script, dove la maggior velocità offerta dalle operazioni I/O asincrone è irrilevante. Notate però che, mentre viene eseguita l'operazione sincrona, il programma si interrompe completamente. Se il programma deve rispondere all'utente o ad altri computer sulla rete, restare in attesa che si completi la richiesta sincrona può creare antipatici ritardi.

## Il modulo HTTP

Un altro modulo centrale si chiama "http" e offre funzionalità per gestire server HTTP ed effettuare richieste HTTP.

Ecco quel che serve per avviare un semplice server HTTP:

---

```
var http = require("http");
var server = http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write("<h1>Hello!</h1><p>You asked for <code>" +
    request.url + "</code></p>");
  response.end();
});
server.listen(8000);
```

---

Se eseguite lo script sulla vostra macchina, potete far puntare il browser su <http://localhost:8000/hello> per effettuare una richiesta al vostro server, che risponderà con una piccola pagina HTML.

La funzione passata come argomento a `createServer` viene richiamata ogni volta che un client tenta di connettersi al server. Le variabili `request` e `response` sono oggetti che rappresentano i dati in entrata e in uscita. Il primo contiene informazioni sulla richiesta, tra cui la sua proprietà `url`, che indica per quale URL si è effettuata la richiesta.

Per mandare una risposta, richiamate dei metodi sull'oggetto `response`. Il primo, `writeHead`, compila le righe di header ([Capitolo 17](#)) e accetta un codice di stato (in questo caso, 200 per “OK”) e un oggetto che contiene i valori degli header. In questo caso, informiamo il client che invieremo un documento HTML.

A questo punto viene inviato il corpo della risposta (il documento richiesto) con `response.write`. Potete richiamare questo metodo più volte se volete trasmettere la risposta un pezzo per volta, possibilmente passando al client un flusso di dati man mano che risultano disponibili. Infine, `response.end` segnala la fine della risposta.

La chiamata a `server.listen` fa in modo che il server rimanga in attesa di connessioni sulla porta 8000. Ecco perché, per comunicare con questo server, dovete connettervi a `localhost:8000`, invece di `localhost` soltanto, che userebbe la porta predefinita, 80.

Per interrompere l'esecuzione di uno script Node come questo, che non termina automaticamente perché rimane in attesa di eventi (in questo caso, connessioni di rete), premete `CTRL-C`.

Un vero server Web svolge più operazioni di quelle riportate nell'esempio: esamina il metodo della richiesta (la proprietà `method`), per vedere quale azione il client vuole effettuare, e l'URL della richiesta per capire su quale risorsa si svolge l'azione. Più avanti in questo capitolo vedrete un server più completo.

Per svolgere operazioni come *client* HTTP, possiamo usare la funzione `request` nel modulo `"http"`.

---

```
var http = require("http");
var request = http.request({
```

```
hostname: "eloquentjavascript.net",
path: "/20_node.html",
method: "GET",
headers: {Accept: "text/html"}
}, function(response) {
  console.log("Server responded with status code",
    response.statusCode);
});
request.end();
```

---

Il primo argomento di `request` configura la richiesta, indicando il server al quale si vuole parlare, il percorso da richiedere a quel server, quale metodo usare e così via. Il secondo argomento è la funzione da richiamare quando arriva una risposta; essa viene passata come oggetto che ci permette di ispezionare la risposta, per esempio per trovare il suo codice di stato.

Proprio come l'oggetto `response` che abbiamo visto nel server, l'oggetto restituito da `request` ci permette di trasferire i dati in flusso col metodo `write` e terminare la richiesta col metodo `end`. L'esempio non usa `write` perché le richieste GET non contengono dati nel proprio corpo.

Per effettuare richieste di URL col protocollo sicuro (HTTPS), Node offre un pacchetto `https`, che contiene le proprie funzioni di richiesta, simili a quelle di `http.request`.

## Flussi di dati

Abbiamo visto due esempi di flussi di dati negli esempi HTTP: l'oggetto `response` sul quale può scrivere il server e l'oggetto `request` restituito da `http.request`.

I flussi di dati sono un concetto largamente usato nelle interfacce di Node. Tutti i flussi di dati in scrittura hanno un metodo `write`, che può ricevere una stringa o un oggetto `Buffer`. Il loro metodo `end` chiude il flusso e, se gli è stato passato un argomento, scriverà un blocco di dati prima di concludersi. Entrambi i metodi possono ricevere una funzione di callback come argomento aggiuntivo, che richiameranno quando hanno finito di scrivere o quando si chiude il flusso.

Con la funzione `fs.createWriteStream`, è possibile impostare un flusso di dati in scrittura che punti a un file. Potete quindi usare il metodo `write` sull'oggetto risultante per scrivere il file un pezzo per volta, invece che tutto in una volta come con `fs.writeFile`.

I flussi di dati in lettura sono un po' più complicati. Sia la variabile `request`, passata alla funzione di callback del server HTTP, sia la variabile `response` passata al client HTTP sono flussi di dati in lettura (un server legge richieste e scrive risposte, mentre il client prima scrive una richiesta e poi legge una risposta). La lettura dei flussi di dati si serve di

gestori di eventi, invece che di metodi.

Gli oggetti che fanno scattare eventi in Node hanno un metodo `on`, simile al metodo del browser `addEventListener`. Gli date un nome di evento e una funzione ed esso registrerà la chiamata a quella funzione ogni volta che si verifica l'evento dato.

I flussi in lettura hanno eventi "data" ed "end". Il primo scatta ogni volta che arrivano dei dati e il secondo viene richiamato quando il flusso arriva alla fine. Questo modello è adatto soprattutto per dati di "streaming", che possono essere elaborati immediatamente, anche se il documento non è ancora tutto disponibile. Un file si può leggere come flusso in lettura attraverso la funzione `fs.createReadStream`.

Il codice che segue crea un server che legge corpi di richieste e li restituisce al client in flussi, come testo in maiuscole.

---

```
var http = require("http");
http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  request.on("data", function(chunk) {
    response.write(chunk.toString().toUpperCase());
  });
  request.on("end", function() {
    response.end();
  });
}).listen(8000);
```

---

La variabile `chunk` passata al gestore di eventi `data` sarà un oggetto `Buffer` binario, che possiamo convertire in stringa richiamando `toString`, che lo convertirà con la codifica predefinita (UTF-8).

Il codice seguente, eseguito durante l'esecuzione del server di conversione in maiuscole, trasmetterà una richiesta a quel server e scriverà la risposta che ottiene:

---

```
var http = require("http");
var request = http.request({
  hostname: "localhost",
  port: 8000,
  method: "POST"
}, function(response) {
  response.on("data", function(chunk) {
    process.stdout.write(chunk.toString());
  });
});
```

```
request.end("Hello server");
```

---

Nell'esempio, si scrive su `process.stdout` (l'output standard del processo, come flusso in scrittura) invece che su `console.log`. Non possiamo usare `console.log` perché aggiunge un carattere di a capo dopo ogni brano di testo che scrive, cosa che in questo caso non va bene.

## Un semplice server di file

Mettiamo ora insieme le nostre conoscenze sui server HTTP e sul modo di comunicare col sistema di file per creare un ponte tra i due: un server HTTP che permette l'accesso remoto a un sistema di file. Un server di questo tipo ha molti usi: permette alle applicazioni basate sul Web di registrare e condividere dati o di dare a un gruppo di persone accesso condiviso a una serie di file.

Quando trattiamo i file come risorse HTTP, possiamo usare i metodi `GET`, `PUT` e `DELETE` del protocollo, rispettivamente per leggere, scrivere ed eliminare file. Interpreteremo il percorso nella richiesta come il percorso del file al quale la richiesta si riferisce.

Poiché non vogliamo condividere il nostro intero sistema di file, interpreteremo quei percorsi come aventi origine nella directory di lavoro del server, che è la directory dove si avvia il programma. Se il mio server gira sotto `/home/marijn/public/` (o `C:\Users\marijn\public\` in Windows), allora una richiesta per `/file.txt` si riferisce a `/home/marijn/public/file.txt` (o `C:\Users\marijn\public\file.txt`).

Costruiremo il programma un pezzo per volta, usando un oggetto `methods` per memorizzare le funzioni che gestiscono i vari metodi HTTP.

---

```
var http = require("http"), fs = require("fs");
var methods = Object.create(null);
http.createServer(function(request, response) {
    function respond(code, body, type) {
        if (!type) type = "text/plain";
        response.writeHead(code, {"Content-Type": type});
        if (body && body.pipe)
            body.pipe(response);
        else
            response.end(body);
    }
    if (request.method in methods)
        methods[request.method](urlToPath(request.url),
                               respond, request);
}
```

```
else
    respond(405, "Method " + request.method +
        " not allowed.");
}).listen(8000);
```

---

Con questo, avviamo un server che restituisce solo risposte di errore 405, il codice di stato che indica che il server non gestisce un certo metodo.

La funzione `respond` viene passata alle funzioni che gestiscono i vari metodi e agisce da funzione di callback per concludere la richiesta. Accetta come argomenti un codice di stato HTTP, un corpo e un eventuale tipo di contenuti. Se il valore passato come corpo è un flusso in lettura, userà un metodo `pipe` per trasferire un flusso in lettura a uno in scrittura. Altrimenti, si presume che il corpo sia o `null` (in quanto non esiste) oppure una stringa, e viene passato direttamente al metodo `end` della risposta.

Per ottenere un percorso dall'URL della richiesta, la funzione `urlToPath` fa uso del modulo integrato `"url"` per interpretare l'URL. Accetta il nome del percorso, che sarà una cosa come `/file.txt`, lo converte per eliminare i codici di escape come `%20` e aggiunge all'inizio un solo punto per ottenere un percorso relativo alla directory corrente.

---

```
function urlToPath(url) {
    var path = require("url").parse(url).pathname;
    return "." + decodeURIComponent(path);
}
```

---

Se vi preoccupa la sicurezza della funzione `urlToPath`, avete ragione. Ci torneremo sopra negli esercizi.

Impostiamo ora il metodo `GET` perché restituisca un elenco di file quando legge una directory e restituisca il contenuto del file quando legge un solo file.

Un punto interessante è quale header `Content-Type` scegliere quando restituiamo il contenuto di un file. Poiché questi file possono essere di qualunque tipo, il server non può restituire lo stesso tipo per tutti. NPM, però, ci aiuta: il pacchetto `mime` (gli indicatori del tipo di contenuto come `text/plain` si chiamano anche *tipi MIME*) riconosce il tipo giusto di un gran numero di estensioni.

Se eseguite il seguente comando `npm` nella directory dove vive lo script del server, potete usare `require("mime")` per ottenere accesso alla libreria:

---

```
$ npm install mime
npm http GET https://registry.npmjs.org/mime
npm http 304 https://registry.npmjs.org/mime
mime@1.2.11 node_modules/mime
```

---

Quando un file richiesto non esiste, il codice di errore HTTP da restituire è 404.

Useremo `fs.stat`, che esamina le informazioni sui file, per scoprire se il file esiste e se si tratta di una directory.

---

```
methods.GET = function(path, respond) {
  fs.stat(path, function(error, stats) {
    if (error && error.code == "ENOENT")
      respond(404, "File not found");
    else if (error)
      respond(500, error.toString());
    else if (stats.isDirectory())
      fs.readdir(path, function(error, files) {
        if (error)
          respond(500, error.toString());
        else
          respond(200, files.join("\n"));
      });
    else
      respond(200, fs.createReadStream(path),
              require("mime").lookup(path));
  });
};
```

Poiché deve interagire col disco e la cosa potrebbe richiedere del tempo, `fs.stat` è asincrono. Quando il file non esiste, `fs.stat` passa un oggetto errore con una proprietà "ENOENT" per code nella sua funzione di callback. Sarebbe carino se Node definisse diversi sottotipi di `Error` per errori diversi, ma purtroppo non lo fa. Invece, risponde solo con dei misteriosi codici in stile Unix.

Decidiamo di riportare tutti gli errori che non ci aspettavamo col codice di stato 500, che indica che il problema è nel server, mentre i codici di errore che iniziano con 4 (come 404) indicano errori di richiesta. In alcune situazioni, questa scelta non risulta accurata, ma per il nostro piccolo programma di esempio dobbiamo accontentarci.

L'oggetto status restituito da `fs.stat` ci dice un sacco di cose sul file, comprese le sue dimensioni (proprietà `size`) e l'ultimo aggiornamento (proprietà `mtime`). Qui ci interessa sapere se si tratta di una directory o di un file, cosa che scopriamo col metodo `isDirectory`.

Usiamo `fs.readdir` per leggere l'elenco dei file in una directory e, in un'altra funzione di callback, lo restituiamo all'utente. Per i file normali, creiamo un flusso in lettura con `fs.createReadStream` e lo passiamo a `respond`, insieme al tipo di contenuto rilevato dal nome file attraverso il modulo "mime".

Il codice per gestire richieste `DELETE` è un po' più semplice.

---

```
methods.DELETE = function(path, respond) {
  fs.stat(path, function(error, stats) {
    if (error && error.code == "ENOENT")
      respond(204);
    else if (error)
      respond(500, error.toString());
    else if (stats.isDirectory())
      fs.rmdir(path, respondErrorOrNothing(respond));
    else
      fs.unlink(path, respondErrorOrNothing(respond));
  });
};
```

---

Vi chiederete come mai cercare di eliminare un file restituisce uno stato 204, invece di un errore. Quando il file che vogliamo eliminare non esiste, potreste pensare che l'obiettivo della richiesta sia già stato soddisfatto. Lo standard HTTP incoraggia a effettuare richieste *idempotenti*, che significa che applicarle più volte non produce un risultato diverso.

---

```
function respondErrorOrNothing(respond) {
  return function(error) {
    if (error)
      respond(500, error.toString());
    else
      respond(204);
  };
}
```

---

Quando una risposta HTTP non contiene dati, si può usare il codice di stato 204 (“no content” o nessun contenuto). Poiché dobbiamo fornire funzioni callback che o riportano un errore o restituiscono una risposta 204 in alcune situazioni, ho scritto una funzione `respondErrorOrNothing`.

Questo è il gestore delle richieste PUT:

---

```
methods.PUT = function(path, respond, request) {
  var outStream = fs.createWriteStream(path);
  outStream.on("error", function(error) {
    respond(500, error.toString());
  });
  outStream.on("finish", function() {
```

```
    respond(204);
});
request.pipe(outStream);
};
```

---

Qui, non dobbiamo verificare se il file esiste. Se esiste, ci limitiamo a sovrascriverlo. Anche in questo caso usiamo pipe per spostare dati da un flusso in lettura a uno in scrittura, che qui avviene dalla richiesta al file. Se la creazione del flusso non riesce, viene lanciato un evento “error”, che riportiamo nella risposta. Quando i dati vengono trasferiti correttamente, pipe chiude entrambi i flussi e fa scattare un evento “finish” sul flusso in scrittura. A quel punto, possiamo comunicare al client una risposta positiva con lo stato 204.

Da [eloquentjavascript.net/code/file\\_server.js](http://eloquentjavascript.net/code/file_server.js) potete scaricare tutto lo script del server. Potete eseguirlo con Node per avviare il vostro server di file. E naturalmente potete modificarlo ed estenderlo per risolvere gli esercizi di questo capitolo o per fare altri esperimenti.

Per effettuare richieste HTTP, si può usare curl dalla riga di comando, una funzionalità largamente disponibile sui sistemi di tipo Unix. La sessione che segue effettua una piccola prova sul server. Notate che si usa -X per impostare il metodo della richiesta e -d per includere un corpo.

---

```
$ curl http://localhost:8000/file.txt
File not found
$ curl -X PUT -d hello http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
hello
$ curl -X DELETE http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
File not found
```

---

La prima richiesta di `file.txt` non ha successo perché il file non esiste ancora. La richiesta `PUT` crea il file e la richiesta successiva (miracolo!) lo recupera. Dopo averlo eliminato con una richiesta `DELETE`, il file risulta di nuovo mancante.

## Gestione degli errori

Nel codice per il server di file, ci sono sei posizioni dove dirottiamo esplicitamente le eccezioni che non sappiamo come gestire nelle risposte di errore. Poiché non si propagano automaticamente alle funzioni callback, ma vengono a esse passate come argomenti, le eccezioni vanno gestite esplicitamente ogni volta. Questo annulla completamente i vantaggi della gestione delle eccezioni e precisamente la possibilità di centralizzare la

gestione delle condizioni di errore.

Che cosa succede quando qualcosa *lancia* un'eccezione nel sistema? Poiché non stiamo usando blocchi `try`, l'eccezione si propaga fino in cima alla pila delle chiamate. In Node, ciò interrompe il programma e stampa informazioni sull'eccezione (compresa una traccia della pila) nel flusso di errore standard del programma.

Ciò significa che il server si bloccherà ogni volta che incontra un problema nel codice stesso; cosa che non succede con i problemi di tipo asincrono, che vengono passati come argomenti alle funzioni callback. Se volessimo gestire tutte le eccezioni sollevate nell'esecuzione di una richiesta ed essere sicuri di trasmettere una risposta, dovremmo aggiungere blocchi `try/catch` a tutte le funzioni callback.

Ciò però non può funzionare. Molti programmi per Node sono scritti in modo da gestire meno eccezioni possibile, in quanto si ritiene che eventuali eccezioni non possano essere gestite dal programma e la risposta migliore è che il programma smetta di funzionare.

Un'altra soluzione è di ricorrere alle promesse, descritte nel [Capitolo 17](#), che rilevano le eccezioni sollevate dalle funzioni callback e le propagano come errori. Potete caricare una libreria di promesse in Node e usare quella per gestire il controllo asincrono. Sono poche le librerie di Node che offrono promesse integrate, ma non è difficile impostarne. Il modulo "promise" di NPM contiene una funzione, `denodeify`, che accetta una funzione asincrona come `fs.readFile` e la converte in una funzione che restituisce una promessa.

---

```
var Promise = require("promise");
var fs = require("fs");
var readFile = Promise.denodeify(fs.readFile);
readFile("file.txt", "utf8").then(function(content) {
  console.log("The file contained: " + content);
}, function(error) {
  console.log("Failed to read file: " + error);
});
```

---

Potete confrontarla con la mia versione di un server di file basato su promesse, che trovate su [eloquentjavascript.net/code/file\\_server\\_promises.js](http://eloquentjavascript.net/code/file_server_promises.js). È un pochino più pulita in quanto le funzioni possono ora restituire dei risultati e il routing delle eccezioni è implicito invece che esplicito.

Riporto qualche riga di listato del server di file basato sulle promesse per illustrare la differenza di stile nella programmazione.

L'oggetto `fsp` di questo listato contiene varianti di tipo promessa di un gran numero di funzioni, avvolte da `Promise.denodeify`. L'oggetto restituito dal gestore del metodo, con proprietà `code` e `body`, diverrà il risultato finale della catena di promesse e verrà usato per stabilire che tipo di risposta trasmettere al client.

---

```
methods.GET = function(path) {
  return inspectPath(path).then(function(stats) {
    if (!stats) // Does not exist
      return {code: 404, body: "File not found"};
    else if (stats.isDirectory())
      return fsp.readdir(path).then(function(files) {
        return {code: 200, body: files.join("\n")};
      });
    else
      return {code: 200,
              type: require("mime").lookup(path),
              body: fs.createReadStream(path)};
  });
};

function inspectPath(path) {
  return fsp.stat(path).then(null, function(error) {
    if (error.code == "ENOENT") return null;
    else throw error;
  });
}
```

---

La funzione `inspectPath` avvolge `fs.stat`, che gestisce le situazioni dove il file non si trova. In quel caso, invece di un errore abbiamo un risultato `null` che indica il successo. Tutti gli altri errori si possono propagare. Quando la promessa restituita da quei gestori non ha esito, il server HTTP risponde con un codice di stato 500.

## Riepilogo

Node è un sistema semplice che ci permette di eseguire script JavaScript in un contesto diverso dal browser. Fu progettato inizialmente per svolgere compiti di rete, come nodo di reti, ma si presta a tutti i tipi di operazioni. Se vi piace JavaScript, automatizzare operazioni di routine quotidiana con Node dà ottimi risultati.

NPM offre librerie per tutte le esigenze, che potete recuperare e installare con un semplice comando. Node offre inoltre una serie di moduli integrati, compresi i moduli "`fs`", per lavorare col sistema di file, e "`http`" per gestire server HTTP ed effettuare richieste in quel protocollo.

In Node, tutte le operazioni di input e output si svolgono in modo asincrono, a meno che non scegliate deliberatamente una variante sincrona, come per esempio `fs.readFileSync`. Potete prevedere funzioni di callback, che Node richiamerà al

momento opportuno quando l'operazione di I/O che avete richiesto è completa.

## Esercizi

### Ancora negoziazione dei contenuti

Nel Capitolo 17, il primo esercizio era di effettuare diverse richieste a [eloquentjavascript.net/author](http://eloquentjavascript.net/author), chiedendo differenti tipi di contenuto attraverso diversi tipi di header Accept.

Ripetete l'esercizio usando la funzione `http.request` di Node. Come minimo, richiedete documenti di tipo `text/plain`, `text/html` e `application/json`. Ricordate che gli header di una richiesta si possono passare come oggetti, nella proprietà `headers` del primo argomento di `http.request`.

Stampate il contenuto delle risposte a ciascuna richiesta.

### Tappare una falla

Per semplificare l'accesso remoto ad alcuni file, potrei decidere di installare il server di file definito in questo capitolo sulla mia macchina, nella directory `/home/marijn/public`. Poi, un bel giorno, scopro che qualcuno ha ottenuto accesso a tutte le password che avevo memorizzato sul server.

Che cosa sarà successo?

Se non vi è chiaro, ripensate alla funzione `urlToPath`, definita come segue:

---

```
function urlToPath(url) {  
  var path = require("url").parse(url).pathname;  
  return "." + decodeURIComponent(path);  
}
```

---

Pensate ora che i percorsi passati alle funzioni “fs” possono essere relativi e contenere “`..`” per risalire di una directory. Che cosa succede quando un client trasmette richieste a URL come i seguenti?

---

```
http://myhostname:8000/.../.config/config/google-  
chrome/Default/Web%20Data  
http://myhostname:8000/.../.ssh/id_dsa  
http://myhostname:8000/.../.../etc/passwd
```

---

Modificate `urlToPath` per risolvere il problema. Tenete conto del fatto che Node su Windows ammette sia barre in avanti, sia barre rovesciate per separare le directory.

Riflettete anche sul fatto che appena mettete su Internet un sistema meno che blindato,

qualunque baco può essere sfruttato per danneggiare la macchina.

## **Creare directory**

Sebbene il metodo `DELETE` sia fatto per eliminare directory (attraverso `fs.rmdir`), il server di file per il momento non prevede sistemi per *creare* una nuova directory.

Aggiungete supporto per un metodo `MKCOL`, che crei una directory richiamando `fs.mkdir`. `MKCOL` non è uno dei metodi nativi di HTTP, ma esiste proprio per questo scopo nello standard *WebDAV*, che specifica una serie di estensioni per HTTP per scrivere risorse e non solo per leggerle.

## **Uno spazio pubblico sul Web**

Visto che il server di file può gestire qualunque tipo di file con l'header `Content-Type` appropriato, potete usarlo per ospitare un sito Web. Dal momento che consente a chiunque di eliminare e sostituire file, sarebbe un esempio interessante di sito Web, che può essere modificato, vandalizzato e distrutto da chiunque sappia impostare una richiesta HTTP adatta. Eppure, è sempre un sito Web.

Impostate una semplice pagina HTML che contenga un semplice script JavaScript. Trasferite i file in una directory del server di file e apriteli nel browser.

Come evoluzione del progetto, soprattutto se volete passarci tutto il fine settimana, raccogliete tutte le conoscenze che avete acquisito da questo libro per costruire un'interfaccia più intuitiva per modificare il sito dal suo interno.

Usate un modulo HTML ([Capitolo 18](#)) per modificare il contenuto dei file che costituiscono il sito, permettendo all'utente di aggiornarli sul server attraverso le richieste HTTP descritte nel [Capitolo 17](#).

Cominciate con un solo file modificabile. Poi, fate in modo che l'utente possa selezionare i file da modificare. Approfittate del fatto che il nostro server di file restituisce elenchi di file quando legge una directory.

Non lavorate direttamente sul codice che si trova sul server, perché se fate un errore rischiate di compromettere tutto. Invece, salvate il lavoro al di fuori della directory accessibile pubblicamente e copiatevelo solo quando siete pronti a provarlo.

