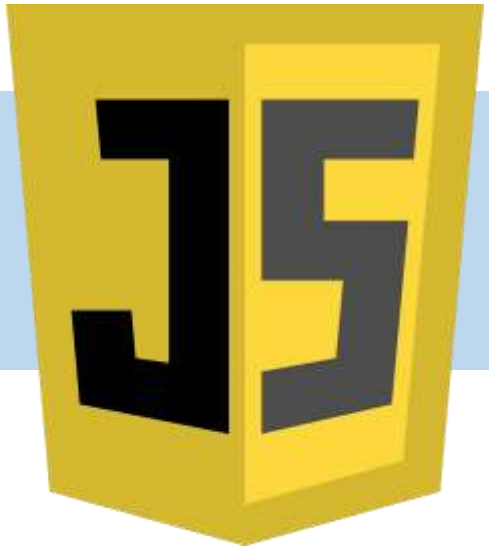
Corso
Analista | Programmatore



Javascript

Giuseppe Solla

DOCENTE

da Wikipedia - 1

In informatica **JavaScript** è un linguaggio di programmazione multi paradigma orientato agli eventi, comunemente utilizzato nella programmazione Web lato client (esteso poi anche al lato server) per la creazione, in siti web e applicazioni web, di effetti dinamici interattivi tramite funzioni di script invocate da *eventi* innescati a loro volta in vari modi dall'utente sulla pagina web in uso (mouse, tastiera, caricamento della pagina ecc...).

da Wikipedia - 2

Originariamente sviluppato da Brendan Eich della Netscape Communications con il nome di **Mochan** e successivamente di **LiveScript**, in seguito è stato rinominato "*JavaScript*" ed è stato formalizzato con una sintassi più vicina a quella del linguaggio Java di Sun Microsystems (che nel 2010 è stata acquistata da Oracle). Standardizzato per la prima volta il 1997 dalla ECMA con il nome ECMAScript, l'ultimo standard, di giugno 2022, è ECMA-262 Edition 13 ed è anche uno standard ISO (ISO/IEC 16262).

Introduzione

JavaScript è uno dei linguaggi di programmazione più popolari al mondo e rappresenta certamente un'ottima scelta come primo linguaggio per imparare a programmare.
Utilizziamo JavaScript principalmente per creare

- siti web
- applicazioni web
- Applicazioni server-side usando Node.js

ma JavaScript non si limita a queste cose e può anche essere usato per

- creare applicazioni per dispositivi mobili tramite strumenti come React Native
- creare programmi per microcontrollori and the internet of things (IoT)
- creare applicazioni per smartwatch

In pratica, può essere usato per qualsiasi cosa.

E’ un linguaggio di programmazione

Ad alto livello: fornisce astrazioni che ti permettono di ignorare i dettagli della macchina durante l'esecuzione. Gestisce automaticamente la memoria tramite un garbage collector, quindi puoi concentrarti sul codice, senza preoccuparti troppo della gestione della memoria come in altri linguaggio tra cui C, e inoltre è dotato di svariati costrutti che ti permettono di aver a che fare con variabili e oggetti estremamente importanti.

Si intende per garbage collector una modalità automatica di gestione della memoria, mediante la quale un sistema operativo, o un compilatore e un modulo di run-time liberano porzioni di memoria non più utilizzate dalle applicazioni. In altre parole, il *garbage collector* annoterà le aree di memoria non più *referenziate*, cioè allocate da un processo attivo, e le libererà automaticamente. La garbage collection è stata inventata nel 1959 da John McCarthy per il linguaggio di programmazione Lisp.

E’ un linguaggio di programmazione

Dinamico: in opposizione ai linguaggi di programmazione statici, un linguaggio dinamico esegue nel runtime molti processi che un linguaggio statico svolge durante il compile-time.

Questo si riflette in svariati pro e contro e da ciò derivano caratteristiche di notevole importanza tra cui la tipizzazione dinamica, il binding dinamico, la riflessione, la programmazione funzionale, l'alterazione del runtime di un oggetto, la chiusura e molto altro.

In informatica il **binding** è il processo tramite cui viene effettuato il collegamento fra una entità di un software ed il suo corrispettivo valore.

E’ un linguaggio di programmazione

Dinamicamente tipizzato: il tipo di dato contenuto in una variabile non è fissato, quindi puoi riassegnare a una variabile qualsiasi tipo di dato, ad esempio, un intero a una variabile che contiene una stringa.

E’ un linguaggio di programmazione

Debolmente tipizzato: in opposizione alla tipizzazione forte, nei linguaggi debolmente tipizzati il tipo di oggetto non è fisso, offrendo una maggiore flessibilità impedendo l'uso di type safety (sicurezza rispetto ai tipi) e type checking (fornito da TypeScript - che è sviluppato a partire da JavaScript).

TypeScript ha nel suo nome la sua missione: fornire "un JavaScript con il supporto opzionale della tipizzazione stretta" e la cui compilazione genera codice JavaScript standard. Una soluzione ideale per i professionisti che, abituati al tradizionale approccio orientato agli oggetti (con linguaggi come Java, C++ o C#) soffrono il passaggio a JavaScript per l'assenza della gestione dei tipi di dato.