Se il vostro computer è connesso direttamente a Internet, senza un firewall, un router o altre periferiche di isolamento, potete chiedere a un amico di provare il vostro sito. Per vedere se può funzionare, andate su [whatismyip.com](http://whatismyip.com), copiate l'indirizzo IP indicato nella barra degli indirizzi del browser e aggiungete: 800 dopo l'indirizzo IP per selezionare la porta giusta. Se arrivate al vostro sito, vuol dire che è online e tutti lo possono vedere.

## PROGETTO: UN SITO WEB DI SKILL-SHARING

Una riunione di *skill-sharing* (condivisione di talenti) è un evento dove si incontrano persone che hanno un interesse comune per scambiarsi piccole presentazioni informali sulle conoscenze che ciascuno ha sull'argomento. Per esempio, a una riunione di skill-sharing sul giardinaggio qualcuno può spiegare come si coltiva il sedano. Oppure, in un gruppo che si interessa di programmazione, potreste andare a raccontare come funziona Node.js.

Queste riunioni, spesso chiamate *User Group* quando hanno per argomento i computer, sono ottime occasioni per ampliare il proprio orizzonte, imparare nuove aree da sviluppare o semplicemente incontrarsi con persone con interessi simili. Nelle grandi città è probabile che esistano già degli eventi di questo tipo su JavaScript. Di solito, partecipare non costa nulla; quelli che ho visitato mi sono sembrati molto cordiali e amichevoli.

In quest'ultimo progetto, lo scopo è di impostare un sito Web dove gestire le presentazioni date in un incontro di skill-sharing. Immaginate che ci sia un gruppetto di persone che si incontra regolarmente nell'ufficio di uno di loro per parlare di monocicli. Quando l'organizzatore degli incontri si trasferisce in un'altra città, nessuno si fa avanti per sostituirlo: dobbiamo risolvere questo problema con un sistema che consenta ai partecipanti di proporre e discutere presentazioni tra di loro, senza un organizzatore preposto.



Potete scaricare il codice completo per il progetto da <http://eloquentjavascript.net/code/skillsharing.zip>.

## Piano del progetto

Il progetto ha una parte *server*, scritta per Node.js, e una parte *client*, scritta per il browser. Il server memorizza i dati del sistema e li fornisce al client. Gestisce inoltre i file HTML e JavaScript del sistema per il lato client.

Il server mantiene un elenco delle presentazioni proposte per la riunione successiva e il client lo visualizza. Ogni presentazione riporta il nome del presentatore, un titolo, un sommario e un elenco di commenti associati. Il client permette agli utenti di proporre nuovi argomenti (aggiungendoli all'elenco), eliminarne e commentare quelli esistenti. Quando un utente effettua una di queste richieste, il client trasmette una richiesta HTTP per istruire il server.

The screenshot shows a web page with a form at the top where a user can enter their name ('Bob'). Below the form, there is a presentation proposal card. The card has a title 'Unituning' in bold, followed by 'by Carlos'. Underneath the title is a summary: 'Modifying your cycle for extra style'. Below the summary, there are three lines of text from participants: 'Alice: Will you talk about raising a cycle?', 'Carlos: Definitely!', and 'Alice: I'll be there'. At the bottom of the card are two buttons: 'Add comment' and 'Delete talk'.

L'applicazione sarà impostata in modo che sia visibile un elenco *dal vivo* delle presentazioni proposte, con i relativi commenti. Ogni volta che qualcuno, non importa dove, propone una nuova presentazione o aggiunge un commento, tutti quelli che hanno la pagina aperta nel browser vedono immediatamente gli aggiornamenti. Questa è un po' una sfida, in quanto non c'è modo per il server Web di aprire una connessione col client, né si può sapere con certezza quali client stanno esaminando un dato sito Web in un determinato momento.

Una soluzione comune a questo problema è il cosiddetto *long polling*, che è proprio uno degli obiettivi di progettazione per Node.

## Modello di comunicazione long polling

Per poter notificare immediatamente a un client che qualcosa è cambiato, dobbiamo avere una connessione con quel client. Poiché i browser tradizionalmente non accettano connessioni e i client si trovano dietro strutture che le bloccherebbero comunque, non ha molto senso che a iniziare lo scambio sia il server.

Possiamo fare in modo che il client apra la connessione e la tenga aperta, in modo che il server possa usarla per trasmettere informazioni quando serve.

Una richiesta http, però, permette solo un semplice flusso di dati, dove il client invia una richiesta, il server replica con una sola risposta e tutto finisce lì. Esiste una tecnologia

che trova supporto nei browser moderni, detta *WebSockets*, che consente di aprire delle connessioni per lo scambio di dati arbitrari. Usarla correttamente, però, non è facile.

In questo capitolo, utilizzeremo una tecnica relativamente semplice, detta long polling, dove i client continuano a chiedere nuovi dati al server attraverso richieste HTTP normali; quando non trova nulla di nuovo, il server si limita a tenere in sospeso la risposta.

Fintanto che il client ha una richiesta di polling aperta, le informazioni dal server arriveranno immediatamente. Per esempio, se Alice ha la nostra applicazione per skill-sharing aperta nel browser, il suo browser avrà trasmesso una richiesta di aggiornamenti e rimarrà in attesa di una risposta. Quando Gianni propone una sua presentazione al gruppo “Monociclo da discesa estremo”, il server rileva che Alice è in attesa di aggiornamenti e le trasmette informazioni sulla nuova presentazione in risposta alla sua richiesta aperta. Il browser di Alice riceve i dati e aggiorna lo schermo per mostrare la presentazione.

Per evitare che le connessioni scadano (vengano chiuse per mancanza di attività), le tecniche di long polling permettono di impostare un tempo massimo per ciascuna richiesta, scaduto il quale il server risponde comunque, anche se non c’è nulla di nuovo da riferire, e il client apre una nuova richiesta. Riavviare periodicamente la richiesta rende anche la tecnica più robusta, in quanto permette ai client di recuperare i dati se si verificano interruzioni temporanee sulla connessione o problemi col server.

Un server occupato in operazioni di long polling può avere migliaia di richieste in attesa, con altrettante connessioni TCP aperte. Node, che semplifica la gestione di connessioni multiple senza creare thread di controllo separati per ciascuna, offre una buona soluzione per questo problema.

## Interfaccia HTTP

Prima di affrontare il server e il client, pensiamo al punto dove si incontrano: l’interfaccia HTTP sulla quale comunicano.

Programmiamo l’interfaccia in modo che si basi su JSON e, come nel file server del [Capitolo 20](#), cercheremo di fare buon uso dei metodi HTTP. L’interfaccia ha come punto centrale il percorso `/talks`. Per i file statici, ossia la pagina HTML e il codice JavaScript per il sistema lato client, useremo percorsi che non iniziano con `/talks`.

Una richiesta GET a `/talks` restituisce un documento JSON come questo:

---

```
{"serverTime": 1405438911833,
  "talks": [{"title": "Unituning",
             "presenter": "Carlos",
             "summary": "Modifying your cycle for extra style",
             "comment": []}]}

---


```

Il campo `serverTime` sarà usato per permettere operazioni affidabili di long polling e ci torneremo sopra più avanti.

Per creare una nuova presentazione, si effettua una richiesta PUT a un URL come /talks/Unituning, dove la parte che segue la seconda barra è il titolo della presentazione. Il corpo della richiesta PUT deve contenere un oggetto JSON con proprietà `presenter` (presentatore) e `summary` (sommario).

Poiché i titoli delle presentazioni possono contenere spazi e altri caratteri che normalmente non compaiono in un URL, le relative stringhe vanno convertite con la funzione `encodeURIComponent`.

---

```
console.log("/talks/" + encodeURIComponent("How to Idle"));
// → /talks/How%20to%20Idle
```

---

Una richiesta per creare una presentazione sul surplace può assomigliare alla seguente:

```
PUT /talks/How%20to%20Idle HTTP/1.1
Content-Type: application/json
Content-Length: 92
{"presenter": "Dana",
"summary": "Standing still on a unicycle"} [Stare fermi sul monociclo]
```

---

Questi URL offrono inoltre supporto a richieste GET per recuperare la rappresentazione in formato JSON della presentazione e DELETE per eliminare una presentazione.

Per aggiungere un commento, si usa una richiesta POST a URL come /talks/Unituning/comments, con un oggetto JSON che ha autore e messaggio memorizzati nelle proprietà `author` e `message` nel corpo della richiesta.

---

```
POST /talks/Unituning/comments HTTP/1.1
Content-Type: application/json
Content-Length: 72
{"author": "Alice",
"message": "Will you talk about raising a cycle?"} [Parlerai di come
alzare il monociclo?]
```

---

Per dare supporto al modello long polling, le richieste GET devono riportare un parametro `changesSince`, che serve al client per indicare che vuole gli aggiornamenti avvenuti da un certo momento. Se ci sono aggiornamenti, vengono restituiti immediatamente. Quando non ce ne sono, la risposta viene ritardata finché succede qualcosa o finché non è trascorso un certo periodo di tempo (qui useremo 90 secondi).

L'intervallo di tempo va indicato come numero di millisecondi trascorsi dall'inizio del 1970, che è lo stesso tipo di numero restituito dalla funzione `Date.now()`. Perché possa ricevere tutti gli aggiornamenti senza doppioni, il client deve passare l'orario dell'ultima comunicazione ricevuta dal server. L'orologio del server potrebbe non essere in perfetta sintonia con quello del client, ma anche se lo fosse, sarebbe impossibile per il client sapere

l'ora precisa di quando il server ha trasmesso una risposta, perché ci vuole sempre del tempo per trasferire dati in rete.

Questo spiega l'esistenza della proprietà `serverTime` nelle risposte trasmesse a richieste GET per `/talks`. Quella proprietà indica al client l'ora esatta, dal punto di vista del server, della creazione dei dati che ha ricevuto. Il client può ora memorizzare l'orario e passarlo nella sua richiesta di polling successiva per evitare che il server ritrasmetta aggiornamenti già visti.

---

```
GET /talks?changesSince=1405438911833 HTTP/1.1
(time passes)      (passa del tempo)
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 95
{"serverTime": 1405438913401,
 "talks": [{"title": "Unituning",
            "deleted": true}]}

---


```

Quando una presentazione è stata modificata o è appena stata creata o ha ricevuto un nuovo commento, la rappresentazione completa della presentazione viene inserita nella risposta alla richiesta di polling successiva. Quando viene eliminata, vengono inseriti nella risposta solo il titolo e la proprietà `deleted`. Il client può quindi aggiungere allo schermo le presentazioni con titoli non ancora visti, aggiornare quelle già presenti e cancellare quelle che sono state eliminate sul server.

Il protocollo descritto in questo capitolo non ha alcun tipo di controllo di accesso. Chiunque può commentare, modificare e persino eliminare presentazioni. Poiché Internet è pieno di vandali, mettere online un sistema come questo senza qualche forma di protezione sarebbe un disastro.

Una soluzione semplice è di anteporre al sistema un *proxy inverso*, ossia un server HTTP che accetta connessioni esterne al sistema e le ritrasmette ai server HTTP che sono in esecuzione locale. Questi proxy possono essere configurati per richiedere nome utente e password; potrete pertanto fare in modo che solo i partecipanti (legittimi) del gruppo di skill-sharing abbiano la password necessaria.

## Il server

Cominciamo adesso a impostare la parte lato server del programma. Il codice di questa parte va eseguito sotto Node.js.

### **Routing**

Il nostro server usa `http.createServer` per avviare un server HTTP. Nella funzione che gestisce le nuove richieste, dobbiamo distinguerle per tipo, in base a metodo e percorso.

Potremmo farlo con una lunga catena di istruzioni `if`, ma esiste un sistema migliore.

Un *router* è un componente che aiuta a recapitare le richieste alle funzioni in grado di gestirle. Potete per esempio istruire il router che le richieste `PUT` con un percorso che soddisfa l'espressione regolare `/^\talks\/(.+)/$` (ossia, che corrisponde a `/talks/` seguito da un titolo) possono essere gestite da una funzione data. Inoltre, il router può estrarre le parti più significative del percorso (in questo caso il titolo della presentazione), che sono tra parentesi nell'espressione regolare, e passarle alla funzione di gestione.

Ci sono diversi pacchetti router su NPM; qui però ne scriviamo uno nuovo per illustrare il principio.

Questo è il listato di `router.js`, che più avanti richiameremo con `require` dal modulo del server:

---

```
var Router = module.exports = function() {
  this.routes = [];
};

Router.prototype.add = function(method, url, handler) {
  this.routes.push({method: method,
                    url: url,
                    handler: handler});
};

Router.prototype.resolve = function(request, response) {
  var path = require("url").parse(request.url).pathname;
  return this.routes.some(function(route) {
    var match = route.url.exec(path);
    if (!match || route.method != request.method)
      return false;
    var urlParts = match.slice(1).map(decodeURIComponent);
    route.handler.apply(null, [request, response]
                           .concat(urlParts));
    return true;
  });
};
```

---

Il modulo esporta il costruttore `Router`. Un oggetto `router` consente la registrazione di nuove funzioni di gestione col metodo `add` e può risolvere richieste col metodo `resolve`.

Quest'ultimo restituisce un valore booleano, che indica se ha trovato una funzione di gestione. Il metodo `some` sull'array di direzioni possibili le prova una per volta, nell'ordine in cui sono state definite, e si ferma restituendo `true` quando ne trova una adatta.

Le funzioni di gestione vengono richiamate con gli oggetti `request` e `response`. Quando l'espressione regolare che corrisponde all'URL contiene dei gruppi, le stringhe corrispondenti vengono passate alla funzione di gestione come argomenti aggiuntivi. Queste stringhe vanno convertite in URL, in quanto l'URL grezzo contiene codici di stile `%20`.

## Servire i file

Quando una richiesta trova un tipo corrispondente tra quelli definiti per il router, il server deve interpretarla come una richiesta per un file della directory pubblica. Si potrebbe usare il server di file definito nel [Capitolo 20](#), ma non abbiamo bisogno e non vogliamo offrire supporto per richieste `PUT` o `DELETE`, mentre sarebbe bello avere qualche funzionalità avanzata, come il supporto per la memorizzazione in cache. Ci orientiamo pertanto verso un file server solido e statico, ampiamente verificato, che troviamo su NPM.

Ho scelto `ecstatic`. Non è l'unico server di questo tipo su NPM, ma funziona bene e risponde ai nostri requisiti. Il modulo `ecstatic` esporta una funzione che può essere richiamata con un oggetto di configurazione per produrre una funzione di gestione delle richieste. Usiamo l'opzione `root` per indicare al server dove cercare i file. La funzione di gestione accetta parametri `request` e `response` e può essere passata direttamente a `createServer` per impostare un server di soli file. Poiché vogliamo prima di tutto andare a vedere le richieste che ci interessano, la avvolgiamo in un'altra funzione.

---

```
var http = require("http");
var Router = require("./router");
var ecstatic = require("ecstatic");
var fileServer = ecstatic({root: "./public"});
var router = new Router();
http.createServer(function(request, response) {
  if (!router.resolve(request, response))
    fileServer(request, response);
}).listen(8000);
```

---

Le funzioni ausiliarie `respond` e `respondJSON` sono usate nel codice del server per trasmettere risposte con una sola chiamata di funzione.

---

```
function respond(response, status, data, type) {
  response.writeHead(status, {
    "Content-Type": type || "text/plain"
  });
  response.end(data);
}

function respondJSON(response, status, data) {
```

```
    respond(response, status, JSON.stringify(data),  
            "application/json");  
}
```

---

## Presentazioni come risorse

Il server mantiene le presentazioni proposte in un oggetto `talks`, che contiene proprietà i cui nomi sono i titoli delle presentazioni. Poiché le presentazioni saranno esposte come risorse HTTP in `/talks/[titolo]`, dobbiamo aggiungere al router delle funzioni di gestione per i vari metodi che il client può usare per manipolarle.

Il gestore per le richieste GET per ciascuna presentazione deve esaminare la presentazione e rispondere o con i rispettivi dati in formato JSON o con una risposta di errore 404.

---

```
var talks = Object.create(null);  
router.add("GET", /^\/talks\/([^\/]*)$/,  
          function(request, response, title) {  
            if (title in talks)  
              respondJSON(response, 200, talks[title]);  
            else  
              respond(response, 404, "No talk '" + title + "' found");  
});
```

---

Per cancellare una presentazione, la si elimina dall'oggetto `talks`.

```
router.add("DELETE", /^\/talks\/([^\/]*)$/,  
           function(request, response, title) {  
             if (title in talks) {  
               delete talks[title];  
               registerChange(title);  
             }  
             respond(response, 204, null);  
           });
```

---

La funzione `registerChange`, che definiremo più avanti, si occupa delle notifiche in long polling alle richieste in attesa.

Per recuperare il contenuto dei corpi delle richieste in formato JSON, definiamo una funzione `readStreamAsJSON`, che legge il contenuto di un flusso, lo interpreta come JSON e richiama quindi una funzione di callback.

---

```
function readStreamAsJSON(stream, callback) {
```

```

var data = "";
stream.on("data", function(chunk) {
  data += chunk;
});
stream.on("end", function() {
  var result, error;
  try { result = JSON.parse(data); }
  catch (e) { error = e; }
  callback(error, result);
});
stream.on("error", function(error) {
  callback(error);
});
}

```

---

Una funzione di gestione che deve leggere risposte JSON è quella per il metodo `PUT`, che si usa per creare nuove presentazioni. Deve prima di tutto verificare se i dati hanno proprietà `presenter` e `summary`, che sono stringhe. Altri dati provenienti dall'esterno potrebbero essere robaccia e non vogliamo corrompere il modello interno dei dati, né tantomeno rischiare un crash, quando arrivano richieste non adatte.

Se i dati sembrano validi, la funzione di gestione memorizza un oggetto che rappresenta la nuova presentazione nell'oggetto `talks`, eventualmente sovrascrivendo una presentazione esistente con lo stesso titolo, e richiama di nuovo `registerChange`.

---

```

router.add("PUT", /^\/talks\/([^\/]+)$/,
  function(request, response, title) {
  readStreamAsJSON(request, function(error, talk) {
    if (error) {
      respond(response, 400, error.toString());
    } else if (!talk ||
      typeof talk.presenter != "string" ||
      typeof talk.summary != "string") {
      respond(response, 400, "Bad talk data");
    } else {
      talks[title] = {title: title,
                    presenter: talk.presenter,
                    summary: talk.summary,
                    comments: []};
      registerChange(title);
    }
  });
}

```

```
    respond(response, 204, null);
}
});
});
```

---

Aggiungere un commento a una presentazione funziona in modo simile. Usiamo `readStreamAsJSON` per recuperare il contenuto della richiesta, convalidare i dati e, se sembrano a posto, memorizzarli come commento.

---

```
router.add("POST", /^\/talks\/([^\/]+)/comments$/,
  function(request, response, title) {
  readStreamAsJSON(request, function(error, comment) {
    if (error) {
      respond(response, 400, error.toString());
    } else if (!comment ||
      typeof comment.author != "string" ||
      typeof comment.message != "string") {
      respond(response, 400, "Bad comment data");
    } else if (title in talks) {
      talks[title].comments.push(comment);
      registerChange(title);
      respond(response, 204, null);
    } else {
      respond(response, 404, "No talk '" + title + "' found");
    }
  });
});
```

---

Naturalmente, cercare di aggiungere un commento a una presentazione inesistente deve restituire un errore 404.

## **Supporto per long polling**

L'aspetto più interessante del server è la parte che si occupa del modello long polling. Quando arriva una richiesta GET per `/talks`, può essere una semplice richiesta per tutte le presentazioni o una richiesta di aggiornamento con un parametro `changesSince`.

Ci saranno situazioni dove dobbiamo trasmettere al client un elenco di presentazioni; pertanto, definiamo per prima cosa una piccola funzione ausiliaria che aggiunge il campo `serverTime` a queste risposte.

---

```
function sendTalks(talks, response) {
```

```
respondJSON(response, 200, {
  serverTime: Date.now(),
  talks: talks
});
}
```

---

La funzione di gestione deve ora esaminare i parametri nell'URL della richiesta per vedere se esiste un parametro `changesSince`. Se alla funzione `parse` del modulo "url" passate un secondo argomento con valore `true`, la funzione analizzerà anche la parte query dell'URL. L'oggetto restituito avrà una proprietà `query`, che mantiene un altro oggetto che mappa nomi di parametri con i rispettivi valori.

---

```
router.add("GET", /^\/talks$/, function(request, response) {
  var query = require("url").parse(request.url, true).query;
  if (query.changesSince == null) {
    var list = [];
    for (var title in talks)
      list.push(talks[title]);
    sendTalks(list, response);
  } else {
    var since = Number(query.changesSince);
    if (isNaN(since)) {
      respond(response, 400, "Invalid parameter");
    } else {
      var changed = getChangedTalks(since);
      if (changed.length > 0)
        sendTalks(changed, response);
      else
        waitForChanges(since, response);
    }
  }
});
```

---

Quando manca il parametro `changesSince`, la funzione di gestione mette insieme un elenco di tutte le presentazioni e restituisce quello.

Altrimenti, bisogna prima esaminare il parametro `changesSince` per verificare che sia un numero valido. La funzione `getChangedTalks`, che definiremo tra poco, restituisce un array di presentazioni che sono state modificate da un certo orario. Se l'array restituito è vuoto, il server non ha ancora nulla da trasmettere al client e si limita a memorizzare l'oggetto `response` (attraverso `waitForChanges`) al quale risponderà in futuro.

---

```
var waiting = [];

function waitForChanges(since, response) {
  var waiter = {since: since, response: response};
  waiting.push(waiter);
  setTimeout(function() {
    var found = waiting.indexOf(waiter);
    if (found > -1) {
      waiting.splice(found, 1);
      sendTalks([], response);
    }
  }, 90 * 1000);
}
```

---

Col metodo `splice` si “ritaglia” una parte di un array. Gli si passano un indice e una serie di elementi e il metodo *muta* l’array, eliminando tutti gli elementi che vengono dopo la posizione di indice data. In questo caso, eliminiamo un solo elemento, l’oggetto che rimane in attesa della risposta, il cui indice si trova richiamando `indexOf`. Se passate altri argomenti a `splice`, i loro valori verranno inseriti nell’array nella posizione data, sostituendo gli elementi eliminati.

Quando si memorizza un oggetto `response` nell’array in attesa, viene immediatamente impostato un intervallo di timeout. Dopo 90 secondi, si verifica se la richiesta è ancora in attesa e, se lo è, si trasmette una risposta vuota e la si elimina dall’array in attesa.

Per trovare solo le presentazioni che sono state aggiornate da un determinato momento, dobbiamo seguire la storia delle modifiche. Registrando un evento corrispondente con `registerChange`, il metodo terrà in memoria la modifica, insieme all’ora corrente, in un array `changes`. Quando ha luogo una modifica, vuol dire che ci sono nuovi dati e si può pertanto rispondere immediatamente a tutte le richieste in attesa.

---

```
var changes = [];

function registerChange(title) {
  changes.push({title: title, time: Date.now()});
  waiting.forEach(function(waiter) {
    sendTalks(getChangedTalks(waiter.since), waiter.response);
  });
  waiting = [];
}
```

---

Infine, `getChangedTalks` usa l’array `changes` per costruire un array di presentazioni aggiornate, che comprende gli oggetti con una proprietà `deleted` per le presentazioni che non esistono più. Nel costruire quell’array, `getChanged-Talks` deve controllare che non

riporti due volte la stessa presentazione, in quanto possono essere avvenuti più aggiornamenti della stessa presentazione durante l'intervallo considerato.

---

```
function getChangedTalks(since) {
  var found = [];
  function alreadySeen(title) {
    return found.some(function(f) {return f.title == title;});
  }
  for (var i = changes.length - 1; i >= 0; i--) {
    var change = changes[i];
    if (change.time <= since)
      break;
    else if (alreadySeen(change.title))
      continue;
    else if (change.title in talks)
      found.push(talks[change.title]);
    else
      found.push({title: change.title, deleted: true});
  }
  return found;
}
```

---

E con questo abbiamo finito col codice del server. Eseguendo il programma definito finora, avrete un server in esecuzione sulla porta 8000, che serve file dalla sottodirectory pubblica insieme a un'interfaccia di gestione delle presentazioni sotto l'URL /talks.

## Il client

Il lato client del sito consiste di tre file: una pagina HTML, un foglio stile e un file JavaScript.