E’ un linguaggio di programmazione

Interpretato: è comunemente conosciuto come linguaggio interpretato, che vuol dire che non necessita di una fase di compilazione prima dell'esecuzione del programma, al contrario di C, Java o Go per esempio.

In pratica, per una questione di performance, i browser compilano JavaScript prima di eseguirlo - senza bisogno di nessun un altro step.

E’ un linguaggio di programmazione

Multi-paradigma: il linguaggio non è vincolato da nessun tipo particolare di paradigma di programmazione, al contrario di Java che, ad esempio, impone l'uso della programmazione orientata agli oggetti, oppure C che obbliga alla programmazione imperativa.

Puoi scrivere in JavaScript utilizzando un paradigma orientato agli oggetti usando prototipi e le nuove classi di sintassi (come ES6).

Puoi scrivere in JavaScript in uno stile di programmazione funzionale, con le sue funzioni di prima classe o anche in uno stile imperativo (come in C).

E’ un linguaggio di programmazione

Nel caso ve lo stiate chiedendo,

JavaScript non ha niente a che fare con Java,

la somiglianza è soltanto legata alla scelta del nome e dobbiamo conviverci.

Un po’ di storia

JavaScript è stato creato nel 1995 ed è arrivato molto lontano, nonostante le sue umili origini.

È stato il primo linguaggio di scripting ad essere supportato dagli stessi browser, e grazie a ciò è diventato estremamente competitivo rispetto agli altri linguaggi, essendo tutt'ora l'unico linguaggio di scripting che possiamo usare per creare applicazioni web.

Altri linguaggi esistono, ma devono essere compilati in JavaScript - o più recentemente in WebAssembly.

Ai suoi inizi, JavaScript non era neanche lontanamente importante come lo è oggi ed era principalmente usato per animazione di fantasia e per quella meraviglia conosciuta al tempo come HTML dinamico.

Con le crescenti necessità che le piattaforme web richiedevano (e continuano a richiedere), JavaScript aveva la responsabilità di crescere per andare incontro ai bisogni di uno dei sistemi più utilizzati al mondo.

JavaScript anche oggi viene ampiamente utilizzato fuori dal browser. L'ascesa di Node.js negli ultimi anni ha sbloccato lo sviluppo backend, una volta dominio di Java, Ruby, Python, PHP e dei più tradizionali linguaggi server-side.

JavaScript è anche un linguaggio utilizzato per database e molte altre applicazioni ed è anche utilizzabile per lo sviluppo di applicazioni integrate, app mobili, app TV e molto altro. Quello che è nato come un linguaggio di nicchia all'interno del browser, adesso è il linguaggio più popolare al mondo.

WebAssembly

WebAssembly (**Wasm**, **WA**) è uno standard web che definisce un formato binario e un corrispondente formato testuale per la scrittura di codice eseguibile nelle pagine web. Ha lo scopo di abilitare l'esecuzione del codice quasi alla stessa velocità con cui esegue il codice macchina nativo. È stato progettato come integrazione di JavaScript per accelerare le prestazioni delle parti critiche delle applicazioni Web e in seguito per consentire lo sviluppo web in altri linguaggi oltre a JavaScript. È sviluppato dal World Wide Web Consortium (W3C) con ingegneri provenienti da Mozilla, Microsoft, Google e Apple.

Viene eseguito in una sandbox nel browser Web dopo una fase di verifica formale. I programmi possono essere compilati da linguaggi di alto livello in moduli Wasm e caricati come librerie dalle applet JavaScript (Da Wikipedia).

Breve introduzione alla sintassi

In questa piccola introduzione voglio illustrarvi 5 concetti:

- `spazi vuoti`
- `case sensitivity`
- `literal`
- `identificatori`
- `commenti`

Breve introduzione alla sintassi

Spazi vuoti

JavaScript non considera gli spazi vuoti. Spazi e interruzioni di riga possono essere aggiunti in qualsiasi modo preferisci (almeno in teoria).

In pratica, tenderai a seguire uno stile ben definito, attenendoti a quello comunemente utilizzato, applicandolo tramite un linter o uno strumento di stile come *Prettier*.

Ad esempio, io utilizzo sempre due spazi per ogni indentazione.

Case sensitivity

JavaScript è case sensitive (sensibile alle maiuscole) e una variabile chiamata `something` è diversa da `Something`.

La stessa cosa è valida per qualsiasi identificatore.

Breve introduzione alla sintassi

Literal

Definiamo un **literal** come un valore scritto nel codice sorgente, ad esempio, un numero, una stringa, un booleano o anche un costrutto avanzato come un Object Literal o un Array Literal:

```
5
'Test'
true
['a', 'b']
{color: 'red', shape: 'Rectangle'}
```

Breve introduzione alla sintassi

Un **identificatore** è una sequenza di caratteri che viene utilizzata per identificare una variabile, una funzione o un oggetto. Può iniziare con una lettera, il simbolo del dollaro $ o un underscore _ e contenere cifre. Utilizzando Unicode, una lettera può essere un qualsiasi carattere consentito, ad esempio una emoji.

```
Test
test
TEST
_test
Test1
$test
```

Il simbolo del dollaro è comunemente usato per fare riferimento a elementi DOM (qualunque elemento della pagina caratterizzato da un tag). Alcuni nomi sono riservati per l'uso interno di JavaScript e non possiamo usarli come identificatori.

Breve introduzione alla sintassi

Commenti

I commenti sono una parte estremamente importante di ogni programma, in qualsiasi linguaggio di programmazione. Sono importanti perché ci permettono di inserire annotazioni e aggiungere informazioni importanti che altrimenti non sarebbero disponibili per altre persone durante la lettura del codice. In JavaScript, possiamo scrivere un commento su una riga singola usando //. Tutto ciò che è presente dopo // non viene considerato come codice dall'interprete di JavaScript.

```
// a comment
true // another comment
```

Un altro tipo di commento è quello multi-riga, che inizia con /* e finisce con */.

```
/* some kind
of comment
*/
```

Punto e virgola

Ogni riga di un programma in JavaScript viene terminata facoltativamente usando il punto e virgola.

Facoltativamente perché l'interprete di JavaScript è abbastanza intelligente da introdurre il punto e virgola per te.

Nella maggior parte dei casi, puoi omettere del tutto il punto e virgola nei tuoi programmi senza neanche pensarci.

Questo è un aspetto piuttosto controverso e alcuni sviluppatori utilizzano sempre il punto e virgola, mentre altri non ne fanno mai uso, così avrai sempre a che fare con del codice in cui è presente e altro in cui viene omesso.

I valori

Una stringa `hello` è un **valore**.

Un numero come il `12` è un **valore**.

hello e 12 sono valori. `string` e `number` costituiscono il **tipo** di questi valori.

Il **tipo** è il genere di un valore, la sua categoria. In JavaScript abbiamo molti tipi di valori differenti, ognuno con le proprie caratteristiche e di cui parleremo più avanti nel dettaglio.

Quando abbiamo bisogno di avere un riferimento per un valore, lo assegniamo ad una **variabile**.

La variabile possiede un nome e il valore è ciò è contenuto nella variabile, quindi possiamo avere accesso al valore tramite il nome della variabile.

Le variabili

Una variabile è un valore assegnato ad un identificatore, a cui ci si può riferire per utilizzare la variabile all'interno del programma. Questo accade perché JavaScript è **debolmente tipizzato**, un concetto di cui sentirai parlare spesso. Prima di poter essere utilizzata, una variabile deve essere dichiarata. Ci sono due modi principali per dichiarare le variabili.

Il primo è l'uso di `const`:

```
const a = 0
```

Il secondo è `let`:

```
let a = 0
```

Qual è la differenza? `const` definisce un riferimento costante per un valore. Questo vuol dire che il riferimento non può essere cambiato, che non puoi riassegnargli un nuovo valore. Invece, usando `let` puoi assegnargli un nuovo valore.

Le variabili

Ad esempio, non puoi fare questo:

```
const a = 0
a = 1
```

In questo caso, otterrai l'errore: TypeError: Assignment to constant variable.

Al contrario, puoi farlo usando let:

```
let a = 0
a = 1
```

const non significa "costante" nel senso che può avere in altri linguaggi come C.

In particolare, non significa che il valore non può cambiare ma vuol dire che non può essere riassegnato.

Se la variabile rimanda a un oggetto o un array (parleremo più avanti di oggetti e array) il contenuto dell'oggetto o dell'array può variare liberamente.

Le variabili

Le variabili definite con const devono essere inizializzate nel momento della dichiarazione:

```
const a = 0
```

Quelle definite tramite let possono essere inizializzate in un secondo momento:

```
let a
a = 0
```

Puoi dichiarare variabili multiple con una sola istruzione:

```
const a = 1, b = 2
let c = 1, d = 2
```

Ma non puoi ridichiarare la stessa variabile più di una volta:

```
let a = 1
let a = 2
```

O otterrai l'errore "duplicate declaration".

Le variabili

Il mio consiglio è di utilizzare sempre const e utilizzare let soltanto quando sai che avrai bisogno di riassegnare il valore a una variabile, perché meno potere ha il tuo codice e meglio è.

Un valore che non può essere riassegnato costituisce una potenziale fonte di bug in meno.

Dopo aver visto come funzionano const e let, occorre parlare anche di `var`.

Prima del 2015, var era l'unico modo per dichiarare una variabile in JavaScript. Oggi, un moderno codebase conterrà con ogni probabilità soltanto const e let.

Le dichiarazioni con var hanno ambito globale o di funzione, mentre le dichiarazioni con let e const hanno ambito di blocco. Le variabili var possono essere aggiornate o ri-dichiarate dentro il loro ambito; le variabili let possono essere aggiornate ma non re-dichiarate; le variabili const non possono essere né aggiornate né re-dichiarate.

Ambito di var

Ambito (scope in inglese) significa dove queste variabili sono disponibili per essere usate. Le dichiarazioni con var hanno ambito globale, o della funzione/locale.

L'ambito è globale quando una variabile è dichiarata con var al di fuori di una funzione.

Questo significa che qualsiasi variabile dichiarata con var al di fuori del blocco di una funzione è disponibile per essere usata nell'intera finestra.

var ha l'ambito della funzione quando è dichiarato dentro una funzione.

Questo significa che è disponibile e può essere usato solo dentro la funzione.

Hoisting di var

L'hoisting è un meccanismo di JavaScript dove la dichiarazione di variabili e funzioni è spostato all'inizio del loro ambito prima dell'esecuzione del codice.