### HTML

Quando ricevono una richiesta per un percorso che corrisponde a una directory, uno dei comportamenti convenzionali più comunemente seguiti dai server Web è di cercare un file con nome `index.html`. Il modulo `ecstatic` che usiamo come server di file segue questa convenzione. Quando riceve una richiesta per il percorso `/`, il server va a cercare il file `./public/index.html` (dove `./public` è la radice che abbiamo passato) e, se lo trova, restituisce quel file.

Pertanto, se vogliamo che venga visualizzata una pagina quando un browser punta al nostro server, possiamo metterla in `public/index.html`. Ecco come inizia il nostro file

indice:

---

```
<!doctype html>
<title>Skill Sharing</title>
<link rel="stylesheet" href="skillsharing.css">
<h1>Skill sharing</h1>
<p>Your name: <input type="text" id="name"></p>
<div id="talks"></div>
```

---

Il codice assegna un titolo al documento e richiama un foglio stile, che definisce alcuni stili, uno dei quali aggiunge un bordo intorno a ciascuna presentazione. Seguono un titolo e un campo per il nome utente. L'utente deve inserirvi il suo nome, che sarà aggiunto alle presentazioni e ai commenti che proporrà.

L'elemento `<div>` con `id="talks"` conterrà l'elenco corrente delle presentazioni. Lo script compila l'elenco quando riceve le presentazioni dal server.

Segue poi il modulo per creare una nuova presentazione.

---

```
<form id="newtalk">
  <h3>Submit a talk</h3>
  Title: <input type="text" style="width: 40em" name="title">
  <br>
  Summary: <input type="text" style="width: 40em" name="summary">
  <button type="submit">Send</button>
</form>
```

---

Lo script aggiunge un gestore di eventi "submit" a questo modulo, dal quale effettuare le richieste HTTP per informare il server.

Segue un blocco abbastanza misterioso, il cui attributo `display` è impostato su `none` perché non venga visualizzato sullo schermo. Riuscite a capire di che si tratta?

---

```
<div id="template" style="display: none">
  <div class="talk">
    <h2>{{title}}</h2>
    <div>by <span class="name">{{presenter}}</span></div>
    <p>{{summary}}</p>
    <div class="comments"></div>
    <form>
      <input type="text" name="comment">
      <button type="submit">Add comment</button> [Aggiungi commento]
      <button type="button" class="del">Delete talk</button>
```

```
[Elimina presentazione]
    </form>
</div>
<div class="comment">
    <span class="name">{{author}}</span>: {{message}}
</div>
</div>
```

---

Creare delle strutture DOM complicate con JavaScript produce del codice di pessimo aspetto. Potete migliorarlo leggermente introducendo funzioni ausiliarie come la funzione `elt` del [Capitolo 13](#), ma il risultato sarà sempre più brutto dell'HTML, che è un linguaggio specifico proprio per esprimere strutture DOM.

Per creare strutture DOM per le presentazioni, nel programma definiamo un semplice *sistema di modelli*, che fa uso di strutture DOM nascoste, ma inserite nel documento, per istanziare nuove strutture DOM sostituendo i marcatori tra doppie parentesi con i valori di una data presentazione.

Infine, il documento HTML richiama il file con lo script che contiene il codice lato client.

---

```
<script src="skillsharing_client.js"></script>
```

---

## Si parte

La prima cosa che il client deve fare quando viene caricata la pagina è chiedere al server l'elenco corrente delle presentazioni. Poiché effettueremo molte richieste HTTP, definiamo una piccola funzione wrapper per avvolgere XMLHttpRequest, che accetta un oggetto per configurare la richiesta e una funzione di callback per quando la richiesta ha terminato.

---

```
function request(options, callback) {
    var req = new XMLHttpRequest();
    req.open(options.method || "GET", options.pathname, true);
    req.addEventListener("load", function() {
        if (req.status < 400)
            callback(null, req.responseText);
        else
            callback(new Error("Request failed: " + req.statusText));
    });
    req.addEventListener("error", function() {
        callback(new Error("Network error"));
    });
}
```

```
});  
req.send(options.body || null);  
}  


---


```

La richiesta iniziale visualizza le presentazioni ricevute sullo schermo e inizia il processo di long polling richiamando `waitForChanges`.

---

```
var lastServerTime = 0;  
request({pathname: "talks"}, function(error, response) {  
  if (error) {  
    reportError(error);  
  } else {  
    response = JSON.parse(response);  
    displayTalks(response.talks);  
    lastServerTime = response.serverTime;  
    waitForChanges();  
  }  
});  


---


```

La variabile `lastServerTime` serve per tener nota dell'orario dell'ultimo aggiornamento ricevuto dal server. Dopo la richiesta iniziale, la vista che il client ha delle presentazioni corrisponde alla vista che il server aveva al momento della risposta a quella richiesta. Pertanto, la proprietà `serverTime` inserita nella risposta fornisce un valore iniziale appropriato per `lastServerTime`.

Quando la richiesta non ha esito, non vogliamo che la pagina stia lì a far niente, senza spiegazioni. Definiamo pertanto una semplice funzione `reportError`, che mostra all'utente una finestra di dialogo che indica che qualcosa non ha funzionato.

---

```
function reportError(error) {  
  if (error)  
    alert(error.toString());  
}  


---


```

La funzione verifica se si tratta di un errore e produce la finestra solo se ne trova uno. In quel modo, possiamo anche passare direttamente questa funzione a `request` per le richieste di cui possiamo ignorare la risposta. Con questo, facciamo in modo che l'errore venga segnalato all'utente se la richiesta non ha successo.

## ***Visualizzazione delle presentazioni***

Per poter aggiornare la vista delle presentazioni quando arrivano degli aggiornamenti, il client deve seguire le presentazioni riportate correntemente. In quel modo, quando arriva

una nuova versione di una presentazione che già risulta sullo schermo, la presentazione può essere sostituita (sul posto) con la versione aggiornata. Analogamente, quando arriva la notizia che una presentazione è stata cancellata, può essere rimosso dal documento l'elemento DOM corrispondente.

La funzione `displayTalks` serve sia per costruire la visualizzazione iniziale, sia per gli aggiornamenti. Si serve dell'oggetto `shownTalks`, che associa i titoli delle presentazioni a nodi del DOM per tenere in memoria le presentazioni visualizzate sullo schermo.

---

```
var talkDiv = document.querySelector("#talks");
var shownTalks = Object.create(null);
function displayTalks(talks) {
  talks.forEach(function(talk) {
    var shown = shownTalks[talk.title];
    if (talk.deleted) {
      if (shown) {
        talkDiv.removeChild(shown);
        delete shownTalks[talk.title];
      }
    } else {
      var node = drawTalk(talk);
      if (shown)
        talkDiv.replaceChild(node, shown);
      else
        talkDiv.appendChild(node);
      shownTalks[talk.title] = node;
    }
  });
}
```

---

La struttura DOM per le presentazioni si costruisce con i modelli specificati nel documento HTML. Per prima cosa, dobbiamo definire `instantiate-Template`, che cerca e compila un modello.

Il parametro `name` è il nome del modello. Per trovare l'elemento corrispondente, cerchiamo un elemento il cui nome di classe corrisponda al nome del modello, che è un nodo figlio dell'elemento con ID "template". Il metodo `querySelector` semplifica il compito. La pagina HTML faceva riferimento ai modelli "talk" e "comment".

---

```
function instantiateTemplate(name, values) {
  function instantiateText(text) {
    return text.replace(/\{\{(\w+)\}\}/g, function(_, name) {
```

```

        return values[name];
    });
}

function instantiate(node) {
    if (node.nodeType == document.ELEMENT_NODE) {
        var copy = node.cloneNode();
        for (var i = 0; i < node.childNodes.length; i++)
            copy.appendChild(instantiate(node.childNodes[i]));
        return copy;
    } else if (node.nodeType == document.TEXT_NODE) {
        return document.createTextNode(
            instantiateText(node.nodeValue));
    } else {
        return node;
    }
}

var template = document.querySelector("#template ." + name);
return instantiate(template);
}

```

---

Il metodo `cloneNode`, che hanno tutti i nodi del DOM, crea una copia di un nodo, senza copiarne i nodi figli se non si passa `true` come primo argomento. La funzione `instantiate` costruisce ricorsivamente una copia del modello, riempiendolo man mano.

Il secondo argomento di `instantiateTemplate` dev'essere un oggetto, le cui proprietà mantengono le stringhe che bisogna riempire nel modello. Un marcatore come `{{title}}` verrà sostituito dal valore della proprietà `title` di `values`.

Si tratta di un sistema di modelli molto rudimentale ma sufficiente per `drawTalk`.

---

```

function drawTalk(talk) {
    var node = instantiateTemplate("talk", talk);
    var comments = node.querySelector(".comments");
    talk.comments.forEach(function(comment) {
        comments.appendChild(
            instantiateTemplate("comment", comment));
    });
    node.querySelector("button.del").addEventListener(
        "click", deleteTalk.bind(null, talk.title));
    var form = node.querySelector("form");

```

```
form.addEventListener("submit", function(event) {
  event.preventDefault();
  addComment(talk.title, form.elements.comment.value);
  form.reset();
});
return node;
}
```

---

Dopo l'istanziazione del modello "talk", dobbiamo sistemare diverse altre cose. Primo, dobbiamo riempire i commenti instanziando ripetutamente il modello "comment" e inserendo in coda i risultati al nodo con classe "comments". Poi, dobbiamo collegare dei gestori di eventi al pulsante che elimina la presentazione e al modulo che aggiunge un nuovo commento.

## **Aggiornamenti del server**

I gestori degli eventi registrati da drawTalk richiamano le funzioni deleteTalk e addComment per svolgere le operazioni necessarie per eliminare una presentazione o aggiungere un commento. Poiché queste funzioni dovranno costruire degli URL che facciano riferimento a presentazioni col titolo dato, definiamo una funzione ausiliaria talkURL.

---

```
function talkURL(title) {
  return "talks/" + encodeURIComponent(title);
}
```

---

Quando la funzione talkURL non ha successo, la funzione deleteTalk fa scattare una richiesta **DELETE** e riporta l'errore.

---

```
function deleteTalk(title) {
  request({pathname: talkURL(title), method: "DELETE"},
    reportError);
}
```

---

Per aggiungere un commento, dobbiamo costruirne una rappresentazione in formato JSON e trasmetterla nel corpo di una richiesta POST .

---

```
function addComment(title, comment) {
  var comment = {author: nameField.value, message: comment};
  request({pathname: talkURL(title) + "/comments",
    body: JSON.stringify(comment),
    method: "POST"}, reportError);
```

}

---

La variabile `nameField`, che imposta la proprietà `author` del commento, è un riferimento al campo `<input>` in cima alla pagina, che consente all'utente di specificare il proprio nome. Colleghiamo quel campo anche a `localStorage` per non doverlo ricompilare ogni volta che si ricarica la pagina.

---

```
var nameField = document.querySelector("#name");
nameField.value = localStorage.getItem("name") || "";
nameField.addEventListener("change", function() {
  localStorage.setItem("name", nameField.value);
});
```

---

Il modulo nella parte bassa della pagina, che serve per proporre una nuova presentazione, riceve un gestore di eventi "submit", che impedisce l'effetto predefinito (che ricaricherebbe la pagina), ripulisce il modulo e fa scattare una richiesta PUT per creare la presentazione.

---

```
var talkForm = document.querySelector("#newtalk");
talkForm.addEventListener("submit", function(event) {
  event.preventDefault();
  request({pathname: talkURL(talkForm.elements.title.value),
            method: "PUT",
            body: JSON.stringify({
              presenter: nameField.value,
              summary: talkForm.elements.summary.value
            })}, reportError);
  talkForm.reset();
});
```

---

## Rilevamento delle modifiche

Devo farvi notare a questo punto che le varie funzioni che modificano lo stato dell'applicazione, creando o eliminando presentazioni o aggiungendo commenti, non fanno nulla per assicurare che tali modifiche siano visibili sullo schermo. Le funzioni si limitano a informare il server e si affidano al meccanismo di long polling per far scattare gli aggiornamenti necessari sulla pagina.

Col meccanismo che abbiamo impostato nel server e il modo in cui abbiamo definito `displayTalks` per la gestione delle presentazioni già riportate sulla pagina, le operazioni di long polling sono sorprendentemente semplici.

---

```
function waitForChanges() {
    request({pathname: "talks?changesSince=" + lastServerTime},
        function(error, response) {
            if (error) {
                setTimeout(waitForChanges, 2500);
                console.error(error.stack);
            } else {
                response = JSON.parse(response);
                displayTalks(response.talks);
                lastServerTime = response.serverTime;
                waitForChanges();
            }
        });
}
```

---

Questa funzione viene richiamata una volta quando il programma si avvia; continua poi a richiamare se stessa per avere certezza che ci sia una richiesta di polling sempre attiva. Quando la richiesta non ha esito, non richiamiamo `reportError`, perché aprire una finestra di dialogo tutte le volte che non riusciamo a raggiungere il server è una seccatura, soprattutto quando il server è fermo. L'errore viene invece stampato sulla console (per facilitare il debugging) e si fa un altro tentativo dopo 2,5 secondi.

Quando la richiesta ha successo, i nuovi dati vengono aggiunti allo schermo e si aggiorna `lastServerTime` per riflettere il fatto che abbiamo ricevuto dei dati all'ora corrispondente. La richiesta viene riavviata immediatamente, in attesa dell'aggiornamento successivo.

Col programma del server in esecuzione, se aprite due finestre del browser e le fate puntare a `localhost:8000/`, vedrete che le azioni che svolgete in una finestra sono immediatamente visibili nell'altra.

## Esercizi

I seguenti esercizi richiedono che modifichiate il sistema definito in questo capitolo. Per poterli svolgere, dovete prima scaricare il codice (<http://eloquentjavascript.net/code/skillshare.zip>) e avere Node installato (<http://nodejs.org/>).

### Persistenza su disco

Il server definito per le presentazioni di skill-sharing mantiene i dati solo in memoria. Ciò significa che se si blocca o viene riavviato per qualunque motivo, tutte le presentazioni e i

commenti vanno persi.

Estendete il server in modo che memorizzi i dati su disco e li ricarichi automaticamente quando viene riavviato. Non preoccupatevi dell'efficienza: accontentatevi della soluzione più semplice che trovate.

## **Reimpostare i campi per i commenti**

Sostituire integralmente le presentazioni funziona bene, perché è difficile distinguere tra un nodo del DOM e un suo identico sostituto. Ma ci sono delle eccezioni. Se cominciate a scrivere qualcosa nel campo dei commenti in una finestra del browser e, in un'altra, aggiungete un commento, il campo della prima finestra viene ritracciato, perdendo focus e contenuto.

In una discussione animata, dove più persone aggiungono commenti alla stessa presentazione, questo comportamento sarebbe intollerabile. Riuscite a trovare un sistema per evitarlo?

## **Modelli migliori**

In genere, un sistema di modelli fa qualcosa di più che riempire qualche stringa vuota. Come minimo, consente delle inclusioni condizionali di parte del modello, analoghe a istruzioni `if`, e la ripetizione di parti del modello, in maniera analoga ai cicli.

Se potessimo sostituire una parte del modello per ciascun elemento di un array, non avremmo più bisogno del secondo modello ("comment"). Invece, potremmo specificare che il modello "talk" passi in ciclo sull'array mantenuto dalla proprietà `comments` di una presentazione e rendere i nodi che costituiscono un commento per ciascuno degli elementi dell'array.

Potrebbe essere una cosa come la seguente:

---

```
<div class="comments">
  <div class="comment" template-repeat="comments">
    <span class="name">{{author}}</span>: {{message}}
  </div>
</div>
```

---

L'idea è che ogni volta che si trova un nodo con un attributo `template-repeat` durante l'istanziazione del modello, il codice passi il ciclo sull'array mantenuto nella proprietà che ha il nome dell'attributo. Il codice di istanziazione aggiunge pertanto un'istanza del nodo per ciascun elemento dell'array. Durante il ciclo, il contesto del modello (la variabile `values` in `instantiateTemplate`) dovrebbe puntare all'elemento corrente dell'array in modo da andare a cercare `{{author}}` nell'oggetto `comment`, invece che nel contesto originario (la presentazione).

Riscrivete `instantiateTemplate` per riflettere questa impostazione e modificate il modello di conseguenza, eliminando la resa esplicita dei commenti effettuata dalla

funzione `drawTalk`.

Come potreste aggiungere delle istanziazioni condizionali di nodi, lasciando che alcune parti del modello siano omesse quando un determinato valore è `true` o `false`?

## ***Senza script***

Quando qualcuno visita il nostro sito Web con un browser che ha JavaScript disattivato, o semplicemente non è in grado di interpretare JavaScript, si trova davanti una pagina che non funziona. Il che non è carino.

Alcuni tipi di applicazioni Web non si possono proprio realizzare senza JavaScript. In altri casi, potreste non avere la pazienza o le risorse necessarie per occuparvi dei client che non sono in grado di eseguire script. Per pagine che hanno una vasta base di utenti, però, conviene prevedere del supporto anche per costoro.

Provate a trovare il modo di impostare il sito Web per le presentazioni di skill-sharing mantenendo delle funzionalità di base anche quando i visitatori non hanno JavaScript. Non potrete mantenere gli aggiornamenti automatici e gli utenti dovranno ricaricare la pagina col vecchio sistema. Sarebbe bello, però, se potessero vedere le presentazioni esistenti, crearne di nuove e inviare commenti.

Non sentitevi obbligati a realizzare questa soluzione in codice: è sufficiente che la descriviate. Come vi sembra la nuova impostazione rispetto a quel che abbiamo fatto all'inizio: più o meno elegante?

## JAVASCRIPT E PRESTAZIONI

Per eseguire un programma su un computer, bisogna risolvere il conflitto tra il linguaggio di programmazione e il formato delle istruzioni proprio della macchina. Questo si può fare scrivendo un programma che *interpreta* altri programmi, come abbiamo visto nel [Capitolo 11](#), ma di solito si preferisce *compilare* (tradurre) il programma in codice macchina.

Alcuni linguaggi di programmazione, come il C, sono progettati per esprimere esattamente quel che il computer sa far meglio: risultano pertanto più facili da compilare con efficienza. JavaScript, però, è stato progettato in modo diverso, pensando soprattutto alla semplicità del codice e alla facilità d'uso, e ben poche delle sue funzionalità corrispondono direttamente a quelle della macchina. JavaScript risulta pertanto difficile da eseguire in modo efficiente.

Eppure, i *motori* JavaScript più moderni (i programmi che compilano ed eseguono il codice JavaScript) riescono a eseguire gli script a velocità sorprendenti. È possibile scrivere programmi JavaScript che sono meno di 10 volte più lenti di programmi equivalenti scritti in C. Se vi sembra una differenza enorme, considerate che i motori JavaScript più vecchi (così come le versioni contemporanee di linguaggi simili, come Python e Ruby) erano molto più lenti, addirittura 100 volte più lenti del C. In confronto, le ultime versioni di JavaScript sono velocissime, tanto che vi capiterà raramente di dover scegliere un altro linguaggio solo per problemi di prestazioni.

Vi può tuttavia capitare di dover riscrivere del codice per evitare gli aspetti di JavaScript che rimangono lenti. Per darvi un esempio, in questo capitolo prendiamo un programma lento e lo modifichiamo per renderlo più veloce. Strada facendo, discuteremo il modo in cui i motori JavaScript compilano i vostri programmi.

### Compilazione per gradi

Per prima cosa, dovete capire che i compilatori per JavaScript non si limitano a compilare il programma una volta sola, come fanno i compilatori classici per linguaggi come il C. Invece, il codice è compilato e ricompilato quando serve, intanto che il programma è in esecuzione.

Tradizionalmente, compilare un programma di grandi dimensioni richiede tempo. Il che non è mai stato un problema, poiché con i linguaggi tradizionali i programmi vengono compilati in anticipo e distribuiti in forma già compilata.

Per JavaScript la situazione è diversa. Un sito Web può comprendere grandi quantità di codice, che viene recuperato in forma testuale e deve essere compilato ogni volta che si apre il sito. Se ci vogliono cinque minuti per farlo, gli utenti si seccano. Un compilatore JavaScript deve pertanto essere in grado di iniziare a eseguire il programma, non importa quanto grande, quasi istantaneamente.

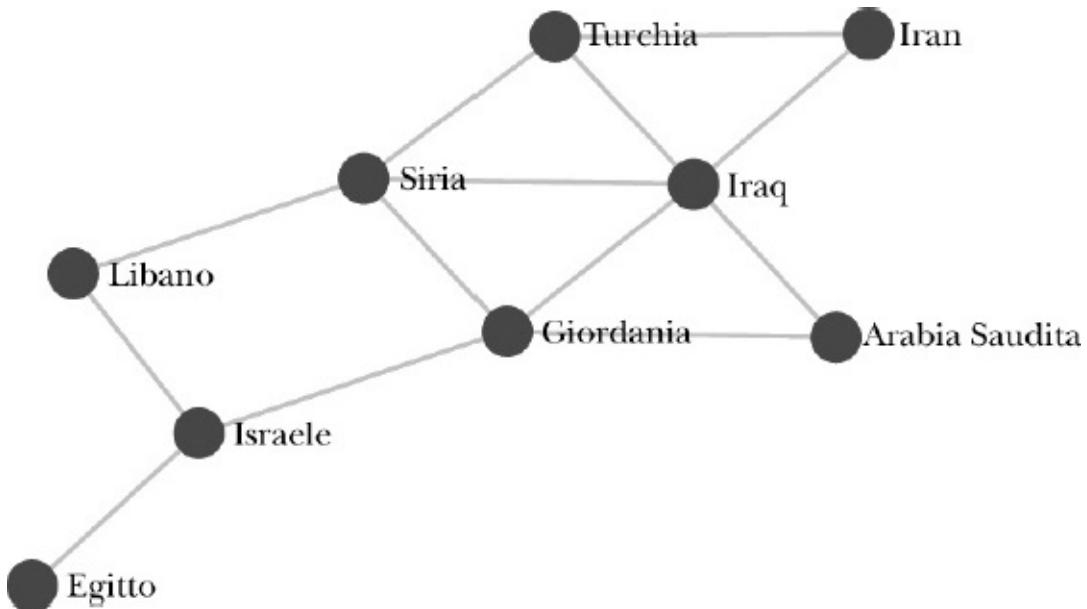
Per far questo, i compilatori per JavaScript utilizzano diverse strategie di compilazione. Quando si apre un sito Web, gli script vengono compilati sommariamente, in modo superficiale. L'esecuzione non è rapida, ma parte subito. In effetti, alcuni motori JavaScript non compilano le funzioni finché non vengono richiamate. Ogni browser ha un suo motore JavaScript e ciascun motore ha le sue strategie.

In un programma tipico, gran parte del codice viene eseguito solo poche volte e parte di esso non viene eseguito per niente. Per queste parti del programma, la strategia di compilazione sommaria è sufficiente, perché non richiede troppo tempo. Le funzioni richiamate più spesso, però, o quelle che contengono cicli che svolgono molto lavoro, vanno trattate diversamente. Durante l'esecuzione del programma, il motore JavaScript prende nota di quanto spesso sono eseguiti i vari brani del programma. Quando uno di essi sembra richiedere molto tempo (in gergo, questi brani di programma si chiamano *hot code* o codice caldo), la funzione viene ricompilata con un compilatore più avanzato, ma più lento del primo. Questo secondo compilatore svolge più operazioni di ottimizzazione e produce pertanto codice più veloce. Alcuni motori hanno persino un terzo compilatore, ancora più avanzato e più lento, per brani di codice molto caldi.