Questo significa che se facciamo:

```
console.log (greeter);
var greeter = "say hello"
```

è interpretato come

```
var greeter;
console.log(greeter); // greeter is undefined
greeter = "say hello"
```

Quindi variabili var sono issate all'inizio del loro ambito e inizializzate con un valore di undefined.

Le variabili

Le variabili in JavaScript non sono associate a un tipo particolare (sono *untyped)*.

Una volta assegnato un valore di qualche tipo a una variabile, puoi riassegnare la variabile per contenere un valore di qualsiasi altro tipo senza problemi.

Primitivi e Oggetti

In JavaScript esistono principalmente due tipi: **primitivi** e **oggetti**.

Primitivi

I tipi primitivi sono:

- numeri
- stringhe
- booleani
- simboli

E due tipi speciali: null e undefined.

Oggetti

Qualsiasi valore che non è primitivo (una stringa, un numero, un booleano, null o undefined) è un **oggetto**. Gli oggetti hanno **proprietà** e anche **metodi** che possono agire su di esse.

Primitivi e Oggetti

Un'espressione è una singola unità di codice JavaScript che il motore JavaScript può valutare per restituire un valore.

Le espressioni hanno una complessità variabile. Iniziamo con le più semplici, chiamate espressioni primarie:

```
2
0.02
'something’
true
false
this  //the current scope
undefined
i  //where i is a variable or a constant
```

Primitivi e Oggetti

Le espressioni aritmetiche sono espressioni che prendono una variabile e un operatore (o più operatori a breve) e restituiscono un numero:

```
1 / 2
i++
i -= 2
i * 2
```

Le espressioni possono anche restituire delle stringhe:

```
'A ' + 'string'
```

Le espressioni logiche fanno uso di operatori logici e restituiscono un valore booleano:

```
a && b
a || b
!a
```

Espressioni più avanzate includono oggetti, funzioni e array, e le introdurremo più avanti.

Gli operatori

Gli operatori ti permettono di ottenere espressioni complesse combinando due espressioni semplici.
Possiamo classificare gli operatori in base agli operandi su cui agiscono. Alcuni operatori lavorano con 1 operando, ma la maggior parte ne usa 2, mentre solo un operatore lavora con 3 operandi.
In questa prima introduzione agli operatori, inizieremo a parlare di quelli che ci sono più familiari: gli operatori con 2 operandi.
Ne ho già introdotto uno, parlando delle variabili: l'operatore di assegnazione =, che viene utilizzato per assegnare un valore a una variabile.

```
let b = 2
```

Adesso, occupiamoci di un altro set di operatori binari con cui hai familiarità dalla matematica di base.

Gli operatori

L'operatore di addizione (+)

```
const three = 1 + 2
const four = three + 1
```

L'operatore + agisce anche sulle stringhe, concatenandole, quindi fai attenzione:

```
const three = 1 + 2
three + 1 // 4
'three' + 1 // three1
```

L'operatore di sottrazione (-)

```
const two = 4 - 2
```

Gli operatori

L'operatore di divisione (/)

Restituisce il quoziente tra il primo operando e il secondo:

```
const result = 20 / 5 //result === 4
const result = 20 / 7 //result === 2.857142857142857
```

Se dividi per zero, JavaScript non ti segnala nessun errore ma restituisce il valore Infinity (o -Infinity se il valore è negativo).

```
1 / 0 //Infinity
-1 / 0 //-Infinity
```

Gli operatori

L’operatore modulo (%)

Questo operatore restituisce il resto di una divisione e può essere molto utile in molte applicazioni:

```
const result = 20 % 5 //result === 0
const result = 20 % 7 //result === 6
```

Il resto di una divisione per zero è sempre NaN, un valore speciale che vuol dire "non un numero" ("Not a Number"):

```
1 % 0 //NaN
-1 % 0 //NaN
```

Gli operatori

L'operatore di moltiplicazione (*)

Moltiplica due numeri:

```
1 * 2 //2
-1 * 2 //-2
```

L'operatore di elevamento a potenza (**)

Eleva il primo operando alla potenza del secondo operando:

```
1 ** 2 //1
2 ** 1 //2
2 ** 2 //4
2 ** 8 //256
8 ** 2 //64
```

Ordine di priorità

Ogni istruzione complessa con operatori multipli nella stessa riga introduce un problema di priorità, ad esempio:

```
let a = 1 * 2 + 5 / 2 % 2
```

Il risultato è 2.5, ma perché? Quali operazioni vengono eseguite prima e in un momento successivo? Alcune operazioni hanno precedenza su altre.

Le regole che stabiliscono la priorità sono elencate in questa tabella:

OPERATORE	DESCRIZIONE
* / %	moltiplicazione/divisione
+ -	addizione/sottrazione
=	assegnazione

Ordine di priorità

Le operazioni dello stesso livello (come + e-) sono eseguite nell'ordine in cui vengono incontrate, da sinistra a destra.

Seguendo queste regole, l'esempio precedente si risolve in questo modo:

```
let a = 1 * 2 + 5 / 2 % 2
let a = 2 + 5 / 2 % 2
let a = 2 + 2.5 % 2
let a = 2 + 0.5
```

Ordine di priorità

Dopo l'operatore di assegnazione e gli operatori aritmetici, il terzo set di operatori che voglio introdurre è quello degli operatori di confronto.