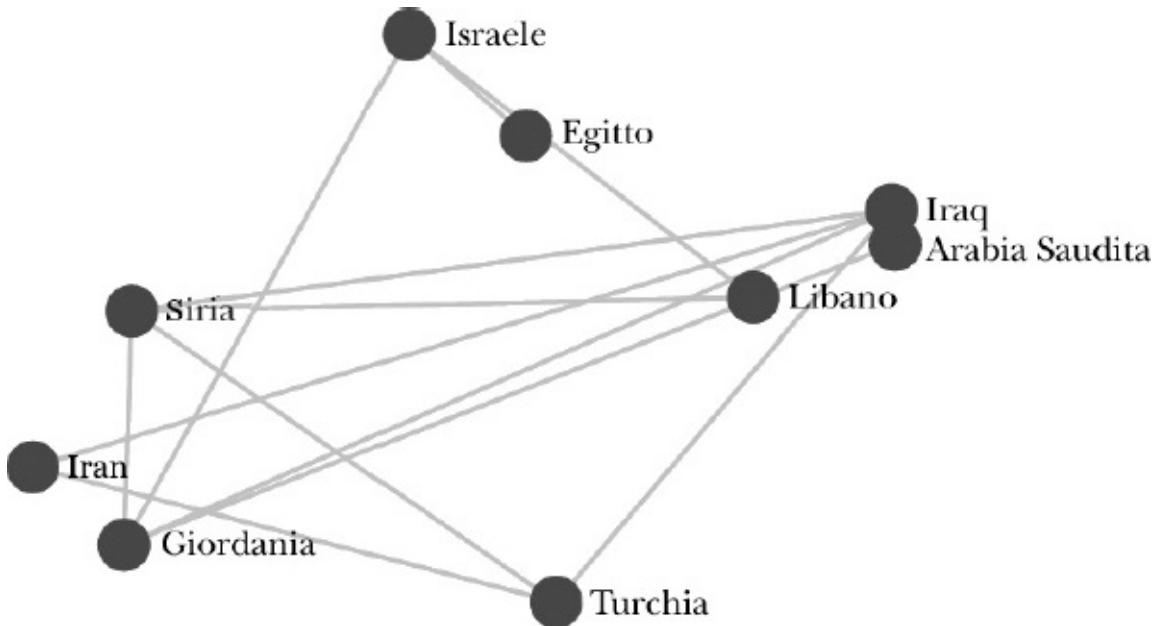
Questo intreccio di esecuzione e compilazione significa che prima che il compilatore avanzato cominci a lavorare con un brano di codice, il programma è già stato eseguito più volte. Questo rende possibile osservare il codice in esecuzione e raccogliere informazioni su di esso. Più avanti nel capitolo vedremo come questo può consentire al compilatore di creare codice più efficiente.

## Layout di un grafo

Il nostro problema di esempio riguarda i grafi. Un *grafo* è una serie di punti (nodi) che hanno tra loro delle connessioni, dette *archi* (edges). Può essere usato per rappresentare reti di strade, alberi genealogici, la distribuzione del flusso di controllo in un programma per computer e così via. La figura che segue mostra un grafo che rappresenta alcuni paesi del Medioriente, con connessioni che collegano quelli che hanno un confine terrestre in comune:



Il lavoro di costruire un'immagine come questa partendo dalla definizione del grafo si chiama *layout del grafo*. Pianificare il layout comporta di assegnare una posizione per ciascun nodo, in modo che i nodi connessi tra loro siano contigui, pur lasciando abbastanza spazio tra uno e l'altro. Un layout non strutturato dello stesso grafo è molto più difficile da interpretare:



Trovare un layout di bell'aspetto per un grafo è un problema ben noto e piuttosto difficile. Per i grafi più grandi, con moltissime connessioni, non esiste una singola soluzione che funzioni in tutti i casi; esistono tuttavia delle soluzioni per grafi di forme specifiche.

Per tracciare un piccolo grafo (fino a 100 nodi) che non sia troppo complicato, possiamo seguire l'impostazione di layout *basata sulla forza*. In questo approccio, si esegue una piccola simulazione fisica sui nodi, trattando le connessioni tra nodi come se fossero molle, mentre i nodi si respingono come se fossero poli elettrici carichi.

In questo capitolo, vedremo come realizzare un layout basato sulla forza e ne osserveremo le prestazioni. Possiamo eseguire la simulazione fisica calcolando ripetutamente le forze che agiscono su ciascun nodo e spostando i nodi in risposta alle

spinte. In un programma come questo, le prestazioni sono importanti perché arrivare a un layout stabile e di bell’aspetto può richiedere moltissime iterazioni, e ciascuna iterazione deve calcolare un gran numero di forze.

## Definizione di un grafo

Possiamo rappresentare un grafo come un array di oggetti `GraphNode`, ciascuno dei quali mantiene la propria posizione e un array degli altri nodi di cui condivide gli archi. Le posizioni iniziali dei nodi sono calcolate a caso.

---

```
function GraphNode() {
    this.pos = new Vector(Math.random() * 1000,
                          Math.random() * 1000);
    this.edges = [];
}

GraphNode.prototype.connect = function(other) {
    this.edges.push(other);
    other.edges.push(this);
};

GraphNode.prototype.hasEdge = function(other) {
    for (var i = 0; i < this.edges.length; i++)
        if (this.edges[i] == other)
            return true;
};
```

---

Per rappresentare posizioni e forze, ci serviamo del tipo `Vector`, già visto nei capitoli precedenti.

I metodi `connect` e `hasEdge` offrono un modo per connettere un nodo a un altro nella rappresentazione del grafo e per verificare se i due nodi sono connessi.

La funzione `treeGraph` costruisce un semplice grafo, sul quale metteremo alla prova il programma di layout. Accetta due parametri, che specificano la profondità dell’albero e il numero di rami da creare a ciascun livello, per costruire ricorsivamente un grafo ad albero con la forma specificata.

---

```
function treeGraph(depth, branches) {
    var graph = [];
    function buildNode(depth) {
        var node = new GraphNode();
        graph.push(node);
        if (depth > 1)
```

```

        for (var i = 0; i < branches; i++)
            node.connect(buildNode(depth - 1));
    return node;
}
buildNode(depth);
return graph;
}

```

---

I grafi ad albero non contengono cicli; risultano pertanto relativamente facili da rappresentare e persino il programmino semplice che definiamo in questo capitolo riesce a ottenere forme gradevoli.

Il grafo creato da `treeGraph(3, 5)` è un albero di profondità 3 con 5 rami.



Perché possiate esaminare i layout prodotti dal codice di questo capitolo, ho definito una funzione `drawGraph` che disegna il grafo su un elemento Canvas. La funzione è definita nel codice che trovate su [http://eloquentjavascript.net/code/draw\\_graph.js](http://eloquentjavascript.net/code/draw_graph.js) ed è disponibile nello spazio protetto (sandbox) online.

## Una prima funzione di layout basata sulla forza

Sposteremo i nodi uno per volta, calcolando le forze che agiscono sul nodo corrente e spostandolo immediatamente in base a esse.

La forza applicata da una molla (virtuale) si può calcolare approssimativamente con la legge di Hooke, che dice che tale forza è proporzionale alla differenza tra la lunghezza della molla a riposo e la lunghezza della molla all'estensione attuale. La variabile `springLength` definisce la lunghezza a riposo delle molle per le connessioni; la variabile `springStrength` definisce una costante di rigidità della molla, che moltiplicheremo per la lunghezza per ricavare la forza risultante.

---

```
var springLength = 40;
var springStrength = 0.1;
```

---

Per simulare la repulsione tra nodi, ci serviamo di un'altra legge fisica, quella di Coulomb, che dice che la forza di repulsione tra due particelle (dello stesso segno) caricate elettricamente è inversamente proporzionale al quadrato della distanza che le separa. Quando due nodi sono quasi sovrapposti, il quadrato della distanza è minimo e la forza di repulsione risultante sarà pertanto altissima. Man mano che i nodi si allontanano, il quadrato della distanza aumenta esponenzialmente e la forza repulsiva diminuisce rapidamente.

Moltiplicheremo anche questa volta i valori per una certa costante, `repulsionStrength`, che definisce la forza repulsiva tra nodi.

---

```
var repulsionStrength = 1500;
```

---

La forza che agisce su un dato nodo si calcola passando in ciclo su tutti gli altri nodi e applicando la forza repulsiva per ciascuno di essi. Quando un altro nodo condivide una connessione col nodo corrente, viene applicata anche la forza della molla.

Entrambe le forze dipendono dalla distanza tra i due nodi. La funzione calcola un vettore, `apart`, che rappresenta il percorso dal nodo corrente all'altro nodo. La funzione prende poi la lunghezza del vettore per trovare la distanza effettiva. Quando la distanza è meno di uno, la impostiamo su uno per evitare divisioni per zero o numeri troppo piccoli, che produrrebbero valori `NaN` o forze tanto colossali da catapultare il nodo nello spazio siderale.

Con la distanza così ottenuta, possiamo calcolare la grandezza della forza che agisce tra i due nodi dati. Per passare da una grandezza a un vettore di forza, dobbiamo moltiplicare la grandezza per una versione normalizzata del vettore `apart`. Normalizzare un vettore significa crearne uno con la stessa direzione ma con una lunghezza di uno, cosa che si può fare semplicemente dividendo il vettore per la sua lunghezza.

---

```
function forceDirected_simple(graph) {
  graph.forEach(function(node) {
    graph.forEach(function(other) {
      if (other == node) return;
      var apart = other.pos.minus(node.pos);
      var distance = Math.max(1, apart.length);
      var forceSize = -repulsionStrength / (distance * distance);
      if (node.hasEdge(other))
        forceSize += (distance - springLength) * springStrength;
      var normalized = apart.times(1 / distance);
      node.pos = node.pos.plus(normalized.times(forceSize));
    });
  });
}
```

```
});  
});  
}  
}
```

---

Per provare una realizzazione del nostro sistema di layout di grafi, usiamo la funzione `runLayout`, che ripete il modello 4000 volte (`steps`) e prende nota del tempo che ci vuole. Per darci qualcosa da vedere intanto che esegue i calcoli, la funzione ritraccia il layout corrente ogni 100 passi.

---

```
function runLayout(implementation, graph) {  
    var totalSteps = 0, time = 0;  
    function step() {  
        var startTime = Date.now();  
        for (var i = 0; i < 100; i++)  
            implementation(graph);  
        totalSteps += 100;  
        time += Date.now() - startTime;  
        drawGraph(graph);  
        if (totalSteps < 4000)  
            requestAnimationFrame(step);  
        else  
            console.log(time);  
    }  
    step();  
}
```

---

Possiamo ora eseguire il nostro primo programma e vedere quanto tempo ci vuole.

---

```
<script>  
    runLayout(forceDirected_simple, treeGraph(4, 4));  
</script>
```

---

Sul mio computer, con la versione 38 di Chrome, le 4000 iterazioni hanno richiesto un po' più di quattro secondi. Che è un sacco di tempo. Vediamo se si può far meglio.

## Analisi dinamica del programma

Magari avete già un'idea del perché il nostro primo programma è lento. Ma i problemi di prestazione possono arrivare da direzioni inattese e se cominciate a modificare il codice nella *presunzione* che la vostra idea lo farà girare più velocemente, rischiate di perder tempo inutilmente con modifiche che in realtà non aiutano.

La nostra funzione `runLayout` misura il tempo richiesto dal programma. Il che è già un buon inizio. Per migliorare qualcosa, dovete prima misurarla per sapere se le modifiche che vi apporterete avranno un effetto positivo.

Gli strumenti per sviluppatori dei browser moderni offrono un sistema anche migliore per misurare il tempo richiesto dal programma. Si tratta dello strumento *profiler*. Mentre il programma è in esecuzione, lo strumento di analisi raccoglie informazioni sul tempo richiesto dalle varie parti del programma.

Se il vostro browser dispone di uno strumento di profilazione, lo trovate nell'interfaccia degli strumenti di sviluppo, magari come scheda della barra superiore. Il profiler di Google Chrome produce una tabella a tre colonne per il nostro programma.

---

Self	Total	Function
3787.5 ms 47.48 %	5015.4 ms 62.87 %	(anonymous function)
2528.3 ms 31.70 %	7545.7 ms 94.60 %	(anonymous function)
1229.9 ms 15.42 %	1229.9 ms 15.42 %	Vector
188.4 ms 2.36 %	188.4 ms 2.36 %	(garbage collector)
44.3 ms 0.56 %	7590.1 ms 95.15 %	forceDirected_simple
39.3 ms 0.49 %	39.3 ms 0.49 %	stroke
22.2 ms 0.28 %	22.2 ms 0.28 %	fill
11.1 ms 0.14 %	11.1 ms 0.14 %	clearRect
6.0 ms 0.08 %	6.0 ms 0.08 %	arc
4.0 ms 0.05 %	88.6 ms 1.11 %	drawGraph
2.0 ms 0.03 %	7680.7 ms 96.29 %	step

---

I numeri riportano il tempo impiegato per l'esecuzione di una data funzione, sia in millisecondi, sia come percentuale del tempo impiegato in totale. La prima colonna riporta solo il tempo di permanenza del controllo (di flusso) nella funzione, mentre la seconda colonna comprende anche il tempo di permanenza nelle funzioni richiamate dalla funzione.

Da questi dati, rileviamo che il tempo richiesto per disegnare il grafo è minuscolo rispetto al tempo necessario per la simulazione. Nel browser, possiamo fare clic sulla definizione di ogni funzione per ulteriori informazioni. Con questo, ho potuto verificare che le due funzioni anonime nelle prime due righe del profilo sono le funzioni passate a `forEach` in `forceDirected_simple`. Come potevamo immaginare, il calcolo delle forze è l'operazione che richiede più tempo.

La riga contrassegnata da `(garbage collector)` indica il tempo trascorso per ripulire la memoria non più utilizzata. Dato che il nostro programma crea un gran numero di oggetti vettore, il 2,36% del tempo passato per ripulire la memoria è straordinariamente piccolo. Bisogna però dire che Chrome ha un `garbage collector` molto efficiente. Notate però che più del 15% del tempo è servito per creare oggetti vettore. Qui c'è spazio per del lavoro di ottimizzazione, come vedremo più avanti. Prima, però, esaminiamo un'altra fonte di rallentamenti.

## Funzioni in linea

Nessuno dei metodi del vettore, come `plus`, viene riportato nel profilo: eppure, vengono usati pesantemente. La ragione di ciò è che il motore di JavaScript li ha *messi in linea*. Ciò significa che, invece di lasciare che il codice della funzione interna richiami un metodo per aggiungere vettori, il codice per quest'ultima operazione viene inserito direttamente all'interno della funzione e nel codice compilato non si effettuano chiamate ai metodi.

Sono diversi i modi in cui l'inserimento in linea aumenta la velocità del codice. Funzioni e metodi, a livello macchina, si richiamano attraverso un protocollo che richiede di inserire gli argomenti e l'indirizzo di restituzione (il punto dove l'esecuzione deve continuare quando la funzione restituisce) da qualche parte dove la funzione li possa trovare. Anche il modo in cui una chiamata di funzione passa il controllo a un'altra parte del programma richiede spesso di salvare parte dello stato del processore, in modo che la funzione richiamata possa usare il processore senza interferire con i dati di cui la funzione chiamante ha ancora bisogno. Tutto ciò non è più necessario quando le funzioni sono in linea.

Inoltre, un buon compilatore cerca sempre il modo per semplificare il codice. Se le funzioni fossero trattate come scatole nere in grado di fare qualunque cosa, il compilatore avrà ben poco materiale su cui lavorare. D'altro canto, se può vedere e inserire il corpo della funzione nella sua analisi, può trovare altre opportunità per ottimizzare il codice risultante.

Per esempio, un motore JavaScript potrebbe evitare di creare alcuni degli oggetti vettore del codice. In un'espressione come la seguente, se riusciamo a guardare al di là dei metodi, è chiaro che le coordinate del vettore sono il risultato della somma delle coordinate della forza al prodotto delle coordinate normalizzate (`normalized`) per la variabile `forceSize`. Pertanto, non c'è bisogno di creare l'oggetto intermedio prodotto dal metodo `times`.

---

```
force.plus(normalized.times(forceSize))
```

---

JavaScript, però, è un linguaggio dinamico. Come fa il compilatore a sapere qual è la funzione del metodo `plus`? E che cosa succede se qualcuno, in un secondo tempo, cambia il valore memorizzato in `Vector.prototype`? La volta successiva, dopo che il codice ha messo in linea l'esecuzione delle funzioni, può continuare a usare la vecchia definizione violando i presupposti del programmatore sul comportamento del programma.

Ecco dove l'alternanza tra esecuzione e compilazione comincia a dare i suoi frutti. Quando viene compilata, una funzione calda è già stata eseguita diverse volte. Se, in quei cicli di esecuzione, ha sempre richiamato la stessa funzione, ha senso cercare di metterla in linea. Il codice viene ottimisticamente compilato immaginando che, in futuro, si richiamerà la stessa funzione nello stesso posto.

Per gli altri casi, dove si richiama invece un'altra funzione, il compilatore inserisce una prova, dove si confronta la funzione richiamata con quella in linea. Se non si tratta della stessa funzione, il codice compilato in modo ottimistico è inutile e il motore

JavaScript deve *de-ottimizzare*, il che significa che torna a una versione meno ottimizzata del codice.

## Ritorno ai cicli di vecchia memoria

Il fatto che in cima al profilo compaiano due funzioni anonime passate a `forEach` nella nostra funzione di simulazione indica che quelle due funzioni non erano in linea. Si tratta di funzioni che vengono richiamate molto spesso e vorremmo che il compilatore ottimizzasse il corpo della funzione di simulazione il meglio possibile.

Nel momento in cui scrivo, nessun motore JavaScript sembra in grado di mettere in linea chiamate `forEach`. Ciò significa che si deve creare per ogni chiamata un oggetto funzione, che le variabili contemplate (`graph` e `node`) devono essere messe in una partizione speciale condivisa della memoria e che avvengono un sacco di chiamate di funzioni.

La soluzione è semplice. Basta usare un ciclo `for` tradizionale. Il codice risulta meno gradevole da vedere, ma sono disposto a pagarne il prezzo per i miei cicli interni.

---

```
function forceDirected_forloop(graph) {
    for (var i = 0; i < graph.length; i++) {
        var node = graph[i];
        for (var j = 0; j < graph.length; j++) {
            if (i == j) continue;
            var other = graph[j];
            var apart = other.pos.minus(node.pos);
            var distance = Math.max(1, apart.length);
            var forceSize = -1 * repulsionStrength / (distance * distance);
            if (node.hasEdge(other))
                forceSize += (distance - springLength) * springStrength;
            var normalized = apart.times(1 / distance);
            node.pos = node.pos.plus(normalized.times(forceSize));
        }
    }
}
```

---

Con questo, il codice diventa circa il 30% più veloce sia in Chrome 38, sia in Firefox 32. Se ripeto l'analisi nel browser, la funzione `forceDirected_forloop` assorbe gran parte del tempo. Interessante notare come sparisca completamente dal profilo il costruttore `vector`, cosa che suggerisce che il motore ha evitato di creare vettori oppure ha messo in linea il costruttore.

## Evitare lavoro inutile

Potete rendere più veloce il codice trovando un modo meno dispendioso per fare lo stesso lavoro, come abbiamo appena fatto. A volte, invece, potete anche trovare il modo di evitare completamente del lavoro. Ci sono situazioni dove questo può migliorare di molto le prestazioni, senza bisogno di operazioni di micro-ottimizzazione come sostituire i cicli `forEach` con cicli `for`.

Nel nostro progetto di esempio, c'è una (piccola) opportunità per svolgere meno lavoro. Per ogni coppia di nodi, le forze in gioco vengono calcolate due volte: una quando si sposta il primo nodo, una quando si sposta il secondo. Poiché la forza che il nodo *X* esercita sul nodo *Y* è uguale e inversa a quella esercitata da *Y* su *X*, non abbiamo bisogno di ripetere il calcolo due volte.

La prossima versione della funzione modifica pertanto il ciclo interno in modo che passi solo sui nodi che vengono dopo quello corrente, in modo che ogni coppia sia esaminata una sola volta. Alla fine del ciclo, la funzione aggiorna la posizione di tutti e due i nodi.

---

```
function forceDirected_norepeat(graph) {
  for (var i = 0; i < graph.length; i++) {
    var node = graph[i];
    for (var j = i + 1; j < graph.length; j++) {
      var other = graph[j];
      var apart = other.pos.minus(node.pos);
      var distance = Math.max(1, apart.length);
      var forceSize = -1 * repulsionStrength / (distance * distance);
      if (node.hasEdge(other))
        forceSize += (distance - springLength) * springStrength;
      var applied = apart.times(forceSize / distance);
      node.pos = node.pos.plus(applied);
      other.pos = other.pos.minus(applied);
    }
  }
}
```

---

Analizzando questo codice, noto un altro grosso miglioramento rispetto alla versione precedente: risparmia più del 40% del tempo su Chrome 38 e circa il 30% su Firefox 32 e Internet Explorer 11.

## Creare meno rifiuti

Sebbene alcuni degli oggetti vettore che stiamo usando per svolgere operazioni di aritmetica bidimensionale possano essere ottimizzati al massimo da alcuni motori, ci sono comunque dei costi associati con la creazione di tutti questi oggetti. Per avere un'idea delle dimensioni del costo, scriviamo una versione del codice che svolge i calcoli vettoriali “a mano”, usando variabili locali per entrambe le dimensioni.

---

```
function forceDirected_novector(graph) {
    for (var i = 0; i < graph.length; i++) {
        var node = graph[i];
        for (var j = i + 1; j < graph.length; j++) {
            var other = graph[j];
            var apartX = other.pos.x - node.pos.x;
            var apartY = other.pos.y - node.pos.y;
            var distance = Math.max(1, Math.sqrt(apartX * apartX + apartY *
apartY));
            var forceSize = -repulsionStrength / (distance * distance);
            if (node.hasEdge(other))
                forceSize += (distance - springLength) * springStrength;
            var forceX = apartX * forceSize / distance;
            var forceY = apartY * forceSize / distance;
            node.pos.x += forceX; node.pos.y += forceY;
            other.pos.x -= forceX; other.pos.y -= forceY;
        }
    }
}
```

---

Il nuovo listato è più verboso e ripetitivo, ma analizzandolo nel browser, i miglioramenti sono tali da giustificare un certo appiattimento degli oggetti nel codice che dà problemi di prestazioni. Chrome 38 fa un buon lavoro nell'ottimizzare la creazione di oggetti e il nuovo codice produce un aumento di velocità intorno al 40% rispetto alla versione precedente. Su Firefox 32 arriviamo a oltre il 60%, mentre su Internet Explorer 11, questa ottimizzazione manuale rende il codice più veloce dell’80%.

È pertanto chiaro che la velocità di esecuzione di un brano di JavaScript dipende in gran parte dal motore che lo muove. Proprio perché è per sua natura difficile da compilare, JavaScript richiede dei compilatori estremamente complicati. Questa complessità li rende imprevedibili. Piccole modifiche nel codice, il modo in cui viene utilizzato e la versione del motore su cui viene eseguito possono tutti causare grosse fluttuazioni nelle sue prestazioni.

## La raccolta dei rifiuti

Perché gli oggetti sono costosi (in termini di prestazioni)? Ci sono due ragioni. La prima è che il motore deve trovare un posto dove memorizzarli. La seconda che deve sapere quando non servono più e a quel punto liberarsene. Entrambe le cose sono complicate.

Ripensate alla memoria come a una lunga serie di bit. Quando il programma si avvia, riceve un blocco vuoto di memoria e può cominciare a metterci dentro gli oggetti creati man mano, uno dopo l'altro. A un certo punto, però, non c'è più spazio e alcuni degli oggetti che lo occupano non servono più al programma. Il motore di JavaScript deve capire quali oggetti sono in uso e quali non lo sono; potrà poi contrassegnare gli spazi non usati della memoria e riutilizzarli di nuovo.

Lo spazio di memoria del programma però è sempre una bella confusione, con oggetti attivi mescolati a spazio vuoto. Creare un nuovo oggetto richiede per prima cosa di trovare uno spazio abbastanza grande per l'oggetto e per questo ci vuole del tempo. In alternativa, il motore può spostare tutti gli oggetti attivi all'inizio dello spazio di memoria e risparmiare tempo quando poi deve creare nuovi oggetti (che potrà continuare ad aggiungere uno dopo l'altro), ma questo richiede più lavoro.

In linea di massima, per trovare gli oggetti in uso bisogna andare a esaminare tutti gli oggetti raggiungibili, partendo dall'ambito globale e da quello locale che risulta attivo. Tutti gli oggetti referenziati da quei due ambiti, direttamente o indirettamente, sono ancora attivi. Se il programma ha in memoria molti dati, questo può comportare davvero un sacco di lavoro.

Una tecnica chiamata *raccolta dei rifiuti (garbage collection) generazionale* può aiutare a ridurre questi costi. Chrome utilizza questa tecnica, che è probabilmente la ragione per cui eliminare gli oggetti vettori intermedi aveva fatto solo una piccola differenza in quel browser rispetto agli altri. Nel momento in cui scrivo, anche Firefox si sta muovendo in questa direzione.