Puoi utilizzare i seguenti operatori per comparare due numeri o due stringhe.

Gli operatori di confronto restituiscono sempre un valore booleano, cioè true o false.

Ecco gli operatori di confronto di **disuguaglianza**:

```
< significa "minore"
<= significa "minore o uguale"
> significa "maggiore"
>= significa "maggiore o uguale"
```

Esempio:

```
let a = 2
a >= 1 //true
```

Ordine di priorità

In aggiunta a questi, abbiamo 4 operatori di **uguaglianza**. Accettano due valori e restituiscono un booleano:

```
=== verifica l'uguaglianza

!== verifica la disuguaglianza
```

In JavaScript sono anche disponibili == e !=, ma suggerisco caldamente di utilizzare soltanto === e !== per evitare dei subdoli problemi.

Esercizio

Scrivi un programma che dato l'anno corrente e un anno di nascita determini:
- l'età della persona,
- quanti anni sono necessari per raggiungere i 100

Restituisca in output entrambi i risultati.

```
Esempio:
    Input: anno corrente = 2022, anno di nascita = 1990
    Output: età = 32, anni mancanti = 68
```

Scrivi un programma che dato un numero di secondi, calcoli la quantità di ore, minuti e secondi corrispondenti e poi stampi il risultato. L'output avrà solo numeri interi.

```
Esempio:
    Input: 12560
    Output: 3 ore, 29 minuti e 20 secondi.
```

Considera che in un'ora ci sono 60 minuti, in un minuto 60 secondi. Quindi quanti secondi ci sono in un'ora?

Operatore spread

Spread, in inglese, ha il significato letterale di "espandere" o "diffondere". Infatti, questo operatore ci permette di espandere un array o un oggetto, estraendone i dati in modo da utilizzarli in ambiti diversi, per esempio per passarli come parametri ad una funzione, o per creare un nuovo array.

```
function foo(a, b, c) {
  console.log(a, b, c);
}

let valori = ["Ecco", "l'operatore", "spread"];

foo(...valori); // Ecco l'operatore spread
```

In questo modo i valori dell'oggetto verranno estratti e utilizzati come se li avessimo inseriti uno per uno separandoli con una virgola.

Operatore spread

Un altro modo di utilizzare questa sintassi è quello di semplificare la creazione di array. Infatti possiamo utilizzare l'operatore spread per estrarre tutti i dati di un array inserendoli in un altro. Vediamo qui sotto un esempio:

```
let lista_a = [0, 1, 2];

let lista_b = [...lista_a, 3, 4, 5];

console.log(lista_b); // [ 0, 1, 2, 3, 4, 5 ]
```

Condizionali

Dopo aver affrontato gli operatori di confronto, possiamo passare ai condizionali.

L'istruzione `if` viene utilizzata per far prendere al programma una strada o un'altra, a seconda del risultato della valutazione di un'espressione.

Questo è l'esempio più semplice, che viene sempre eseguito:

```
if (true) {
  //do something
}
```

Al contrario, questo non viene mai eseguito:

```
if (false) {
  //do something (? never ?)
}
```

Condizionali

Il condizionale verifica l'espressione che gli fornisci come vera o falsa. Dalla valutazione di un numero si ottiene sempre true, a meno che sia zero. Le stringhe vengono sempre valutate come true, eccetto le stringhe vuote. Queste sono le regole generali di come le espressioni vengono valutate in un contesto booleano.

Hai notato le parentesi graffe? Ciò che è compreso al loro interno è chiamato **blocco** e viene utilizzato per raggruppare un insieme di varie istruzioni

Un blocco può essere inserito in qualsiasi posto puoi inserire una singola istruzione. Se hai una singola istruzione da eseguire dopo il condizionale, puoi omettere il blocco scrivendo solo l'istruzione:

```
if (true) doSomething()
```

Ma mi piace utilizzare sempre le parentesi graffe per essere più chiaro.

Condizionali

Dopo l'istruzione `if`, puoi aggiungere una seconda parte: l'istruzione `else`, che verrà eseguita se la condizione `if` risulta falsa.

```
if (true) {
  //do something
} else {
  //do something else
}
```

Dato che else accetta un'espressione, puoi annidare al suo interno un'altra istruzione if/else:

```
if (a === true) {
  //do something
} else if (b === true) {
  // do something else if
} else {
  // do something else
}
```

Esercizio (javascript)

Scrivi un programma che dato un numero di secondi, calcoli la quantità di ore, minuti e secondi corrispondenti e poi stampi il risultato. L'output avrà solo numeri interi.

```
Esempio:
    Input: 12560
    Output: 3 ore, 29 minuti e 20 secondi.
```

Considera che in un'ora ci sono 60 minuti, in un minuto 60 secondi. Quindi quanti secondi ci sono in un'ora?

Scrivi un programma che dati quattro numeri, restituisca in output il maggiore e il minore.

```
Esempio:
    Input: a = 12, b = 6, c = 4, d = 9
    Output: maggiore = 12, minore = 4
```

Considerate l'uso dell'istruzione **`if(condizione) { } else { }`**

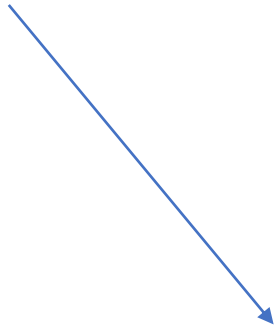
Soluzione con utilizzo delle condizioni

```
let a = 24; let b = 6; let c = 14; let d = 2;
if(a >= b && a >= c && a >= d) {
  console.log("Maggiore = " + a);
} else if (b >= a && b >= c && b >= d) {
  console.log("Maggiore = " + b);
} else if (c >= a && c >= b && c >= d) {
  console.log("Maggiore = " + c);
} else {
  console.log("Maggiore = " + d);
}
if(a <= b && a <= c && a <= d) {
  console.log("Minore = " + a);
} else if (b <= a && b <= c && b <= d) {
  console.log("Minore = " + b);
} else if (c <= a && c <= b && c <= d) {
  console.log("Minore = " + d);
} else {
  console.log("Minore = " + d);
}
```

Soluzione con array e metodi

```
// per comodità inserisco i singoli numeri all'interno di un array
let num = [3, -1, 4, -2];

// soluzione che utilizza i metodi Math.max e Math.min
console.log("Maggiore = " + Math.max(...num));
console.log("Minore = " + Math.min(...num));
```



Operatore spread

Array

Un array è un insieme di elementi. Gli array in JavaScript non costituiscono un tipo per proprio conto.

Gli array sono **oggetti**.

Possiamo inizializzare un array vuoto in 2 modi diversi:

```
const a = []
const a = Array()
```

Il primo è utilizzando la **sintassi letterale array**, mentre il secondo è tramite la funzione integrata Array().

Puoi pre-inserire elementi nell'array tramite questa sintassi:

```
const a = [1, 2, 3]
const a = Array.of(1, 2, 3)
```

Array

Un array può contenere qualsiasi valore, anche valori di diverso tipo:

```
const a = [1, 'Flavio', ['a', 'b']]
```

Siccome possiamo aggiungere un array all'interno di un altro array, possiamo creare array multi-dimensionali, che hanno applicazioni molto utili (ad esempio le matrici):

```
const matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
]
 matrix[0][0] //1
matrix[2][0] //7
```

Array

Puoi inizializzare un nuovo array con un set di valori utilizzando questa sintassi, che prima inizializza un array di 12 elementi e poi inserisce il numero 0 per ciascun elemento:

```
Array(12).fill(0)
```

Puoi ottenere il numero di elementi di un array verificando la proprietà `length`:

```
const a = [1, 2, 3]
a.length //3
```

Array

Nota che puoi impostare la lunghezza di un array. Se assegni a un array un numero più grande della sua attuale capacità non accade nulla. Se invece gli assegni un numero più piccolo, l'array viene troncato in quella posizione:

```
const a = [1, 2, 3]
a //[ 1, 2, 3 ]
a.length = 2
a //[ 1, 2 ]
```

Puoi accedere a ogni elemento di un array facendo riferimento al suo indice, partendo da zero:

```
a[0] //1
a[1] //2
a[2] //3
```

Array

Come aggiungere un elemento a un array

Possiamo aggiungere un elemento alla fine di un array usando il metodo push():

```
a.push(4)
```

O aggiungere un elemento all'inizio di un array con il metodo unshift():

```
a.unshift(0)
a.unshift(-2, -1)
```

Come rimuovere un elemento da un array

Possiamo rimuovere un elemento dalla fine di un array usando il metodo pop():

```
a.pop()
```

O rimuovere un elemento dall'inizio di un array con il metodo shift():

```
a.shift()
```

Array

Come unire due o più array

Puoi unire più array utilizzando il metodo `concat()`:

```
const a = [1, 2]
const b = [3, 4]
const c = a.concat(b) //[1,2,3,4]
a //[1,2]
b //[3,4]
```

Puoi anche utilizzare l'operatore **spread** (...), i tre puntini, in questo modo:

```
const a = [1, 2]
const b = [3, 4]
const c = [...a, ...b]
c //[1,2,3,4]
```

Array

Come trovare un valore specifico in un array

Puoi usare il metodo `find()` per trovare un elemento in un array:

```
a.find((element, index, array) => {  //return true or false })
```

Questo metodo dà come valore di ritorno il primo elemento che restituisce true, oppure undefined se l'elemento non viene trovato.

Una sintassi comune è:

```
a.find(x => x.id === my_id)
```

La riga di codice qui sopra restituisce il primo elemento dell'array che ha `id === my_id`.

`findIndex()` funziona in modo simile a `find()`, ma restituisce l'indice del primo elemento che corrisponde alla ricerca, oppure restituisce `undefined`, se non viene trovato:

```
a.findIndex((element, index, array) => {  //return true or false })
```

Array

Un altro metodo è `includes()`:

```
a.includes(value)
```

Restituisce `true` se a contiene `value`.

```
a.includes(value, i)
```

Restituisce `true` se a contiene `value` dopo la posizione `i`.

Stringhe

Una stringa è una sequenza di caratteri.