La raccolta generazionale sfrutta il fatto che gli oggetti hanno per lo più vita breve. Il sistema divide la memoria disponibile per il programma in due o più *generazioni*. I nuovi oggetti vengono creati nello spazio riservato per l'ultima generazione. Quando questo spazio è pieno, il motore esamina quali degli oggetti che vi si trovano sono ancora attivi e li sposta a una generazione successiva. Se nell'ultima generazione rimane attiva solo una piccola parte degli oggetti quando ciò avviene, il lavoro per spostarli è minimo.

Naturalmente, per calcolare quali degli oggetti sono ancora attivi, il motore deve conoscere tutti i riferimenti agli oggetti nella generazione vivente. Non ha senso che il motore vada a esaminare gli oggetti nelle generazioni precedenti ogni volta che raccoglie quelli dell'ultima. Per questo, quando si crea un riferimento da un oggetto vecchio a uno nuovo, bisogna registrarla per tenerne conto per la prossima raccolta. Per questo, scrivere su oggetti vecchi è un po' più costoso, ma quel costo è più che compensato dal tempo risparmiato durante la raccolta dei rifiuti.

## Scrivere sugli oggetti

Visto che scrivere sugli oggetti vecchi può comportare un costo, probabilmente vi preoccupa il modo in cui l'ultima versione del nostro programma applica le forze. Il ciclo interno aggiorna le proprietà  $x$  e  $y$  degli oggetti con le posizioni dei nodi con la vita più lunga. Per un grafo con  $N$  nodi, il ciclo interno viene eseguito  $N^2$  volte. Nel nostro grafo di prova, che ha 85 nodi, ciò significa 7225 volte per ogni iterazione della simulazione.

Meglio allora calcolare le forze localmente e aggiornare le posizioni dei nodi solo dopo aver calcolato tutte le forze. Per questo ci vuole del codice in più, per costruire gli array che mantengono le forze e per applicare quelle forze alla fine della funzione.

---

```
function forceDirected_localforce(graph) {
    var forcesX = [], forcesY = [];
    for (var i = 0; i < graph.length; i++) {
        forcesX[i] = forcesY[i] = 0;
        for (var j = i + 1; j < graph.length; j++) {
            var node = graph[i];
            for (var other = graph[j]; other != node; other = graph[j + 1]) {
                var apartX = other.pos.x - node.pos.x;
                var apartY = other.pos.y - node.pos.y;
                var distance = Math.max(1, Math.sqrt(apartX * apartX + apartY * apartY));
                var forceSize = -repulsionStrength / (distance * distance);
                if (node.hasEdge(other))
                    forceSize += (distance - springLength) * springStrength;
                var forceX = apartX * forceSize / distance;
                var forceY = apartY * forceSize / distance;
                forcesX[i] += forceX; forcesY[i] += forceY;
                forcesX[j] -= forceX; forcesY[j] -= forceY;
            }
        }
        for (var i = 0; i < graph.length; i++) {
            graph[i].pos.x += forcesX[i];
            graph[i].pos.y += forcesY[i];
        }
    }
}
```

---

Raggruppare gli aggiornamenti delle posizioni in questo modo migliora la velocità del

programma di un altro 20%. È interessante notare che questo aumento delle prestazioni è costante su tutti e tre i browser di prova, anche se due di essi in teoria non usano la tecnica generazionale. È più probabile che ciò dipenda da come effettuano la raccolta dei rifiuti in modo *incrementale*, in quanto suddivide il lavoro di analisi della memoria in blocchi più piccoli, invece di svolgerlo tutto in una volta; cosa quest'ultima che potrebbe causare delle pause rilevabili nell'esecuzione del programma. Questa tecnica richiede anche di tener d'occhio i nuovi riferimenti.

Passo per passo, abbiamo trasformato il nostro programmino, rendendolo meno bello da vedere ma più veloce di circa l'85% (l'ultima versione impiega circa il 15% del tempo della prima).

Voglio incoraggiarvi ancora una volta a non applicare queste tecniche a tutto il codice che scrivete. In molti casi, la differenza non si noterebbe. Solo il codice in cicli interni che viene eseguito *molto* spesso può trarre vantaggio da questo tipo di regolazioni.

## Tipi dinamici

Le espressioni JavaScript come `node.pos`, che recuperano una proprietà da un oggetto, sono tutt'altro che semplici da compilare. In molti linguaggi, le variabili hanno un tipo; pertanto, quando svolgete un'operazione sul valore che mantengono, il compilatore sa già che tipo di operazione vi serve. In JavaScript, solo i valori hanno tipi; una variabile può pertanto mantenere valori di tipi diversi.

Ciò significa che inizialmente il compilatore sa ben poco delle proprietà a cui il codice cerca di accedere e deve produrre codice in grado di gestire tutti i tipi possibili. Se `node` mantiene un valore `undefined`, il codice deve lanciare un errore. Se si tratta di una stringa, deve trovare `pos` in `String.prototype`; se è un oggetto, il modo per estrarre la proprietà `pos` dipende dal tipo di oggetto e così via.

Per fortuna, sebbene JavaScript non lo richieda, in quasi tutti i programmi le variabili hanno un solo tipo. E se il compilatore lo conosce, può usare queste informazioni per creare del codice più efficiente. Se `node` è sempre stato (finora) un oggetto con proprietà `pos` ed `edge`, il programma di ottimizzazione del compilatore può semplicemente creare del codice che recuperi la proprietà dalla sua posizione nota in quest'oggetto, cosa che risulta semplice e veloce.

Gli eventi osservati in passato, però, non danno alcuna garanzia sugli eventi che avranno luogo in futuro. I brani di codice che non sono ancora stati eseguiti potrebbero passare un altro tipo di valore alla nostra funzione: un diverso tipo di oggetto `node`, per esempio, che ha anche una proprietà `id`.

Pertanto, il codice compilato deve ogni volta verificare se valgono sempre gli stessi presupposti e scegliere un'azione appropriata in caso contrario. Alcuni motori potrebbero de-ottimizzare completamente, tornando alle versioni non ottimizzate della funzione. O potrebbero compilare una nuova versione della funzione, in grado di gestire anche il nuovo tipo.

Potete seguire il rallentamento causato dall'incapacità di prevedere i tipi degli oggetti scombinando di proposito l'uniformità degli oggetti di input per la nostra funzione di layout di grafi, come nell'esempio seguente:

---

```
var mangledGraph = treeGraph(4, 4);
mangledGraph.forEach(function(node) {
  var letter = Math.floor(Math.random() * 26);
  node[String.fromCharCode("A".charCodeAt(0) + letter)] = true;
});
runLayout(forceDirected_localforce, mangledGraph);
```

---

Ogni nodo riceve una proprietà in più, il cui nome è una lettera maiuscola a caso. La lettera viene calcolata prendendo il codice del carattere *A* e aggiungendovi un numero casuale.

Se eseguiamo la versione veloce del codice di simulazione sul grafo risultante, l'esecuzione diventa tre volte più lenta su Chrome 38 e nove (!) volte più lenta su Firefox 32. Adesso che i tipi degli oggetti non sono uniformi, il codice deve esaminare le proprietà senza conoscenze a priori sulla forma dell'oggetto, cosa che risulta molto più dispendiosa.

Una tecnica simile si usa anche per aspetti diversi dall'accesso alle proprietà. Per esempio, l'operatore `+` ha significati diversi a seconda del tipo di valori su cui opera. Invece di eseguire sempre tutto il codice che gestisce tutti i significati, un compilatore JavaScript intelligente userà osservazioni precedenti per prevedere il tipo su cui va applicato l'operatore. Se è sempre stato applicato a numeri, può generare un brano di codice macchina molto più semplice. Anche in questo caso si tratta di presupposti, che vanno verificati ogni volta che si esegue la funzione.

La morale della storia è che se un brano di codice deve essere veloce, voi potete risparmiare un po' di lavoro al compilatore mantenendo coerenza nei tipi. I motori JavaScript possono gestire abbastanza bene situazioni dove incontrano una mezza dozzina di tipi diversi, generando codice che gestisce tutti quei tipi e de-ottimizzando solo quando ne incontrano uno nuovo. Anche in quelle situazioni, comunque, il codice risultante è più lento di quel che sarebbe se ci fosse un tipo solo.

## Riepilogo

Grazie all'enorme quantità di denaro che viene investita nel Web, oltre che alle rivalità tra browser, i compilatori JavaScript diventano sempre più efficienti nello svolgere il loro lavoro: accelerare l'esecuzione del codice.

A volte, però, dovete aiutarli un pochino e riscrivere i cicli interni per evitare le funzionalità più onerose di JavaScript, come le funzioni di ordine superiore. Anche creare pochi oggetti (e array e stringhe) può aiutare.

Prima di cominciare a pasticciare col codice per renderlo più veloce in esecuzione,

pensate a come potete fargli svolgere meno lavoro. Qui si trovano spesso le più grandi opportunità per ottimizzare il rendimento.

I motori JavaScript compilano il codice caldo più volte e usano informazioni raccolte in esecuzioni precedenti per compilare il codice in modo più efficiente. Potete contribuire all'efficienza assegnando alle vostre variabili tipi coerenti.

## Esercizi

### **Trovare un percorso**

Scrivete una funzione `findPath` che cerchi il percorso più breve tra due nodi in un grafo. La funzione accetta come argomenti due oggetti `GraphNode` (come quelli che abbiamo usato in questo capitolo) e restituisce `o null`, se non si trova un percorso, o un array di nodi che rappresentano il percorso attraverso il grafo. In quest'array, i nodi contigui devono avere un arco che li colleghi.

Un buon sistema per trovare un percorso in un grafo può essere il seguente:

1. Create un elenco di lavori che contenga un unico percorso con un solo nodo, che è il nodo di partenza.
2. Partite dal primo percorso dell'elenco di lavori.
3. Se il nodo alla fine del percorso corrente è quello di destinazione, restituire questo percorso.
4. Altrimenti, esamine i vicini del nodo alla fine del percorso: se non sono stati esaminati prima (in quanto non si trovano alla fine dei percorsi dell'elenco di lavori), create un nuovo percorso estendendo il percorso corrente fino a quello vicino e aggiungetelo all'elenco di lavori.
5. Se ci sono più percorsi nell'elenco di lavori, passate al percorso successivo e riprendete dal punto 3.
6. Altrimenti, il percorso non c'è.

“Spargendo” i percorsi a partire dal nodo di partenza, questo approccio assicura che si arrivi sempre all’altro nodo dato attraverso il percorso più breve, perché i percorsi più lunghi vengono presi in considerazione solo dopo aver provato tutti i percorsi più brevi.

Realizzate il programma e verificate lo su qualche semplice grafo ad albero. Costruite un grafo con un ciclo (per esempio aggiungendo connessioni a un grafo ad albero col metodo `connect`) e vedete se la vostra funzione riesce a trovare il percorso più breve in presenza di più possibilità.

### **Misura del tempo**

Usate `Date.now()` per misurare il tempo che la vostra funzione `findPath` ci mette a trovare un percorso in un grafo più complicato. Poiché `treeGraph` mette sempre la radice

all'inizio dell'array graph e una foglia alla fine, potete assegnare alla vostra funzione un compito piuttosto sofisticato, come il seguente:

---

```
var graph = treeGraph(5, 3);
console.log(findPath(graph[0], graph[graph.length - 1]).length);
// → 5
```

---

Create un caso di prova con un tempo di esecuzione intorno al mezzo secondo. Fate attenzione a passare numeri più grandi a treeGraph: le dimensioni del grafo aumentano in maniera esponenziale e si fa in fretta a realizzare grafi tanto grandi da richiedere un sacco di tempo e di memoria per trovare i percorsi.

## Ottimizzare

Ora che avete un caso di prova misurato, trovate la maniera di rendere la funzione findPath più veloce.

Pensate sia alla macro-ottimizzazione (ridurre la quantità di lavoro), sia alla micro-ottimizzazione (svolgere il lavoro dato in maniera più efficiente). Prendete in considerazione anche la possibilità di usare meno memoria e assegnare strutture dati più piccole o inferiori di numero.

Se serve, potete cominciare ad aggiungere proprietà id ai nodi per semplificare la memorizzazione di informazioni su di essi in una mappa. Usate una funzione come la seguente:

---

```
function withIDs(graph) {
  for (var i = 0; i < graph.length; i++) graph[i].id = i;
  return graph;
}
```

---

# SUGGERIMENTI PER GLI ESERCIZI

I suggerimenti riportati di seguito possono esservi di aiuto quando siete in difficoltà con gli esercizi del libro. Non vi danno la soluzione, ma dovrebbero aiutarvi a trovarla da soli.

## Struttura dei programmi

### ***Un ciclo per un triangolo***

Potete cominciare con un programma che si limita a stampare i numeri da 1 a 7, che potete derivare con pochi ritocchi dall'esempio di stampa di numeri pari dato precedentemente nel capitolo, dove abbiamo introdotto il ciclo `for`.

Considerate ora l'equivalenza tra numeri e stringhe di caratteri #: se potete passare da 1 a 2 con la formula `1 (= 1)`, potete andare da "`#`" a "`##`" aggiungendo un carattere (`= "#"`). La vostra soluzione può pertanto seguire da vicino il programma di stampa dei numeri.

### ***FizzBuzz***

Il passaggio sui numeri è chiaramente un lavoro da ciclo e scegliere che cosa stampare è un lavoro da esecuzione condizionale. Ricordate il trucchetto dell'operatore resto (%) per verificare se un numero è divisibile per un altro (in questo caso, ha resto zero).

Nella prima versione, ci sono tre risultati possibili per ogni numero e dovete pertanto impostare una catena di `if/else if/else`.

La seconda versione del programma ha una soluzione semplice e una astuta. Nella prima, si aggiunge un altro "ramo" per provare con precisione la condizione data. Il metodo più astuto richiede di costruire una stringa contenente la parola, o le parole, da stampare e stampare la parola, o il numero se non c'è parola, facendo uso dell'operatore `||`.

### ***Scacchiera***

Potete costruire la stringa partendo con uno spazio vuoto ("") e aggiungendo ripetutamente caratteri. Per andare a capo, si usa il carattere "`\n`" (newline).

Usate `console.log` per esaminare quel che il programma produce.

Per lavorare con due dimensioni, vi serve un ciclo all'interno di un ciclo.

Aggiungete parentesi graffe intorno ai corpi di ciascun ciclo per mettere in evidenza i loro punti di partenza e di arrivo. Sforzatevi di far rientrare correttamente le righe dei corpi. L'ordine dei cicli deve seguire l'ordine di costruzione della stringa (riga per riga, da sinistra a destra e dall'alto in basso). Pertanto, il ciclo più esterno costruisce le righe e

quello interno gestisce i caratteri di ciascuna riga.

Vi servono due variabili per tener nota dell'avanzamento. Per sapere se in una certa posizione va inserito uno spazio o un carattere #, potete verificare se la somma dei due contatori è pari (% 2).

La chiusura di una riga col carattere di a capo avviene dopo che la riga è stata costruita; dovete pertanto prevederla dopo il ciclo interno e dentro il ciclo esterno.

## Funzioni

### *Minimo*

Se avete problemi a posizionare correttamente le parentesi per ottenere una funzione valida, cominciate a copiare uno degli esempi del capitolo e modificalo.

Una funzione può contenere molte dichiarazioni return.

### *Ricorsività*

La vostra funzione sarà simile alla funzione interna find dell'esempio ricorsivo findSolution del capitolo, con una catena if/else if/else per provare quale dei tre casi soddisfa il problema dato. L'istruzione else finale, che corrisponde al terzo caso, effettua la chiamata ricorsiva. Ciascuno dei rami deve contenere una dichiarazione return o prevedere qualche altro modo per restituire un valore specifico.

Col numero negativo, la funzione sarà infinitamente ricorsiva, perché il valore passato sarà sempre negativo e sempre più grande, ossia sempre più lontano dalla restituzione di un risultato. Alla fine, consuma tutto lo spazio della pila e si arresta.

### *Conteggi*

Il vostro ciclo deve esaminare tutti i caratteri della stringa, uno per uno, associandoli a un indice che va da zero alla lunghezza della stringa meno uno (< string.length). Se il carattere nella posizione corrente è uguale a quello che sta cercando, la funzione aggiunge 1 a una variabile contatore. Al termine del ciclo si può restituire questa variabile.

Fate attenzione a impostare come locali tutte le variabili usate nella funzione, usando la parola chiave var .

## Strutture dati: oggetti e array

### *La somma di un intervallo*

Per costruire un array, la cosa più semplice è inizializzare una variabile su un array vuoto con [] e poi richiamare ripetutamente il suo metodo push per aggiungere i valori, uno per

volta. Non dimenticatevi di restituire l'array alla fine della funzione.

Poiché il delimitatore finale è inclusivo, dovete usare l'operatore `<=` invece di `<` per trovare la fine del ciclo.

Per verificare se è stato passato l'argomento facoltativo `step`, potete esaminare `arguments.length` o confrontare il valore dell'argomento con `undefined`. Se manca, impostatelo sul suo valore predefinito (1) in cima alla funzione.

Perché `range` possa interpretare i valori negativi per `step`, la soluzione migliore è di scrivere due cicli separati, uno per contare in avanti e uno per contare all'indietro, perché l'operatore di confronto per trovare la fine del ciclo deve essere `>=` invece di `<=` quando si conta all'indietro.

Può inoltre valer la pena di usare un diverso valore di default per `step`, e precisamente `-1`, quando la fine dell'intervallo è minore del suo inizio. In quel modo, `range(5, 2)` restituisce un valore significativo, invece di bloccarsi su un ciclo infinito.

## Invertire l'ordine di un array

Ci sono due sistemi ovvi per impostare `reverseArray`. Il primo è semplicemente di scorrere l'array di input dall'inizio alla fine e usare il metodo `unshift` sul nuovo array per inserire via via ciascun elemento all'inizio. Il secondo è di passare in ciclo partendo dalla fine dell'array e procedere all'indietro col metodo `push`. Per iterare all'indietro su un array, vi serve una specifica (un po' strana) come `(var i = array.length - 1; i >= 0; i--)`.

Invertire l'array con `reverseArrayInPlace` è più difficile. Dovete fare attenzione a non sovrascrivere elementi che vi serviranno in un secondo tempo. Usare `reverseArray` o copiare tutto l'array (un buon sistema per farlo è `array.slice(0)`) funziona, ma è un po' una truffa.

Il trucco sta nello scambiare il primo e l'ultimo elemento, poi il secondo e il penultimo e così via. Potete farlo passando in ciclo su metà della lunghezza dell'array (usando `Math.floor` per arrotondare per difetto, visto che non c'è bisogno di spostare l'elemento centrale in un array di lunghezza dispari) e scambiando l'elemento in posizione `i` con quello in posizione `array.length - 1 - i`. Potete usare una variabile locale per mantenere temporaneamente il valore di uno degli elementi, sovrascrivere l'elemento con la sua immagine specchiata e spostare il valore dalla variabile locale alla posizione dove si trovava l'immagine specchiata dell'elemento.

## Liste

Per costruire una lista è meglio procedere a rovescio, in modo che `arrayToList` possa iterare a rovescio sull'array (fate riferimento all'esercizio precedente) e aggiungere alla lista un oggetto per ciascun elemento. Potete usare una variabile locale per mantenere la parte della lista costruita finora e usare una struttura come `list = {value: x, rest: list}` per aggiungere un elemento.

Per passare su una lista (in `listToArray` e `nth`), potete usare un ciclo `for` come questo:

---

```
for (var node = list; node; node = node.rest) {}
```

---

Riuscite a capire come funziona? Per ogni iterazione del ciclo, `node` punta alla sottolista corrente e il corpo può leggerne la proprietà `value` per ottenere l'elemento corrente. Alla fine di ogni iterazione, `node` si sposta alla sottolista successiva. Quando il valore è `null`, abbiamo raggiunto la fine della lista e il ciclo si conclude.

Analogamente, la versione ricorsiva di `nth` andrà a esaminare una parte ancora più piccola della “coda” della lista, contando a rovescio fino a zero, quando può restituire la proprietà `value` del nodo che sta esaminando. Per ottenere l'elemento in posizione zero, prendete semplicemente la proprietà `value` del suo nodo di testa. Per ottenere l'elemento  $N + 1$ , prendete il valore in posizione  $N$  della lista che si trova nella proprietà `rest` di questa lista.

## **Confronto profondo**

La prova per sapere se avete a che fare con un vero oggetto sarà una cosa come `typeof x == "object" && x != null`. Fate attenzione a confrontare le proprietà solo quando entrambi gli argomenti sono oggetti. In tutti gli altri casi potete restituire immediatamente il risultato di `==`.

Usate un ciclo `for/in` per passare le proprietà. Dovete verificare se i due oggetti hanno lo stesso insieme di nomi di proprietà e se quelle proprietà hanno valore identico. La prima prova si può fare contando le proprietà dei due oggetti e restituendo `false` se il risultato è diverso. Se hanno lo stesso numero di proprietà, esaminate i nomi di quelle di un oggetto e confrontateli uno per uno con quelli delle proprietà dell’altro oggetto. I valori delle proprietà si confrontano con una chiamata ricorsiva a `deepEqual`.

Per restituire il valore corretto dalla funzione, la cosa migliore è restituire immediatamente `false` quando si trova una differenza e `true` solo alla fine della funzione.

## **Funzioni di ordine superiore**

### **Differenza di età tra madri e figli**

Poiché non tutti gli elementi dell’array degli antenati producono dati utili (perché non possiamo calcolare la differenza di età, se non conosciamo la data di nascita della madre), dovremo applicare dei filtri prima di richiamare `average`. Potete farlo al primo passaggio, definendo una funzione `hasKnownMother` come filtro. In alternativa, potete prima richiamare `map` e restituire nella funzione di mappatura o la differenza di età oppure `null`, se non si hanno dati sulla madre. Richiamate quindi `filter` per eliminare gli elementi `null` prima di passare l’array ad `average`.

## **Speranza di vita storica**

L'essenza di questo esempio sta nel raggruppare gli elementi di una collezione in base a qualche aspetto comune. Qui, vogliamo dividere l'array di antenati in array più piccoli che li raggruppino per secolo.

Per questo, mantenete un oggetto che associa i nomi dei secoli (numeri) con array di persone oppure di età. Poiché non sappiamo in partenza che categorie troveremo, dovremo crearle al volo. Per ciascuna persona, dopo averne calcolato il secolo, verifichiamo se si tratta di uno dei secoli noti. Altrimenti, aggiungiamo un array. Aggiungiamo poi la persona (o l'età) all'array del secolo di appartenenza.

Infine, potete usare un ciclo `for/in` per stampare le età medie per ciascun secolo.

## **Tutti e qualcuno**

Le funzioni possono seguire uno schema simile alla definizione di `forEach` all'inizio del capitolo, salvo che devono restituire immediatamente (col valore giusto) quando la funzione predicativa restituisce `false` o `true`. Non dimenticatevi di inserire un'altra dichiarazione `return` dopo il ciclo, in modo che la funzione restituisca il valore giusto quando arriva alla fine dell'array.

## **La vita segreta degli oggetti**

### **Un tipo vettore**

La vostra soluzione può seguire abbastanza fedelmente la struttura del costruttore Rabbit di questo capitolo.

Per aggiungere una proprietà `get` al costruttore, potete usare la funzione `Object.defineProperty`. Per calcolare la distanza tra  $(0, 0)$  e  $(x, y)$ , potete usare il teorema di Pitagora, che dice che il quadrato della distanza che stiamo cercando è uguale al quadrato della coordinata  $x$  più il quadrato della coordinata  $y$ . Pertanto, il numero che cerchiamo è  $\sqrt{x^2 + y^2}$  e il calcolo della radice quadrata in JavaScript si fa con `Math.sqrt`.

### **Un'altra cella**

Dovrete memorizzare tutti e tre gli argomenti del costruttore nell'oggetto istanza. I metodi `minWidth` e `minHeight` devono effettuare le chiamate ai metodi corrispondenti nella cella `inner`, facendo in modo, magari attraverso `Math.max`, che nessun valore restituito sia un numero inferiore alle dimensioni date.

Non dimenticate di aggiungere un metodo `draw` che ritrasmetta la chiamata alla cella interna.

## **Interfacce per sequenze**

Un modo per risolvere questo problema è di dare alla sequenza uno stato di oggetto, in modo che le sue proprietà vengano modificate intanto che ci si lavora sopra. Potreste poi memorizzare un contatore che indichi l'avanzamento dell'oggetto sequenza.