Può essere anche definita come stringa letterale, che è racchiusa tra virgolette singole o doppie:

```
'A string'

"Another string"
```

Personalmente preferisco sempre usare le virgolette singole e utilizzare le doppie soltanto in HTML per definire attributi. Puoi assegnare una stringa a una variabile, in questo modo:

```
const name = 'Flavio'
```

Puoi determinare la lunghezza di una stringa usando la sua proprietà `length`:

```
'Flavio'.length //6

const name = 'Flavio'

name.length //6
```

Stringhe

Questa è una stringa vuota: '' e la sua lunghezza è `0`:

```
''.length //0
```

Due stringhe possono essere unite con l'operatore +:

```
"A " + "string"
```

Puoi utilizzare l'operatore + per inserire delle variabili:

```
const name = 'Flavio'
"My name is " + name //My name is Flavio
```

Un altro modo di definire una stringa avviene tramite l'uso di template literal, definiti all'interno di due backtick (accento grave: `). Sono molto utili soprattutto per rendere più semplice la scrittura di stringhe su più righe.

Con le virgolette singole o doppie non è facile definire una stringa su più riga - c'è bisogno di utilizzare dei caratteri di escape.

Stringhe

Una volta aperto un template literal con il backtick, devi solo premere invio per creare una nuova riga, senza caratteri speciali:

```
const string = `Hey

this
string
is awesome!`
```

I template literal sono fantastici perché forniscono un modo semplice per inserire variabili ed espressioni all'interno di stringhe.

Stringhe

Puoi farlo usando la sintassi ${...}:

```
const v = 'test’
const string = `something ${v}`

//something test
```

All'interno di ${} puoi inserire qualsiasi cosa, persino delle espressioni:

```
const string = `something ${1 + 2 + 3}`
const string2 = `something
  ${foo() ? 'x' : 'y'}`
```

I loop

I loop sono una delle principali strutture di controllo di JavaScript.

Grazie a un loop è possibile automatizzare e ripetere l'esecuzione di un blocco di codice un numero qualsiasi di volte, anche indefinitamente.

JavaScript offre vari modi di eseguire iterazioni attraverso i loop.

Ci concentreremo su questi 3 modi:

- `while`
- `for`
- `for..of`

I loop

while

Il loop while è la struttura più semplice che JavaScript ci offre.
Aggiungiamo una condizione dopo la keyword while e un blocco di codice che verrà eseguito finché la condizione è true.

Esempio:

```
const list = ['a', 'b', 'c’]
let i = 0
while (i < list.length) {
  console.log(list[i]) //value
  console.log(i) //index
  i = i + 1
}
```

Puoi interrompere il ciclo **while** usando la keyword **break**, in questo modo:

```
while (true) {
  if (somethingIsTrue) break
}
```

I loop

Se invece decidi che durante l'esecuzione del loop vuoi saltare dall'iterazione corrente alla successiva, puoi farlo usando il comando continue:

```
while (true) {
  if (somethingIsTrue) continue
  //do something else
}
```

I cicli do..while sono molto simili ai cicli while. La differenza sta nel fatto che la condizione viene valutata *dopo* l'esecuzione del blocco di codice.

Ciò significa che il blocco viene sempre eseguito *almeno una volta*.

Esempio:

```
const list = ['a', 'b', 'c’]
let i = 0
do {
  console.log(list[i]) //value
  console.log(i) //index
  i = i + 1
} while (i < list.length)
```

I loop

for

La seconda importante struttura di iterazione in JavaScript è il loop for.

Utilizziamo la keyword for e aggiungiamo un set di 3 istruzioni: l'inizializzazione, la condizione e l'aggiornamento delle variabili.

Esempio:

```
const list = ['a', 'b', 'c']

for (let i = 0; i < list.length; i++) {
  console.log(list[i]) //value
  console.log(i) //index
}
```

Proprio come il ciclo while, puoi interrompere un loop for usando break o saltare all'iterazione successiva di un loop for tramite continue.

I loop

for...of

Questo loop è relativamente recente (è stato introdotto nel 2015) ed è una versione semplificata del loop for:

```
const list = ['a', 'b', 'c']

for (const value of list) {
  console.log(value) //value
}
```

Le funzioni

In qualsiasi programma JavaScript che presenta una discreta complessità, tutto si svolge grazie a delle funzioni.
Le funzioni costituiscono il nucleo, la parte essenziale di JavaScript.

Cos'è una funzione?
Una funzione è un blocco di codice autonomo.

Ecco un esempio di **dichiarazione di funzione**:

```
function getData() {
  // do something
}
```

Una funzione può essere eseguita in qualsiasi momento desideri, richiamandola, in questo modo:

```
getData()
```

Le funzioni

Una funzione può avere uno o più argomenti:

```
function getData() {  //do something }

function getData(color) { //do something }

function getData(color, age) { //do something }
```

Quando richiamiamo la funzione, forniamo dei parametri che fungono da argomenti:

```
getData('green', 24)

getData('black')
```

Le funzioni

Come potete notare, durante la seconda chiamata della funzione, ho fornito soltanto il parametro black come argomento di color, senza passare nessun paramentro come argomento per age.

In questo caso, age all'interno della funzione è `undefined`.

Possiamo verificare se un valore non è definito usando questo condizionale:

```
function getData(color, age) {
  //do something
  if (typeof age !== 'undefined') {
    //...
  }
}
```

`typeof` è un operatore unitario che ci permette di verificare il tipo di una variabile.

Le funzioni

Puoi anche effettuare il controllo in questo modo:

```
function getData(color, age) {

  //do something
  if (age) {
    //...
  }
}
```

Ma il condizionale sarà falso anche se age è null, `0` o una stringa vuota. Puoi impostare dei valori di default per i parametri, nel caso non vengano passati:

```
function getData(color = 'black', age = 25) {
  //do something
}
```

Puoi passare qualsiasi valore come parametro: numeri, stringhe, booleani, array, oggetti e anche funzioni.