Come minimo, la vostra interfaccia deve esporre il modo per arrivare all'elemento successivo e scoprire se l'iterazione è arrivata alla fine della sequenza (o no). Magari vi tenta l'idea di raggruppare tutto in un solo metodo, `next`, che restituisce `null` o `undefined` quando la sequenza è finita. Con questo, però, avrete un problema se la sequenza contiene `null`. Pertanto, è preferibile avere un metodo separato (o una proprietà `get`) per scoprire se si è arrivati alla fine.

In un'altra soluzione, potete evitare di cambiare stato all'oggetto. Potete esporre un metodo per recuperare l'elemento corrente (senza far avanzare un contatore); un altro per recuperare una nuova sequenza che rappresenti gli elementi dopo quello corrente, oppure un valore speciale quando si arriva alla fine della sequenza. Questa soluzione è piuttosto elegante: il valore di una sequenza rimane sempre lo stesso anche dopo averlo usato e può pertanto essere condiviso con altro codice, senza preoccuparci di quel che può succedere in futuro. Purtroppo, in JavaScript questa soluzione è piuttosto inefficiente perché richiede di creare troppi oggetti durante l'iterazione.

## Progetto: vita elettronica

### *Stupidità artificiale*

Il problema della voracità può essere affrontato in vari modi. Le creature potrebbero smettere di mangiare quando arrivano a un certo livello di energia. Oppure, potrebbero mangiare solo dopo N cicli, mantenendo un contatore dei giri dall'ultimo pasto in una proprietà dell'oggetto `critter`. Oppure, per fare in modo che le piante non si estinguano mai completamente, le creature potrebbero mangiare una pianta solo se ce n'è un'altra vicina, usando il metodo `findAll` sulla vista. Possono funzionare anche una combinazione di questi tre metodi o strategie completamente diverse.

Per far spostare le creature in modo più efficiente, potremmo basarci su una delle strategie di movimento del vecchio mondo, quello senza energia. Sia il comportamento di salto, sia quello del seguire il muro mostravano una maggior varietà di movimenti rispetto a questo vagabondaggio completamente casuale.

Rallentare la riproduzione delle creature è semplicissimo. Basta aumentare il livello di energia minima al quale si riproducono. Naturalmente, rendere l'ecosistema più stabile lo rende anche più noioso. Un mucchio di creature grasse e immobili, che si nutrono di un mare di piante e non si riproducono mai, costituiscono un ecosistema molto stabile, ma che non interessa a nessuno.

### *Predatori*

Molti dei trucchi che aiutano a risolvere l'esercizio precedente funzionano anche per questo. La soluzione che vi suggerisco è di aumentare le dimensioni dei predatori (con tanta energia) e farli riprodurre più lentamente: ciò li rende meno vulnerabili nei periodi di carestia, quando ci sono pochi erbivori.

Oltre a quello di rimanere vivi, un altro obiettivo fondamentale dei predatori è di mantenere vive le proprie fonti di rifornimento. Fate in modo che i predatori caccino più aggressivamente quando ci sono molti erbivori e più lentamente (o per nulla) quando le prede scarseggiano. Poiché gli erbivori si spostano, il semplice trucco di mangiarne uno solo quando ce ne sono altri vicini probabilmente non funziona: la cosa avverrà talmente di rado, che i predatori finiranno col morire di fame. Potete, però, tener nota di quel che succede nei turni precedenti in qualche struttura dati mantenuta sugli oggetti predatore, in modo da basare il loro comportamento su quel che è successo di recente.

## Bachi e gestione degli errori

### Riprova

La chiamata a `primitiveMultiply` deve ovviamente aver luogo in un blocco `try`. Il blocco `catch` corrispondente deve rilanciare l'eccezione quando non è un'istanza di `MultiplicatorUnitFailure` e garantire la riprova della chiamata quando lo è.

Per la riprova, potete usare o un ciclo che si interrompe quando una chiamata riesce, come nell'esempio `look` di questo capitolo, oppure provare con la ricorsività, sperando (probabilmente invano) di non ricevere una catena di fallimenti tanto lunga da riempire la pila.

### La scatola chiusa

Come avrete immaginato, questo esercizio richiede un blocco `finally`. La vostra funzione deve prima sbloccare la scatola e poi richiamare la funzione argomento dall'interno di un corpo `try`. Il blocco `finally` che segue deve richiudere la scatola.

Per essere sicuri di non chiudere la scatola quando è già chiusa, verificate la sua proprietà `locked` all'inizio della funzione e apritela solo se essa risulta `true` in partenza.

## Espressioni regolari

### Stile delle citazioni

La soluzione più ovvia è di sostituire solo le virgolette che hanno un carattere non alfanumerico da almeno un lato, come per esempio `/\w ' | ' \w/`. Dovete tener conto, però, anche dell'inizio e della fine di riga.

Inoltre, dovete assicurarvi che la sostituzione comprenda i caratteri che corrispondono all'espressione regolare \w e non li elimini. Potete farlo racchiudendoli tra parentesi e aggiungendo i loro gruppi alla stringa di sostituzione (\$1, \$2). I gruppi che non trovano corrispondenza non verranno sostituiti.

## Ancora numeri

Innanzitutto, non dimenticatevi della barra rovesciata davanti al punto. Per trovare corrispondenza col segno facoltativo davanti al numero e/o all'esponente si usano [+\\-]? o (\+|-|) (più, meno o niente). La parte più complicata dell'esercizio è trovare corrispondenza sia per "5.", sia per ".5", senza trovarla per ". ". Una buona soluzione per questo è di usare l'operatore | per separare i due casi: una o più cifre, possibilmente seguite da un punto e zero o più cifre, oppure un punto seguito da una o più cifre.

Infine, per accettare indifferentemente la e maiuscola o minuscola, aggiungete un'opzione i all'espressione regolare oppure usate [eE].

## Moduli

### Nomi dei mesi

Questo esercizio segue fedelmente il modulo `weekday`. Un'espressione di funzione, richiamata immediatamente, avvolge la variabile che mantiene l'array di nomi, insieme alle due funzioni che bisogna esportare. Le funzioni sono raccolte in un oggetto e restituite. L'oggetto interfaccia restituito viene memorizzato nella variabile `month`.

### Ritorno alla vita elettronica

Ecco la mia soluzione. Ho inserito tra parentesi le funzioni interne.

---

```
Modulo "grid"
  Vector
  Grid
  directions
  directionNames
Modulo "world"
  (randomElement)
  (elementFromChar)
  (charFromElement)
  View
  World
  LifelikeWorld
```

```
directions [ri-esportato]
Modulo "simple_ecosystem"
  (randomElement) [duplicato]
  (dirPlus)
  Wall
  BouncingCritter
  WallFollower
Modulo "ecosystem"
  Wall [duplicato]
  Plant
  PlantEater
  SmartPlantEater
  Tiger
```

---

Ho ri-esportato l'array `directions` del modulo `grid` dell'oggetto `World` in modo che i moduli costruiti su di esso (gli ecosistemi) non debbano sapere o preoccuparsi dell'esistenza del modulo `grid`.

Ho inoltre duplicato due piccoli valori ausiliari generici, `randomElement` e `Wall`, in quanto vengono usati come particolari interni in diversi contesti e non appartengono alle interfacce di quei moduli.

## **Dipendenze circolari**

Il trucco sta nell'aggiungere l'oggetto `exports`, creato per un modulo, all'oggetto `cache` di `require` prima di eseguire il modulo. Ciò significa che, poiché il modulo non ha ancora avuto la possibilità di prevalere su `module.exports`, non sappiamo se vuole esportare qualche altro valore. Dopo averlo caricato, sull'oggetto `cache` si prevale con `module.exports`, che può avere un valore diverso.

Se intanto che viene caricato il modulo se ne carica un secondo che richiede il primo, il suo oggetto `exports` predefinito, che probabilmente a questo punto è ancora vuoto, sarà in cache e il secondo modulo riceverà un riferimento a esso. Le cose funzionano a condizione che il secondo modulo non cerchi di fare nulla con l'oggetto finché il primo non sia stato caricato completamente.

# **Progetto: un linguaggio di programmazione**

## **Array**

Il modo più semplice per questo è di rappresentare gli array di Egg come array di JavaScript. I valori aggiunti all'ambiente superiore devono essere funzioni.

Potete usare `Array.prototype.slice` per convertire un oggetto simile a un array di

argomenti in un array vero e proprio.

## **Chiusura**

Anche in questo caso seguiamo l'esempio di un meccanismo di JavaScript per ottenere la funzionalità equivalente in Egg. I modelli speciali vengono passati all'ambiente locale in cui sono elaborati, così che possano elaborare le proprie forme derivate in quell'ambiente. La funzione restituita da `fun` si chiude sopra l'argomento `env` assegnato alla funzione in cui è inscritta e usa quell'argomento per creare l'ambiente locale della funzione, quando viene richiamato.

Ciò significa che il prototipo dell'ambiente locale sarà l'ambiente dove è stata creata la funzione, il che permette di accedere alle variabili di quell'ambiente dalla funzione. Quel che serve per realizzare la chiusura è tutto qui, anche se dovete svolgere altro lavoro perché venga compilata in modo efficiente.

## **Commenti**

Controllate che la vostra soluzione gestisca più commenti sulla stessa riga, che possono avere spazi vuoti all'inizio o alla fine.

Un'espressione regolare è probabilmente il sistema migliore per risolvere il problema. Scrivete qualcosa che trovi “spazio vuoto o un commento, una o più volte”. Usate il metodo `exec` o `match` ed esamineate la lunghezza del primo elemento nell'array restituito (tutto quel che avete trovato) per scoprire quanti caratteri vanno eliminati.

## **Sistemare l'ambito di visibilità**

Dovrete passare in ciclo su un ambito per volta, usando `Object.getPrototypeOf` per passare all'ambito immediatamente più esterno. In ciascun ambito, usate `hasOwnProperty` per scoprire se vi esiste la variabile, indicata dalla proprietà `name` del primo argomento di `set`. Se esiste, impostatela sul risultato dell'elaborazione del secondo argomento e restituite quel valore.

Se raggiungerete l'ambito più esterno (`Object.getPrototypeOf` restituisce `null`) senza trovare la variabile, vuol dire che non esiste e bisogna lanciare un errore.

# **Il modello a oggetti del documento: DOM**

## **Costruire una tabella**

Utilizzate `document.createElement` per creare nuovi nodi per elementi, `document.createTextNode` per creare nodi di testo e il metodo `appendChild` per inserire nodi all'interno di altri nodi.

Dovete passare in ciclo sui nomi `keys`, una volta per riempire la riga di testata e poi

una volta per ciascun oggetto nell'array per le righe dei dati.

Non dimenticate l'elemento <table> di chiusura della tabella alla fine della funzione.

## ***Elementi per nome di tag***

La soluzione si esprime nel modo più semplice con una funzione ricorsiva simile alla funzione `talksAbout` definita nel capitolo.

Potete far sì che `byTagName` si richiami ricorsivamente, concatenando gli array risultanti per produrre l'output. Per un approccio più efficiente, definite una funzione interna che si richiami ricorsivamente e abbia accesso a una variabile array definita nella funzione esterna, alla quale può man mano aggiungere gli elementi che trova. Non dimenticate di richiamare la funzione interna una volta da quella esterna.

La funzione ricorsiva deve verificare il tipo del nodo. Qui ci interessano solo i nodi di tipo 1 (`document.ELEMENT_NODE`). Quando ne troviamo, passiamo in ciclo i rispettivi nodi figli ed effettuiamo il confronto su ciascun figlio per vedere se corrisponde a quanto cerchiamo, mentre in parallelo effettuiamo una chiamata ricorsiva sul figlio per ispezionare i suoi nodi figli.

## **Gestire gli eventi**

### ***La tastiera censurata***

Per risolvere l'esercizio, bisogna impedire il comportamento predefinito degli eventi di tastiera. Potete gestire indifferentemente "keypress" o "keydown". Se richiamate `preventDefault` su uno dei due, la lettera non viene digitata.

Per identificare le lettere digitate bisogna esaminare le proprietà `key-Code` o `charCode` e confrontarle coi codici delle lettere che volete filtrare. In "keydown", non dovete preoccuparvi di maiuscole e minuscole, perché identifica solo il tasto premuto. Se invece decidete di gestire eventi "keypress", che identifichino il carattere digitato, dovete controllare sia le maiuscole, sia le minuscole. Ecco un modo per farlo:

---

```
/[qwx]/i.test(String.fromCharCode(event.charCodeAt))
```

---

### ***La scia del mouse***

Il sistema migliore per creare gli elementi è impostare un ciclo per aggiungerli in coda al documento man mano che li trovate. Memorizzate gli elementi della scia in un array per potervi accedere in un secondo tempo e cambiare la loro posizione.

I passaggi in ciclo si possono svolgere mantenendo una variabile contatore e aggiungendovi 1 ogni volta che scatta l'evento "mousemove". Potete poi usare l'operatore

resto (% 10) per ottenere un numero di indice valido e scegliere l'elemento che volete posizionare durante un evento dato.

Un altro effetto interessante si ottiene modellando un piccolo sistema di fisica. Usate un unico evento "mousemove" per aggiornare la coppia di variabili che seguono la posizione del mouse. Usate poi requestAnimationFrame per simulare un effetto dove gli elementi della scia vengono attirati verso la posizione del puntatore del mouse.

Per ogni passo dell'animazione, aggiornate le loro posizioni in relazione alla posizione del puntatore e, se volete, a una certa velocità memorizzata per ciascun elemento. Lascio a voi il compito di trovare un buon modo per ottenere questo effetto.

## Schede

È probabile che vi imbattiate nel problema di non poter usare direttamente la proprietà childNodes del nodo come collezione di nodi per le schede. Intanto, quando aggiungete i pulsanti, anch'essi diventano nodi figli dello stesso oggetto, perché l'oggetto è vivo. Inoltre, i nodi di testo creati per lo spazio vuoto tra nodi sono anch'essi parte dello stesso oggetto e non devono ricevere schede a loro volta.

Per aggirare il problema, cominciate col costruire un array reale di tutti i nodi figli del nodo di partenza che hanno 1 come nodeType .

Nel registrare i gestori di eventi sui pulsanti, le funzioni di gestione dovranno sapere quale elemento della scheda è associato col pulsante. Se li create in un ciclo normale, potete accedere alla variabile index del ciclo dall'interno della funzione, ma non otterrete il numero corretto perché nel frattempo la variabile sarà stata modificata dal ciclo.

Invece, usate un metodo forEach e create le funzioni di gestione dall'interno della funzione passata a forEach . L'indice del ciclo, che viene passato a quella funzione come secondo argomento, sarà qui una variabile locale normale e non verrà sovrascritta dalle iterazioni.

## Progetto: un videogioco a piattaforme

### Game Over

La soluzione più ovvia è di impostare la variabile lives in modo che "viva" in runGame e sia pertanto visibile alla chiusura di startLevel .

Un'altra soluzione, in linea con lo spirito del resto della funzione, può essere di aggiungere un secondo parametro a startLevel , che indichi il numero delle vite. Quando tutto lo stato del sistema è memorizzato negli argomenti di una funzione, richiamare quella funzione offre un sistema elegante per passare a un nuovo stato.

In ogni caso, quando un livello è perso, dovrebbero esserci due transizioni di stato possibili. Se si tratta dell'ultima vita, si ritorna al livello zero con la quantità di vite di partenza. Altrimenti, ripetiamo il livello corrente diminuendo di uno il numero delle vite

restanti.

## **Interrompere il gioco**

Un'animazione si può interrompere restituendo `false` dalla funzione passata a `runAnimation` e riprendere richiamando di nuovo `runAnimation`.

Per comunicare alla funzione passata a `runAnimation` che l'animazione deve essere interrotta (e la funzione può restituire `false`), potete usare una variabile accessibile sia dal gestore dell'evento, sia dalla vostra funzione.

Quando trovate la maniera per deregistrare i gestori registrati da `trackKeys`, ricordate che, per eliminare correttamente un gestore, dovete passare a `removeEventListener` lo stesso identico valore di funzione che avevate passato ad `addEventListener`. Pertanto, il valore della funzione di gestione creata in `trackKeys` deve essere disponibile per il codice che deregistra i gestori.

Potete aggiungere una proprietà all'oggetto restituito da `trackKeys`, che contenga o il valore di quella funzione o un metodo che gestisce direttamente la deregistrazione.

## **Disegnare su un foglio**

### **Forme geometriche**

Il trapezoide (1) si disegna semplicemente con un percorso. Scegliete delle coordinate adatte per il centro e aggiungete i quattro angoli intorno a quel punto.

Il rombo (2) si può disegnare nella maniera semplice, col percorso o con un sistema più interessante basato su una trasformazione rotatoria. Per usare la rotazione, dovete applicare una soluzione simile a quel che abbiamo fatto con la funzione `flipHorizontally`. Poiché dovete ruotare intorno al centro del rettangolo, e non intorno al punto (0,0), dovete prima spostarvi in quel punto, poi ruotare e quindi ritornare indietro.

Per la linea a zig zag (3) non conviene scrivere una nuova chiamata a `lineTo` per ciascuno dei segmenti. Invece, dovete usare un ciclo. Potete fare in modo che ogni iterazione disegni due segmenti di retta (a destra e poi a sinistra) o uno solo, in questo caso utilizzando le posizioni pari (% 2) dell'indice del ciclo per stabilire se bisogna andare verso destra o verso sinistra.

Vi serve un ciclo anche per la spirale (4). Se disegnate una serie di punti dove ciascun punto si sposta in avanti seguendo una traiettoria circolare intorno al centro della spirale, otterrete un cerchio. Se durante il ciclo variate il raggio della circonferenza per il punto corrente e continuate il cerchio, il risultato sarà una spirale.

La stella di esempio (5) si basa su linee tracciate con `quadraticCurveTo`. Potete disegnarne una anche con linee rette. Dividete un cerchio in otto spicchi o tanti spicchi

quante sono le punte della stella. Tracciate le linee che collegano i punti in modo che pieghino verso il centro della stella. Con `quadraticCurveTo`, potete usare il centro come punto di controllo.

## Il grafico a torta

Dovrete richiamare `fillText` e impostare le proprietà `textAlign` e `textBaseline` del contesto fino a portare il testo nella posizione voluta.

Un modo sensato per posizionare le etichette è di posizionarle sulla retta che va dal centro della torta alla mediana dello spicchio. Non conviene posizionare il testo direttamente sulla circonferenza: meglio spostarlo di qualche pixel a lato.

L'angolo di questa linea è `currentAngle + 0.5 * sliceAngle`. Il codice che segue trova una posizione su questa linea, a 120 pixel dal centro:

---

```
var middleAngle = currentAngle + 0.5 * sliceAngle;
var textX = Math.cos(middleAngle) * 120 + centerX;
var textY = Math.sin(middleAngle) * 120 + centerY;
```

---

Per `textBaseline`, la soluzione migliore in questa situazione è di impostarlo sul valore "middle". Nella scelta per `textAlign`, dipende da che parte del cerchio siamo: se siamo sulla sinistra, dovrebbe essere impostato su "right" e se siamo sulla destra su "left", in modo che il testo si allontani sempre dal cerchio.

Se non sapete come scoprire da che parte del cerchio sta un dato angolo, rileggete la spiegazione di `Math.cos` del [Capitolo 13](#). Il coseno di un angolo ci dice a quale coordinata x corrisponde, che a sua volta indica esattamente il lato del cerchio su cui ci troviamo.

## Una palla che rimbalza

La scatola si disegna facilmente con `strokeRect`. Definite una variabile che mantenga le sue dimensioni oppure definitene due se altezza e larghezza della scatola sono diverse. Per costruire una palla rotonda, iniziate un percorso, richiamate `arc(x, y, radius, 0, 7)`, che crea un arco che va da zero a un po' più della circonferenza e riempitela.

Per modellare posizione e velocità della palla, potete usare il tipo `Vector` del [Capitolo 15](#). Assegnate una velocità iniziale, preferibilmente una che non sia puramente verticale od orizzontale; per ogni frame, moltiplicate poi quella velocità per la quantità di tempo trascorso. Quando la palla si avvicina troppo a una parete verticale, invertite la componente x della sua velocità. Analogamente, invertite la componente y quando si tratta di base o "coperchio" della scatola.

Dopo aver trovato i nuovi valori per posizione e velocità della palla, usate `clearRect` per ripulire la scena e ritracciarla con la nuova posizione.

## Riflessioni precalcolate

La soluzione sta nel fatto che, con `drawImage`, possiamo usare un elemento Canvas come

immagine sorgente. È possibile creare un <canvas> aggiuntivo, senza aggiungerlo al documento, e disegnarci una volta gli sprite riflessi. Quando poi disegniamo un fotogramma, ci basta copiare gli sprite già invertiti sull'area di disegno principale.

Bisogna fare un po' di attenzione perché le immagini non caricano istantaneamente.

Facciamo il disegno invertito solo una volta e, se lo facciamo prima che l'immagine sia caricata, il disegno non viene visualizzato. Si può usare una funzione di gestione di eventi "load" sull'immagine per disegnare le immagini invertite sulla tela aggiuntiva. Questa può poi essere usata immediatamente come sorgente del disegno (sarà vuoto solo finché non ci disegniamo sopra il personaggio).

## HTTP

### *Negoziazione dei contenuti*

Riguardate gli esempi di questo capitolo che fanno uso di XMLHttpRequest per rilevare le chiamate ai metodi necessarie per effettuare una richiesta. Se volete, potete usare una richiesta sincrona impostando il terzo parametro di open su false.

Chiedere un tipo di media inesistente restituisce una risposta con codice 406, “Not acceptable” (non accettabile), che è il codice che il server deve restituire quando non può soddisfare l'header Accept .

### *Aspettare più promesse*

La funzione passata al costruttore Promise dovrà richiamare then su ciascuna delle promesse nell'array dato. Quando una di esse ha successo, devono aver luogo due cose: il valore risultante deve essere memorizzato nella giusta posizione dell'array restituito e dobbiamo verificare se questa è l'ultima promessa in attesa, e in questo caso concludere la nostra promessa.

L'ultima operazione si può effettuare con un contatore, che va inizializzato sulla lunghezza dell'array di input e dal quale sottrarremo 1 ogni volta che una promessa ha successo: quando arriva a 0, abbiamo finito. Ricordatevi di prendere in considerazione la situazione dove l'array di input è vuoto (e pertanto nessuna promessa verrà mai risolta).

Gestire gli insuccessi richiede un po' di riflessione, ma è in effetti semplicissimo. Basta passare la funzione di insuccesso della promessa esterna a ciascuna delle promesse dell'array, in modo che un insuccesso in una di esse faccia scattare l'insuccesso per tutta la funzione che le avvolge.

## Moduli e relativi campi

## **Un laboratorio per JavaScript**

Usate `document.querySelector` o `document.getElementById` per ottenere accesso agli elementi definiti nel vostro codice HTML. Un gestore di eventi "click" o "mousedown" sul pulsante può recuperare la proprietà `value` del campo testo e richiamarvi sopra `new Function`.

Ricordate di avvolgere sia la chiamata a `new Function`, sia la chiamata al suo risultato in un blocco `try` per poter catturare le eccezioni prodotte. Poiché, in effetti, non sappiamo che tipo di eccezione ci interessa, dobbiamo catturarle tutte.

Potete usare la proprietà `textContent` dell'elemento di output per riempirlo con un messaggio in formato stringa. Oppure, potete scegliere di memorizzare da qualche parte il vecchio contenuto, creare un nuovo nodo di testo con `document.createTextNode` e metterlo in coda a quell'elemento. Ricordate di aggiungere un a capo alla fine, in modo che l'output non risulti tutto sulla stessa riga.

## **Autocompletamento**

L'evento migliore per aggiornare l'elenco di suggerimenti è "`input`", in quanto scatta non appena cambia il contenuto del campo.

Passate poi in ciclo sull'array di termini per vedere se iniziano con la stringa data. Per esempio, potete richiamare `indexOf` e vedere se il risultato è zero. Ogni volta che trovate una stringa corrispondente, aggiungete un elemento all'elemento `<div>` dei suggerimenti. Ogni volta che iniziate ad aggiornare i suggerimenti, dovete svuotarlo, per esempio impostando la sua proprietà `textContent` sulla stringa vuota.

Potete aggiungere un gestore di eventi "click" a ciascun elemento dei suggerimenti o aggiungerne uno solo all'elemento `<div>` esterno che li mantiene ed esaminare la proprietà `target` dell'evento per identificare il suggerimento selezionato.

Per recuperare il testo dei suggerimenti da un nodo del DOM, potete esaminare la sua proprietà `textContent` oppure impostare un attributo per memorizzare esplicitamente il testo mentre create l'elemento.

## **Il Gioco della vita di Conway**

Per fare in modo che i cambiamenti avvengano concettualmente tutti allo stesso tempo, provate a pensare al calcolo di una generazione come a una funzione pura, che accetta una griglia e produce una nuova griglia che rappresenta il turno successivo.

Per rappresentare la griglia, potete scegliere uno qualunque dei sistemi descritti nei [Capitoli 15 e 17](#). Il conteggio dei vicini vivi si può effettuare con due cicli nidificati, che passano in ciclo su coordinate adiacenti. Attenzione a non contare le celle fuori dal campo e a ignorare la cella (al centro) di cui stiamo contando i vicini.

Per modificare le caselle di spunta in modo che i cambiamenti abbiano effetto per la generazione successiva, potete procedere in due modi. In uno, un gestore di eventi può prender nota dei cambiamenti e aggiornare la griglia corrente per rifletterli, oppure, potete

generare una nuova griglia dai valori contenuti nelle caselle di spunta prima di calcolare il turno successivo.

Se scegliete i gestori di eventi, dovete collegarvi degli attributi che identifichino la posizione corrispondente a ciascuna casella di spunta, in modo che sia facile scoprire quale cella modificare.

Per tracciare la griglia di caselle di spunta, potete usare un elemento `<table>` ([Capitolo 13](#)) o semplicemente inserirle tutte nello stesso elemento inserendo un tag `<br>` (interruzione di riga) tra una riga e l'altra.

## Progetto: Un programma per dipingere

### Rettangoli

Potete usare `relativePos` per trovare l'angolo corrispondente all'inizio del trascinamento del mouse. Per rilevare dove il trascinamento finisce, potete usare `trackDrag` oppure registrare un vostro gestore di eventi.

Quando avete ottenuto due angoli del rettangolo, dovete trovare un modo per tradurli negli argomenti che `fillRect` si aspetta: l'angolo superiore sinistro, la larghezza e l'altezza del rettangolo. Potete usare `Math.min` per trovare le posizioni all'estrema sinistra dell'asse x e del punto più alto dell'asse y. Per recuperare larghezza o altezza, potete richiamare `Math.abs` (il valore assoluto) sulla differenza tra due lati.

Visualizzare il rettangolo mentre il mouse viene trascinato richiede un insieme di numeri simile, ma nel contesto della pagina invece che relativo all'area di disegno. Per non dover riscrivere le stesse procedure due volte, potete impostare una funzione `findRect` che converta due punti in un oggetto con proprietà `top`, `left`, `width` e `height`.

Potete poi creare un nodo `<div>` e impostarne `style.position` su `absolute`. Quando impostate gli stili per il posizionamento, non dimenticate di aggiungere "px" dopo i numeri. Il nodo va aggiunto al documento (potete inserirlo in coda a `document.body`) e poi eliminato quando termina il trascinamento del mouse e il rettangolo risulta disegnato sull'area di disegno.

### Tavolozza per i colori

Anche qui dovete usare `relativePos` per identificare il pixel sul quale l'utente ha fatto clic. La funzione `pixelAt` dell'esempio dimostra come recuperare i valori per un pixel dato. Aggiungerli a una stringa `rgb` richiede semplicemente dei concatenamenti di stringhe.

Ricordatevi di verificare che l'eccezione che catturate sia un'istanza di `SecurityError`, in modo da non gestire accidentalmente il tipo di eccezione sbagliato.

### Secchiello e riempimento

Data una coppia di coordinate di partenza e i dati sull'immagine per tutto il foglio, l'esempio che segue dovrebbe funzionare:

1. Create un array per mantenere le informazioni sulle coordinate già colorate.
2. Create un array temporaneo (lista di lavoro) per mantenere le coordinate da esaminare. Aggiungetevi la posizione iniziale.
3. Quando la lista di lavoro è vuota, abbiamo finito.
4. Eliminate una coppia di coordinate dalla lista di lavoro.
5. Se quelle coordinate sono già parte dell'array di pixel colorati, ritornate al passo 3.
6. Colorate il pixel alle coordinate correnti e aggiungete le coordinate all'array di pixel colorati.
7. Aggiungete alla lista di lavoro le coordinate dei pixel adiacenti che hanno lo stesso colore del pixel di partenza.
8. Ritornate al passo 3.

La lista di lavoro può essere semplicemente un array di oggetti vettore. La struttura dati che tiene nota dei pixel colorati sarà consultata molto spesso. Pertanto, analizzarla integralmente tutte le volte che si visita un nuovo pixel richiederà un sacco di tempo. Potete invece creare un array contenente un valore per ciascun pixel, usando di nuovo lo schema  $x + y \times$  larghezza per associare le posizioni con ciascuno dei pixel. Nel verificare se un pixel è già stato colorato, potete accedere direttamente al campo che corrisponde al pixel corrente.

Potete confrontare i colori esaminando la parte che vi interessa dell'array di dati, confrontando un campo per volta. Oppure, potete “condensare” un colore in un solo numero o stringa e confrontare quelli. Se scegliete questa soluzione, controllate che ogni colore produca un valore univoco. Per esempio, la semplice somma dei componenti del colore non sarà sufficiente, perché ci saranno più colori che danno lo stesso totale.

Quando elencate i vicini di un punto dato, fate attenzione a escludere quelli che non sono dentro al foglio o il programma finirà col perdere in calcoli senza fine.

## Node.js

### Ancora negoziazione dei contenuti

Non dimenticate di richiamare il metodo `end` sull'oggetto restituito da `http.request` per dare avvio alla richiesta.

L'oggetto risposta passato alla funzione di callback di `http.request` è un flusso in lettura. Ciò significa che non è esattamente semplice recuperare da esso tutto il corpo della risposta. La seguente funzione di utilità legge un intero flusso e richiama una funzione di callback col risultato, seguendo il solito schema di passare eventuali errori come primo

argomento della funzione di callback:

---

```
function readStreamAsString(stream, callback) {  
    var data = "";  
    stream.on("data", function(chunk) {  
        data += chunk;  
    });  
    stream.on("end", function() {  
        callback(null, data);  
    });  
    stream.on("error", function(error) {  
        callback(error);  
    });  
}
```

---

## **Tappare una falla**

È sufficiente eliminare tutte le occorrenze di due punti che hanno da entrambi i lati o una barra o una barra rovesciata o la fine della stringa. Il sistema migliore è attraverso il metodo `replace` con un'espressione regolare. Non dimenticate l'opzione `g` nell'espressione regolare, altrimenti il metodo sostituirà una sola istanza: il rischio è che la gente riesca poi a scavalcare questa misura di sicurezza e ad aggiungere doppi punti nei percorsi richiesti. Ricordate inoltre di effettuare la sostituzione dopo aver decodificato la stringa o anche in questo caso qualcuno potrebbe aggirare il filtro e codificare un punto o una barra.

Un altro caso potenzialmente preoccupante è quando i percorsi iniziano con una barra e vengono interpretati come percorsi assoluti. Poiché `urlToPaths` aggiunge un punto davanti al percorso, è impossibile creare richieste che diano questo percorso. Più barre di seguito, all'interno del percorso, sono strane, ma saranno trattate dal sistema di file come una barra sola.

## **Create directory**

Potete usare la funzione che implementa il metodo `DELETE` come modello per il metodo `MKCOL`. Quando non si trova il file, provate a creare una directory con `fs.mkdir`. Se in quel percorso esiste una directory, potete restituire una risposta 204 in modo che le richieste di creazione di directory siano idempotenti. Se in quella posizione esiste un file diverso da una directory, restituite un codice di errore. In queste situazioni, è appropriato il codice 400 (“bad request”).

## **Uno spazio pubblico sul Web**

Potete creare un elemento <textarea> per mantenere il contenuto del file che si sta modificando. Potete usare una richiesta GET, attraverso XMLHttpRequest, per recuperare il contenuto corrente del file. Potete usare URL relativi, come *index.html*, invece di <http://localhost:8000/index.html>, per far riferimento a file sullo stesso server di quello dove è in esecuzione lo script.

Poi, quando l'utente fa clic su un pulsante (potete usare un elemento <form> e un evento "submit", oppure semplicemente un gestore di eventi "click"), effettuate una richiesta PUT sullo stesso URL, col contenuto del campo <textarea> come corpo della richiesta, per salvare il file.

Potete poi aggiungere un elemento <select> che contiene tutti i file nella directory radice del server aggiungendo elementi <option> contenenti le righe restituite da una richiesta GET all'URL /. Quando l'utente seleziona un altro file (un evento "change" sul campo), lo script deve recuperare e visualizzare quel file. Inoltre, accertatevi di usare il nome file selezionato quando salvate il file.

Purtroppo, il server è troppo elementare per riuscire a leggere correttamente file dalle sottodirectory, perché non ci dice se quel che abbiamo recuperato con una richiesta GET è un file o una directory. Riuscite a pensare a un modo per estendere il server e risolvere questo problema?

## Progetto: Un sito Web di skill-sharing

### **Persistenza su disco**

La soluzione più semplice che ho trovato è di convertire tutto l'oggetto `talks` in formato JSON e scaricarlo su un file con `fs.writeFile`. Esiste già una funzione (`registerChange`) che viene richiamata ogni volta che cambiano i dati sul server. Può essere estesa per fare in modo che scriva su disco i nuovi dati.

Scegliete un nome file, per esempio `./talks.json`. Quando il server si avvia, può provare a leggere quel file con `fs.readFile`; se ha successo, può usare il contenuto del file come dati di partenza.

Attenzione però: l'oggetto `talks` era partito come oggetto senza prototipo, per poter usare senza rischi l'operatore `in`. `JSON.parse` restituisce invece oggetti normali con `Object.prototype` come prototipo. Se usate JSON come formato file, dovrete copiare le proprietà dell'oggetto restituito da `JSON.parse` in un nuovo oggetto senza prototipo.

### **Reimpostare i campi per i commenti**

L'approccio più adatto è semplicemente di memorizzare lo stato di un campo `commenti` di una presentazione (il suo contenuto e se è attivo) prima di ri-tracciare la presentazione e quindi reimpostare il campo al suo stato precedente.

In un'altra soluzione, invece di sostituire semplicemente la vecchia struttura DOM con la nuova, potete confrontarle ricorsivamente, nodo per nodo, e aggiornare solo le parti

effettivamente modificate. Questa soluzione è molto più difficile da realizzare, ma è meno specifica e continua a funzionare anche se aggiungessimo altri campi testo.

## Modelli migliori

Potete modificare `instantiateTemplate` in modo che la sua funzione interna accetti come argomento non solo un nodo, ma anche un contesto corrente. Potete poi, quando passate in ciclo sui nodi figli di un nodo, verificare se il figlio ha un attributo `template-repeat`. Se ce l'ha, non istanziateolo solo una volta, ma passate in ciclo sull'array indicato dal valore dell'attributo e istanziate il nodo una volta per ciascun elemento dell'array, passando l'elemento di array corrente come contesto.

Le istanziazioni condizionali si ottengono in modo analogo, con attributi chiamati, per esempio, `template-when` (modello per *quando*) e `template-unless` (modello per *tranne quando*), che provocano l'istanziazione di un nodo solo quando una data proprietà è `true` (o `false`).

## Senza script

Due aspetti centrali dell'impostazione descritta in questo capitolo, un'interfaccia HTTP pulita e la rappresentazione di modelli lato client, non funzionano senza JavaScript. I moduli HTML normali possono trasmettere richieste `GET` e `POST`, ma non `PUT` né `DELETE`; inoltre, possono trasmettere i dati solo a un URL fisso.

Pertanto, andrebbe reimpostato il server in modo che accetti commenti, nuove presentazioni e presentazioni eliminate attraverso richieste `POST`, i cui corpi non sono in formato JSON ma usano il formato URL previsto per i modelli in HTML ([Capitolo 17](#)). Queste richieste dovrebbero restituire ogni volta tutta la pagina per permettere agli utenti di vedere il nuovo stato del sito dopo ogni modifica. Questa soluzione non è troppo difficile da realizzare e potrebbe essere affiancata all'interfaccia HTTP “pulita”.

Il codice per produrre le presentazioni dovrebbe essere duplicato sul server. Il file `index.html`, invece che statico, dovrebbe essere generato dinamicamente aggiungendo un gestore specifico al router. In questo modo, comprende già commenti e presentazioni correnti quando viene servito.

## JavaScript e prestazioni

### Trovare un percorso

La lista di lavoro può essere un array, al quale potete aggiungere percorsi col metodo `push`. Non potete usare il metodo integrato `forEach` per passare in ciclo sugli elementi, perché quello usa la lunghezza iniziale dell'array per delimitare il ciclo. Poiché aggiungeremo nuovi elementi all'array dall'interno del ciclo, la sua lunghezza va ricontrrollata dopo ogni iterazione.

Se usate gli array per rappresentare i percorsi, potete estenderli col metodo `concat`, come in `path.concat([node])`.

Per scoprire se un nodo è già stato visto, potete passare in ciclo normalmente sull'elenco di lavori oppure usare il metodo `some`.

## Ottimizzare

La migliore opportunità di macro-ottimizzazione sta nel fare a meno del ciclo interno che rileva se un nodo è già stato esaminato. Farlo in un oggetto è molto più veloce che iterare sull'elenco in cerca del nodo. Poiché gli oggetti mappa, in JavaScript, come nomi di proprietà vogliono stringhe e non oggetti, dobbiamo ricorrere a un espediente come `withIDs` per poter usare una mappa per associare informazioni con un oggetto dato. La prossima versione di JavaScript definisce un tipo `Map` per gli oggetti, che è una vera mappa, le cui chiavi possono essere valori JavaScript qualunque e non solo stringhe.

Un altro miglioramento si può fare cambiando il modo in cui i percorsi sono memorizzati. Per estendere un array con un nuovo elemento senza modificare l'array esistente, bisogna copiare tutto l'array. Una struttura dati come la lista di [Capitolo 4](#) non ha questo problema, in quanto consente a più liste di condividere i dati che hanno in comune.

Potete fare in modo che la vostra funzione memorizzi internamente i percorsi come oggetti con proprietà `last` e `via`, dove `last` è l'ultimo nodo del percorso e `via` è o `null`, oppure un altro oggetto analogo. In questo modo, per estendere un percorso basta creare un oggetto con due proprietà, invece di copiare tutto un array. Ricordate di convertire la lista in un array vero e proprio prima di restituirla.

## **INDICE ANALITICO**

(operatore)

= (operatore)

: (operatore)

| (array)

(operatore)

= (operatore)

(operatore)

= (operatore)

re commerciale (carattere)

$\&$  (operatore)

&

>= (operatore)

`<lt;= (operatore)`

ó (operatore)

- (operatore)

+ (operatore)

= (operatore)

: (operatore)

: (operatore)

= (operatore)

$\equiv$  (operatore)

(operatore)

(operatore)

(operatore)

00 (codice di stato HTTP)

## 04 (codice di stato HTTP)

d (contesto Canvas)

00 (codice di stato HTTP)

## 04 (codice di stato HTTP)

## 05 (codice di stato HTTP)

06 (codice di stato HTTP)

00 (codice di stato HTTP)

## A

(tag HTML)

capo (carattere)

abelson Hal

ccept (header)

ccess-Control-Allow-Origin (header)

ctionTypes (oggetto)

ctiveElement (proprietà)

ctorAt (metodo)

ddEventListener (metodo)

ddizione

lbero sintattico astratto, v. albero sintattico

lbero sintattico

lbero

lert (funzione)

ll (funzione)

lt (attributo)

LT (tasto)

lternanza di maiuscole e minuscole

ltKey (proprietà)

mbiente

mbito di livello superiore, vedere ambito globale

mbito di visibilità globale

mbito di visibilità locale

mbito di visibilità

MD

nalisi dinamica

nalogia coi libri

NCESTRY\_FILE (raccolta dati)

nimate (funzione)

nimazione fluida

nimationi

come evitare salti di avanzamento

frame, per

fuori dallo schermo

gatto nell'ellissi

gioco

interrompere

mettere in scadenza

recuperare elementi per nome di tag

rilevamento della collisione

scia del mouse

sprite

SVG e Canvas

ntenato (elemento)

ppendChild (metodo)

ppiattire degli array (esercizio)

apple

pplicazione (di funzioni), v. funzioni, applicazione

pplicazioni web

pply (metodo)

pprossimazione

ppunti (esempio)

rc (metodo)

rcTo (metodo)

rea di disegno

arcTo (metodo)

bezierCurve (metodo)

caricamento dei dati sui pixel

cerchi

`clearRect` (metodo)

contesto

definizione

dimensioni

`drawGraph` (funzione)

interfacce

percorso

prestazioni

richieste per altri domini

riempimento e bordi

`stroke` (metodo)

tavolozza (esercizio)

`toDataURL` (metodo)

tracciare del testo

trasformazioni

rgomento facoltativo

rgomento

`rguments` (oggetto)

`rgv` (proprietà)

ritmetica

`rmstrong`, Joe

`rray` (costruttore)

`rray` (prototipo)

`rray`

appiattire

cercare

come tabella

creazione

definizione

elemento casuale da

`every` (metodo)

filtrare  
`forEach` (metodo)  
funzioni di ordine superiore e  
indicizzazione  
`map` (metodo)  
metodi  
`reverse` (metodo)  
`slice` (metodo)  
`some` (metodo)  
`splice` (metodo)  
rrotondamento  
rte ASCII  
`ssert` (funzione)  
strazione  
urelio Marco  
utocompletamento (esercizio)  
 `autofocus` (attributo)  
`verage` (funzione)

## B

abbage, Charles  
achi  
definizione  
espressioni regolari e  
proprietà `lastIndex` e  
`ackground` (CSS)  
`ackgroundReadFile` (funzione)  
acktracking  
anks Ian  
arra (carattere)  
arra degli indirizzi  
arra rovesciata carattere di

escape

nelle espressioni regolari

nelle stringhe

arra rovesciata marcatore di confine

eforeunload (evento)

Jerners-Lee Tim

ezierCurve (metodo)

ind (metodo)

it

lob (tipo)

occo di commenti

occo

lock-level (elementi)

lur (evento)

lur (metodo)

ody (proprietà)

ody (tag HTML)

*look of Programming*

olean (funzione)

oleans

conversione in

definizione

esecuzione condizionale

tipi immutabili

order (CSS)

order-radius (CSS)

ouncingCritter (tipo)

ox shadow (CSS)

r (tag HTML)

reak (parola chiave)

rowser

ambito globale

bitmap e

console

doctype,

eventi.

finestre di dialogo

long polling

sessionStorage

sicurezza

spazio privato dei nomi

storia di JavaScript

supporto per campi colore

supporto per JavaScript

velocità di risposta

XMLHttpRequest,

rowserify

uffer, (tipo)

utton (tag HTML)

yName (oggetto)

## C

: (programming language)

ache in linea

ache

alcolo

all (metodo)

ampi dei form in HTML5

ancelAnimationFrame (funzione)

anvas (proprietà)

anvas (tag HTML)

anvasDisplay (tipo)

apitalizzazione

con la tastiera censurata  
nei nomi di header  
nei nomi di proprietà  
nei nomi di variabile  
apitolo  
appello del gatto (esercizio)  
arattere alfanumerico  
arattere  
aratteri di escape  
in HTML  
negli URL  
nelle espressioni regolari  
nelle stringhe  
arestia  
aricamento  
'ascading Style Sheets, vedere CSS  
ase (parola chiave)  
atch (metodo)  
atch (parola chiave)  
atena alimentare  
ID  
hange (evento)  
harAt (metodo)  
harCode (proprietà)  
harCodeAt (metodo)  
hat  
hecked (attributo)  
hiamata ai metodi  
hildNodes (proprietà)  
hiusura  
hrome

icli

conclusione di

corpo

definizione

for,

nidificati

while

ifre (numeri)

ircle (tag SVG)

itazioni in JSON

lass (attributo)

lassName (proprietà)

learInterval (funzione)

learRect (metodo)

learTimeout (funzione)

lick (evento)

lient

lientHeight (proprietà)

lientWidth (proprietà)

lientX (proprietà)

lientY (proprietà)

loneNode (metodo)

losePath (metodo)

MD (tasto)

ode golf

odice caldo/hot code

odice macchina

odice

odifica

odifica URL

odifiche di caratteri

oefficiente phi

oerenza  
oin (tipo)  
ollaborazione  
ollezione  
olor (CSS)  
olwidths (funzione)  
OMMENT\_NODE code  
ommenti su una riga  
ommenti  
lommonJS  
ompatibilità con versioni precedenti  
ompilazione per gradi  
omplessità  
ompletamento  
omplicazione  
omponibilità  
omportamento predefinito  
omportamento  
omposizione  
omputer  
oncat (metodo)  
oncatenamento  
onfigurazione  
onfine  
onfini di parola  
onfirm (funzione)  
onfronto  
reare valori booleani  
onfronto  
di colori  
di nodi DOM

di numeri  
di oggetti  
di stringhe  
di valori undefined  
per costrutti switch  
profondo (esercizio)  
profondo  
onigli (esempio)  
onsole JavaScript  
onsole.log,  
onteggi (esercizio)  
onteggi in base zero  
ontent-Length (header)  
ontent-Type (header)  
ontinue (parola chiave)  
ontrollo  
di accesso  
di versione  
ortografico (esempio)  
ontrols (oggetto)  
onvalida  
onvenzioni  
onversione tra maiuscole e minuscole  
oordinate  
orpo (HTTP)  
orrelazione  
orrispondenze  
oseno  
ostanti  
ostruttori  
ereditarietà  
Error

operatore instanceof e  
presentazione  
proprietà prototype per  
senza new (parola chiave)  
reateElement (metodo)  
reatePaint (funzione)  
reateReadStream (funzione)  
reateServer (funzione)  
reateTextNode (metodo)  
reateWriteStream (funzione)

reature

animazione  
comportamento  
comportamento  
gestori di azioni e  
predatori

rockford Douglas  
ronologia (non c'è)  
SS  
TRL (tasto)  
tr1Key (proprietà)  
urdi e tastiera censurata  
url (programma)

## D

ark Blue (game)  
ata- (prefisso per nomi di attributi)  
atabase  
ataTable (funzione)  
ate (campo)  
ate (costruttore)  
ate (tipo)

ate.now (funzione)  
ati  
blclick (evento)  
ebouncing  
ebugger (dichiarazione)  
ebugging  
con asservimenti  
con catch (parola chiave)  
con console.log  
con use strict  
definizione  
Error (costruttore)  
presentazione  
decentralizzazione  
decodeURIComponent (funzione)  
efault (parola chiave)  
efine (funzione)  
efineProperty (funzione)  
ELETE (metodo)  
elete (operatore)  
enodeify (funzione)  
enominazione  
eottimizzazione  
eserializzazione  
igrammi di flusso  
iario  
ichiarazione with  
ichiarazioni  
ifferenza di età (esercizio)  
ijkstra, Edsger  
imensioni dei programmi

imensioni  
ipendenza  
ipendenza circolare  
ipendenza tra variabili statistiche  
irections (oggetto)  
irectory  
irezione della bussola  
isabled (attributo)  
isco rigido  
isegno  
caselle di scelta  
creare un programma di  
e foglio (elemento)  
sottosistemi nel videogioco  
isplay (CSS)  
isplay (oggetto)  
ivisione  
)NA  
o (ciclo)  
otype  
)document Object Model, *vedere* DOM  
ocumentazione  
ocumentElement (proprietà)  
ollaro (carattere \$)  
)OM  
ddEventListener,  
canvas (elemento)  
cloneNode,  
costruzione  
creare strutture in JavaScript  
CSS  
documentElement

eventi e  
form  
immagini  
interfacce  
oggetti  
presentazione  
proprietà degli oggetti  
selettori e  
value (proprietà)  
OMDisplay (tipo)  
omini  
rawGraph (funzione)  
rawImage (metodo)  
rawTable (funzione)  
rawTalk (funzione)  
ue punti (carattere)  
uplicazione

## E

ccezioni non catturate  
CMAScript  
cosistema  
cstatic (modulo)  
ffetti collaterali  
ffetto hover  
fficienza  
leganza  
LEMENT\_NODE code  
lementFromChar (funzione)  
lements (proprietà)  
levazione a potenza  
lse (parola chiave)

`lt` (funzione)

`mail` (campo)

`ncodeURIComponent` (funzione)

`nd` (metodo)

`NOENT` (codice di stato)

ntropia

reditarietà

`rror` (tipo)

rrori di digitazione

rrori in fase di esecuzione

`sc` (tasto)

'scher M.C.

secuzione condizionale

sercizi

sperienza utente

sportazione

spressione letterale

spressioni regolari

backtracking

caratteri di escape

confine/limite

corrispondenze

creazione

debugging

insiemi di caratteri

`lastIndex` (proprietà)

metodi

metodo `replace` e

operatori greedy

opzione globale

opzione

presentazione  
raggruppamento  
spressioni  
**val**  
**valuate** (funzione)  
**very** (metodo)  
videnziazione della sintassi (esempio)  
**xec** (metodo)  
**xit** (metodo)  
**xports** (oggetto)

## F

**alse**  
**ar** saltare lo stack  
**attoria** (esempio)  
**iglet** (modulo)  
**ile** (tipo)  
**le** *inifile* statici  
**ileReader** (oggetto)  
**iles** (proprietà)  
**ill** (metodo)  
**illColor** (proprietà)  
**illRect** (metodo)  
**illStyle** (proprietà)  
**illText** (metodo)  
**ilter** (metodo)  
**inally** (parola chiave)  
**nestra** di alert  
**nestra** di dialogo  
**inish** (evento)  
**irefox**  
**irstChild** (proprietà)

izzBuzz (esercizio)

essibilità

lipHorizontally (funzione)

lipHorizontally (metodo)

ussi

usso di controllo

ocus (evento)

ocus (metodo)

ocus per la tastiera, *vedere* focus

ogli stile, *vedere* CSS

old (funzione)

ont-family (CSS)

ont-weight (CSS)

or (attributo)

or (ciclo)

or/in (ciclo)

orEach (metodo)

orm (proprietà)

orm (tag HTML)

ormato dati

ormato document

ormato file

orme (esercizio)

orme a zig zag

orzatura del tipo

ammentazione

attale (esempio)

romCharCode (funzione)

s (modulo)

sp (oggetto)

unction (costruttore)

unction (parola chiave)

unction (prototipo)

inzioni

ambito di visibilità

argomenti errore

callback gestori di eventi

callback promesse

come proprietà

come spazi dei nomi

come valori

come valori

corpo

definizione

definizione

di carica dei moduli

di impostazione

di ordine superiore

dichiarazione

divisione del programma e

ed effetti collaterali

flussi in lettura

flussi in scrittura

## G

AME\_LEVELS (dataset)

estione degli errori

avvio di script

catch (metodo)

catch (parola chiave)

codici di stato HTTP

comportamento predefinito

con promesse

DOM e  
eccezioni non gestite  
ENOENT  
eventi di tastiera  
focus (evento)  
funzioni di callback  
load (evento)  
long polling e  
nei flussi in lettura  
nei giochi  
oggetti evento  
parentesi angolari  
per server di file  
presentazione  
programmazione asincrona  
propagazione degli eventi  
scroll (evento)  
target (proprietà)  
then (metodo)  
estione delle eccezioni  
blocco del programma  
eccezioni non catturate  
gestione degli errori e  
metodo toDataURL e  
presentazione  
try (parola chiave)  
ET (metodo)  
etAttribute (metodo)  
etBoundingClientRect (metodo)  
etContext (metodo)  
etDate (metodo)

etDay (metodo)  
etElementById (metodo)  
etElementsByClassName (metodo)  
etElementsByName (metodo)  
etElementsByTagName (metodo)  
etFullYear (metodo)  
etHours (metodo)  
etImageData (metodo)  
etItem (metodo)  
etMinutes (metodo)  
etMonth (metodo)  
etPrototypeOf (funzione)  
etResponseHeader (metodo)  
etSeconds (metodo)  
etTime (metodo)  
etURL (funzione)  
etYear (metodo)  
iocatore (personaggio)  
ioco  
accelerazione del personaggio  
azioni  
esecuzione  
movimenti e collisioni  
piattaforma  
seguire i tasti  
tracciare lo schermo  
ioco della vita di Conway  
iorno della settimana (esempio)  
lobalCompositeOperation (proprietà)  
oethe Johann Wolfgang von  
oogle

raffe, *vedere* parentesi graffe

rafico a torta (esempio)

rammatica

raphNode (tipo)

rid (tipo)

roupBy (funzione)

uerre dei browser

## H

1 (tag HTML)

asEvent (funzione)

ash (#), (carattere)

asOwnProperty (metodo)

ead (proprietà)

ead (tag HTML)

eader (in HTTP)

eight (CSS)

elp (link)

idden (elemento)

ighlightCode (funzione)

ost (header)

ref (attributo)

tml (tag HTML)

HTML

tp (modulo)

TP

tps (modulo)

TPS

hypertext Markup Language, *vedere* HTML

hypertext Transfer Prototol, *vedere* HTTP

## I

'O asincrono

'O sincrono

d (attributo)

identificatore

identità

f (parola chiave)

img (tag HTML)

nimmagini

alternative

e Canvas

nel videogioco a piattaforme

specchiate

vettoriali e bitmap

implements (parola riservata)

incapsulazione definizione

incapsulazione ereditarietà e

index (proprietà)

*index.html*

indexOf (metodo)

indirizzo IP

infinito (ciclo)

infinity

inline (elemento)

inner (funzione)

innerHeight (proprietà)

innerWidth (proprietà)

input (evento)

input (tag HTML)

input di testo

input

inquinamento dello spazio dei nomi

`insertBefore` (metodo)  
insieme di dati  
insieme vuoto  
`install` (comando)  
`instanceof` (operatore)  
`newInstanceTemplate` (funzione)  
integrazione  
interfacce  
moduli e  
oggetti come  
progettazione  
sequenza  
tabella  
interfaccia  
a schede (esercizio)  
grafica utente  
utente  
interface (parola riservata)  
internazionalizzazione  
internet Explorer  
internet  
interno (ciclo)  
interpretazione  
interrompere il gioco (esercizio)  
interruzione dei programmi  
interruzione di riga  
inversione (esercizio)  
inversione  
NVIO (tasto)  
sEven (esercizio)  
sInside (funzione)

sNaN (funzione)

stanza

struzione

terazione

## J

acques, lo scoiattolo mannaro

avaScript

isponibilità di

flessibilità di

Object Notation, *vedere* JSON

punti deboli di

sintassi

storia di

usi di

versioni di

o in (metodo)

OURNAL (raccolta dati)

SON

SON.parse (funzione)

SON.stringify (funzione)

## K

Kernighan, Brian

eyCode (proprietà)

eydown (evento)

ypress (evento)

eyup (evento)

hasekhemwy

Inuth, Donald

## L

abel (tag HTML)

ast-Modified (header)

astChild (proprietà)

astIndex (proprietà)

astIndexOf (metodo)

eft (CSS)

genda

ge di Hooke

ggere il codice

ggibilità

ength (proprietà),

per array

per stringhe

et (parola chiave)

evel (tipo)

brerie

ifeLikeworld (tipo)

ineCap (proprietà)

ineTo (metodo)

inewidth (proprietà)

ngua mongola, separatore di vocali

nguaggi di

programmazione

forza dei

storia

nguaggi specifici per

domini

nguaggi umani

nguaggio Egg

ambiente

analisi

array supporto per

booleani supporto per

commenti

compilazione

funzioni

imbrogli

interprete

presentazione

uniformità

`ink` (tag HTML)

`nk`

`ste` (esercizio)

`isten` (metodo)

`oad` (evento)

`ocalhost`,

`ocalStorage` (oggetto)

`ong` polling

## M

`iaggiore di`

`AIUSC` (tasto)

`ap` (metodo)

`rappe`

`iassimo`

`atch` (metodo)

`iematica`

`ath` (oggetto)

`ath.abs` (funzione)

`ath.ceil` (funzione)

`ath.cos` (funzione)

`ath.floor` (funzione)

`ath.max` (funzione)

`ath.min` (funzione)

ath.PI constant  
ath.random (funzione)  
ath.round (funzione)  
ath.sin (funzione)  
ath.sqrt (funzione)  
ax-height (CSS)  
ax-width (CSS)  
imedia (tipo)  
Meditoriente grafo di  
Mefistofele  
memoria  
meno  
menu contestuale  
essage (evento)  
messaggi di errore  
ETA (tasto)  
etaKey (proprietà)  
ethod (attributo)  
ethods (oggetto)  
metodi  
micro-ottimizzazione  
Microsoft  
Microsoft Paint  
ime (modulo)  
IME (tipo)  
ini applicazione  
inimo  
inore di  
Miró Joan  
KCOL (metodo)  
kdir (funzione)

modello mentale

odule (oggetto)

moduli

grandi

interfacce

NPM

quando sono necessari

scopo

sistema AMD

modulo (operatore)

moltiplicazione

ondo

Mosaic

otori fisici

OUNTAINS (raccolta dati)

ouse

cursore

pulsante

ouse

ousedown (evento)

ousemove (evento)

ouseout (evento)

ouseover (evento)

ouseup (evento)

oveTo (metodo)

Mozilla

ultiple (attributo)

ultiplier (funzione)

nuro

utabilità

**N**

ame (attributo)

aN

egoziazione dei contenuti (esercizio)

emici (esempio)

erd

Jetscape

ew (operatore)

extSibling (proprietà)

idificazione

di ambito

di arrays

di espressioni

di funzioni

nelle espressioni regolari

occioline e scoiattolo mannaro (esempio)

ode programma

ode\_modules (directory)

Iode.js

console.log e

DELETE (metodo)

flussi e

fs (modulo)

GET (metodo)

HTTP (modulo)

interruzione degli script

long polling e

NPM

presentazione

programmazione asincrona e

PUT (metodo)

server di file (esempio)

sistema di moduli

odeList (tipo)

odeType (proprietà)

odeValue (proprietà)

odi figli

odi figli diretti

odi padre

odo di testo

odo foglia

omi dei mesi (esercizio)

otazione scientifica

JPM

ull,

umber (campo)

umber (funzione)

umeri

binari

conversione in

corrispondenza con espressioni regolari

decimali

esadecimali

in virgola mobile

interi

notazione

pari

precisione dei

rappresentazione

tipi immutabili

valori speciali

umero casuale

**O**

**bject** (prototipo)  
**bject.keys** (funzione)  
**bstacleAt** (metodo)  
**ffsetHeight** (proprietà)  
**ffsetWidth** (proprietà)  
**ggetti**  
come mappe  
creazione  
evento  
globali  
identità  
immutabili  
**instanceof** (operatore)  
**Math**  
passare in ciclo su  
proprietà  
storia della programmazione orientata agli oggetti  
**n** (metodo)  
**nclick** (attributo)  
**pen** (metodo)  
**OpenGL**  
operatore binario  
operatori logici  
operatori  
**ption** (tag HTML)  
**ptions** (proprietà)  
ra Unix  
redinamento  
stacoli  
ttimizzazione  
analisi dinamica

chiarezza del codice  
compilazione  
raggruppare gli aggiornamenti  
rispetto dei tipi  
utput  
verflow (CSS)

## P

(tag HTML)  
ackage (parola riservata)  
ackage.json (file)  
adding (CSS)  
aesaggio (esempio)  
ageX (proprietà)  
ageXOffset (proprietà)  
ageY (proprietà)  
ageYOffset (proprietà)  
'alef Thomas  
arametri  
arentesi angolari  
definizione  
flussi in scrittura  
funzioni di callback  
interfaccia promesse  
lettura dei file  
Node e  
per cicli multipli  
presentazione  
arentesi  
cicli for  
cicli while  
dichiarare la precedenza

dichiarazioni if  
espressioni regolari  
espressioni  
funzioni e  
arentesi graffe  
arentesi quadre  
arentNode (proprietà)  
arole chiave  
arole riservate  
arse (funzione)  
arseApply (funzione)  
arseExpression (funzione)  
assword (campo)  
ercorso  
assoluto  
sistema di file  
URL  
eredità di memoria  
hi (funzione)  
i greco  
ila delle chiamate  
ipe (metodo)  
itagora (teorema di)  
ixel  
izza e scoiattolo mannaro (esempio)  
lant (tipo)  
lantEater (tipo)  
lauger, P.J.  
layer (tipo)  
lus (funzione)  
olimorfismo

olling  
op (metodo)  
opper Karl  
orte  
ose (del personaggio)  
osition (CSS)  
osizionamento  
assoluto  
fisso  
relativo  
ost (metodo)  
ostMessage (metodo)  
ower (esempio)  
re (tag HTML)  
re-elaborazione  
recedenza  
redatori (esercizio)  
restazioni  
compilazione  
espressioni regolari e  
funzioni sincrone  
motori JavaScript  
thread singoli  
videogiochi  
revalenza  
revedibilità  
reventDefault (metodo)  
reviousSibling (proprietà)  
rimitiveMultiply (esercizio)  
rivacy  
rivate (parola riservata)

**r**ivate (proprietà)  
**r**oblemi coi numeri (esempio)  
**r**ocess (oggetto)  
**r**ogetti (capitoli sui)  
**r**ogetto di skill-sharing  
**r**ogramma di disegno (esempio)  
**r**ogrammazione  
ambito  
asincrona,  
creazione  
difensiva  
difficoltà della  
gioia della  
JavaScript  
letteraria  
orientata agli oggetti  
poco elegante  
programmazione web  
rogrammi  
romessa  
**r**omise (costruttore)  
**r**ompt (funzione)  
**r**omptDirection (funzione)  
**r**omptInteger (funzione)  
ropagazione, *vedere* propagazione degli eventi  
roprietà  
ambito globale e  
analisi  
assegnazione  
condivise  
`console.log`

denominazione  
eliminazione  
enumerabili  
interfacce e  
metodi e  
modelli di  
`Object.keys` (funzione)  
parentesi quadre e  
presentazione  
prove sulle  
uso delle parentesi  
`rotected` (parola riservata)  
rotocolli  
rototipi  
creare oggetti da  
funzioni e  
interferenza  
`Object.prototype`  
presentazione  
`prototype` (proprietà)  
rove di controllo  
roxy  
roxy inverso  
seudo array, *vedere* oggetto simile a un array  
seudo-selettore  
`ublic` (parola riservata)  
untatore  
unto (carattere)  
unto di domanda  
unto di interruzione  
unto e virgola

ure (funzione)

ush (metodo)

ut (metodo)

## Q

uadraticCurveTo (metodo)

uadrato

uerySelector (metodo)

uerySelectorAll (metodo)

## R

accolta dei rifiuti

adice

adice quadrata

aggruppamento

andomElement (funzione)

ange (funzione)

eadAsDataURL (metodo)

eadAsText (metodo)

eaddir (funzione)

eadFile (funzione)

eadFileSync (funzione)

eadStreamAsJSON (funzione)

eadStreamAsString (funzione)

ect (SVG tag)

ecupero degli errori

educe (metodo)

educeAncestors (funzione)

eferenceError (tipo)

egExp (costruttore),

egexp golf (esercizio)

egisterChange (funzione)

**egole** (CSS)

**impostare il campo commenti** (esercizio)

**elatedTarget** (proprietà)

**elativePos** (funzione)

**emoveChild** (metodo)

**emoveEventListener** (metodo)

**emoveItem** (metodo)

**ename** (funzione)

**eplace** (metodo)

**eplaceChild** (metodo)

**eplaceSelection** (funzione)

**equest** (funzione)

**equestAnimationFrame** (funzione)

**equire** (funzione)

**\requireJS**

**esponseText** (proprietà)

**esponseXML** (proprietà)

**\sto** (operatore)

**\store** (metodo)

**\sult** (proprietà)

**\turn** (parola chiave)

**\verse** (metodo)

**gb** (CSS)

**\cerca**

**\cetta zuppa di piselli**, analogia con

**\chiamo** (di funzioni), *vedere* applicazione delle funzioni

**\conoscimento** di maiuscole e minuscole

**\corsività** ramificata,

**\corsività**

**\entro dal margine**

**\ga di comando**

ghe di codice  
levamento delle collisioni  
manere in ascolto (TCP)  
petizione dei tasti  
petizione  
produzione  
pulire  
torno a capo  
utilizzo  
otate (metodo)  
outer (tipo)  
owHeights (funzione)  
TextCell (tipo)  
un (funzione)  
unAnimation (funzione)  
unGame (funzione)  
unLayout (funzione)  
unLevel (funzione)

## S

afari  
alto  
andbox  
ave (metodo)  
cacchiera (esercizio)  
cale (metodo)  
caricare  
catola chiusa (esercizio)  
catola  
celta di un valore con switch  
celte di routing  
celte

cia del mouse (esercizio)  
coiattolo mannaro (esempio)  
**cript** (tag HTML)  
crivere codice  
**croll** (evento),  
**earch** (metodo)  
ecchiello e riempimento (esercizio)  
ecolo  
ecure HTTP, *vedere* HTTPS  
**ecurityError** (tipo)  
egnale  
egni  
egno di omissione ^ (carattere)  
egno di sottolineatura (carattere)  
eguire il muro  
**elect** (tag HTML)  
**lected** (attributo)  
**electionEnd** (proprietà)  
**electionStart** (proprietà)  
**elf** variabile  
emplicità  
**end** (metodo)  
eno  
equenza (esercizio)  
equenza  
erializzazione  
erie  
erver  
di conversione in maiuscole (esempio)  
di file (esempio)  
di file (modulo)

`sessionStorage` (oggetto)  
`setAttribute` (metodo)  
`setInterval` (funzione)  
`setItem` (metodo)  
`setRequestHeader` (metodo)  
`setTimeout` (funzione)

**ezioni**

`hift` (metodo)  
`hiftKey` (proprietà)

**imile a un array** (oggetto)

**imulazioni**

**intassi**

assegnazione  
cicli  
dichiarazioni  
esecuzione condizionale  
espressioni  
funzioni come spazio dei nomi  
funzioni  
gestione degli errori  
modalità `strict`  
notazione con parentesi graffe  
notazione della dichiarazione  
parole riservate  
stringhe  
variabili

`ize` (attributo)  
`kipSpace` (funzione)  
`lice` (metodo)

**MTP**

**ollevare eccezioni**

ome (metodo)  
omme (esempio)  
omme (esercizio)  
ottrazione  
ource (proprietà)  
ovraccarico, coi numeri  
pazio dei nomi XML  
pazio dei nomi  
pazio pubblico (esercizio)  
pazio unificatore  
pazio vuoto  
pecialForms (oggetto)  
pecifiche di standard (non c'è)  
peranza di vita (esercizio)  
plice (metodo)  
plit (metodo)  
prise  
quare (esempio)  
rc (attributo)  
tabilità  
tack, *vedere* pila delle chiamate  
tandard output  
tandard  
tar Trek  
tat (funzione)  
tatic (parola riservata)  
tato  
tats (tipo)  
tatus (proprietà)  
tatusText (proprietà)  
tdout (proprietà)

tile delle citazioni (esercizio)

tile di programmazione

cicli

complessità e

incapsulazione

interfacce

promesse

rientri a margine

spazio vuoto

uso del segno di sottolineatura

**topPropagation** (metodo)

toria

**trechCell** (esercizio)

trict mode

**tring** (funzione)

tringhe

metodi

notazione

proprietà

ricerca

tipi immutabili

**troke** (metodo)

**trokeRect** (metodo)

**trokeStyle** (proprietà)

**trokeText** (metodo)

**trong** (tag HTML)

trumenti di sviluppo

trumenti

trumento

gomma

rettangolo (esercizio)

truttura dati viva  
truttura del codice  
truttura  
trutture a livelli  
trutture dati  
nell'esempio sul coniglio assassino  
semplici  
trutture dati  
trutture di prova (testing framework)  
trutture  
tupidità artificiale (esercizio)  
`tyle` (tag HTML)  
`ubmit` (evento)  
`um` (funzione)  
ussman Gerald  
VG  
volgere la pila  
`witch` (parola chiave)  
`yntaxError` (tipo)

## T

`AB` (tasto)  
`abella` (esempio)  
`abelle`  
`abindex` (attributo)  
`able` (tag HTML)  
`ableFor` (funzione)  
`abulatore` (carattere)  
`ig` con chiusura implicita  
`ig` di apertura  
`ig` di chiusura  
`agName` (proprietà)

alksAbout (funzione)  
ingente  
arget (proprietà)  
istiera censurata (esercizio)  
istiera  
ivolozza dei colori (esercizio)  
'CP  
d (tag HTML)  
emplate-repeat (attributo)  
empo  
entacoli (analogia)  
ezoria  
ernario (operatore)  
est (metodo)  
uesto suggerito (esempio)  
EXT\_NODE code  
ext-align (CSS)  
extAlign (proprietà)  
extarea (tag HTML)  
extBaseline (proprietà)  
extCell (tipo)  
extContent (proprietà)  
h (tag HTML)  
hen (metodo)  
his,  
iread  
hrow (parola chiave)  
meout  
imes (metodo)  
pi dinamici  
pi statici

pi

title (tag HTML)

odataURL (metodo)

oLowerCase (metodo)

ools (oggetto)

oString (metodo)

oUpperCase (metodo)

r (tag HTML)

accia dello stack

rackDrag (funzione)

rackKeys (funzione)

ransform (CSS)

ranslate (metodo)

transmission Control Protocol, vedere TCP

asmissione

attino (carattere)

reeGraph (funzione)

iangolo (esercizio)

rim (metodo)

ovare un percorso (esercizio)

rue

ry (parola chiave)

irco e tastiera censurata

itti e qualcuno (esercizio)

witter

ype (attributo)

ype (proprietà)

ypeof (operatore)

## U

nario (operatore)

ndefined,

Jnicode

niformità

nità di misura (CSS)

Jniversal Resource Locator, *vedere* URL

Jnix

nlink (funzione)

nshift (metodo)

RL (campo)

r1 (modulo)

JRL

JRL relativi

r1ToPath (funzione)

se strict

JTF

## V

alore di restituzione speciale

alori di restituzione

alue (attributo)

alutazione in corto circuito

ar (parola chiave)

ariabile contatore

ariabili

correlazioni

da parametri

debugging

definizione

for (ciclo)

funzioni come

globali

locali

oggetti

presentazione  
ector (tipo)  
ettore (esercizio)  
iew (tipo)  
irgola (carattere)  
irgolette doppie (carattere)  
irgolette semplici (carattere)  
irgolette  
irus  
ita artificiale  
ite (esercizio)  
ocabolario  
oid (operatore)

W  
all (tipo)  
allFollower (tipo)  
eb browser, *vedere* browser  
eb worker  
eb, *vedere* World Wide Web  
ebDAV  
ebgl (contesto per Foglio)  
ebSockets  
hich (proprietà)  
hile (ciclo)  
*Why's Poignant Guide to Ruby*  
idth (CSS)  
indow variable  
indows  
ithContext (funzione)  
ord (carattere)  
orld (tipo)

World Wide Web

rite (metodo)

riteFile (funzione)

riteHead (metodo)

WWW, vedere World Wide Web

## X

ML

MLHttpRequest,

mlns (attributo)

## Y

ield (parola riservata)

## Z

awinski, Jamie

eroPad (funzione)

## **Circa l'autore**

Marijn Haverbeke è un autore e programmatore indipendente, che si interessa principalmente di linguaggi di programmazione e strumenti per programmatori. Passa gran parte del suo tempo a lavorare su software open source, tra cui l'editor CodeMirror e il motore di inferenza di tipo Tern.

# **Informazioni sul Libro**

# **Padroneggia il linguaggio del Web!**

JavaScript è il motore di quasi tutte le applicazioni web più moderne, dalle app social ai giochi per browser più all'avanguardia. Facile da usare, anche per i principianti, JavaScript è tuttavia un linguaggio flessibile e complesso, che consente di realizzare applicazioni davvero potenti.

Questa guida completa a JavaScript scava nel profondo del linguaggio per mostrare come scrivere un codice elegante ed efficiente. L'autore vi guida fin dall'inizio con esempi, esercizi e interi capitoli su progetti specifici, facendovi fare esperienza sulla realizzazione di programmi completi.

L'opera riflette lo stato attuale di JavaScript e dei browser web, ed è arricchita di nuovi materiali, come un capitolo dedicato al rendimento del codice in JavaScript e una descrizione dettagliata di concetti come la ricorsività e le chiusure.

Tutti i sorgenti sono disponibili online in uno spazio protetto interattivo, dove è possibile modificare, eseguire e verificare istantaneamente i risultati del codice.