Le funzioni

Una funzione ha un valore di ritorno. Di default una funzione restituisce `undefined`, a meno che non aggiungi la keyword `return` con un valore:

```
function getData() {
  // do something
  return 'hi!'
}
```

Quando invochiamo la funzione, possiamo anche assegnare il valore di ritorno a una variabile:

```
function getData() {
  // do something
  return 'hi!'
}

let result = getData()          //result adesso contiene una stringa con il valore hi!.
```

Le funzioni

Per avere più valori, devi utilizzare un oggetti o un array, in questo modo:

```
function getData() {
  return ['Flavio', 37]
}

let [name, age] = getData()
```

Le funzioni possono essere definite all'interno di altre funzioni:

```
const getData = () => {
  const dosomething = () => {}
  dosomething()
  return 'test'
}
```

Le funzioni annidate non possono essere chiamate dall'esterno della funzione che le racchiude. Puoi anche avere una funzione come valore di ritorno di una funzione.

Le funzioni freccia

La funzioni freccia sono state introdotte di recente in JavaScript.

Vengono utilizzate molto spesso al posto delle funzioni "ordinarie", quelle che abbiamo appena descritto nel capitolo precedente. Troverai ovunque entrambe le forme.

Visivamente, permettono di scrivere funzioni con una sintassi più breve, da:

```
function getData() {
  //...
}
```

→

```
() => {
  //...
}
```

Ma considera che queste funzioni non hanno un nome. Le funzioni freccia sono anonime e dobbiamo assegnarle a una variabile. Possiamo assegnare una funzione ordinaria a una variabile in questo modo:

```
let getData = function getData() {
  //...
}
```

Le funzioni freccia

In questa operazione, possiamo rimuovere il nome dalla funzione:

```
let getData = function() {  //... }
```

e invocare la funzione tramite il nome della variabile:

```
let getData = function() {  //... }
getData()
```

Analogamente, possiamo fare la stessa cosa per le funzioni freccia:

```
let getData = () => {  //... }
getData()
```

Le funzioni freccia

Se il corpo della funzione contiene solo una singola istruzione, possiamo omettere le parentesi e scrivere tutto su una sola riga:

```
const getData = () => console.log('hi!')
```

I parametri vengono passati all'interno delle parentesi:

```
const getData = (param1, param2) =>
  console.log(param1, param2)
```

Se la funzione ha un parametro (soltanto uno), puoi omettere del tutto le parentesi:

```
const getData = param => console.log(param)
```

Le funzioni freccia ti permettono di avere un return implicito - i valori vengono restituiti senza bisogno di usare la keyword `return`.

Le funzioni freccia

Funziona quando abbiamo una istruzione su una riga nel corpo della funzione:

```
const getData = () => 'test’
getData() //'test’
```

Come nelle funzioni ordinarie, possiamo avere dei valori di default per i parametri nel caso in cui non vengano passati:

```
const getData = (color = 'black', age = 2) => {

  //do something
}
```

E come le funzioni regolari, abbiamo soltanto un valore di ritorno. Le funzioni freccia possono anche contenere altre funzioni freccia o anche funzioni regolari. Questi due tipi di funzioni sono molto simili, la differenza più consistente rispetto alle funzioni ordinarie è evidente quando vengono usate come metodi per oggetti.

Oggetti

Qualsiasi valore che non appartiene a un tipo primitivo (una stringa, un numero, un booleano, un simbolo, null o undefined) è un **oggetto**.

Ecco come definiamo un oggetto:

```
const car = {
}
```

Questa è la **sintassi letterale di un oggetto**, una delle cose migliori in JavaScript.

Puoi anche utilizzare la sintassi new Object:

```
const car = new Object()
```

Un'altra sintassi è Object.create():

```
const car = Object.create()
```

Oggetti

Puoi anche inizializzare un oggetto usando la keyword new prima della funzione con la lettera maiuscola. Questa funzione agisce come costruttore per un oggetto. Al suo interno, possiamo inizializzare gli argomenti come parametri, per impostare lo stato iniziale dell'oggetto:

```
function Car(brand, model) {
  this.brand = brand
  this.model = model
}
```

Inizializziamo un nuovo oggetto in questo modo:

```
const myCar = new Car('Ford', 'Fiesta')
myCar.brand //'Ford'
myCar.model //'Fiesta'
```

Gli argomenti degli oggetti sono **sempre passati per riferimento**.

Oggetti

Se assegni a una variabile lo stesso valore di un'altra, nel caso sia di tipo primitivo come un numero o una stringa, viene passato per valore:

```
let age = 36
let myAge = age
myAge = 37
age //36
const car = {
  color: 'blue'
}
const anotherCar = car
anotherCar.color = 'yellow'
car.color //'yellow'
```

Anche gli array e le funzioni, sotto sotto sono degli oggetti, quindi è molto importante capire come funzionano.

Proprietà degli oggetti

Gli oggetti hanno **proprietà**, che sono formate da un'etichetta associata a un valore.

Il valore di una proprietà può essere di qualsiasi tipo: può essere un array, una funzione o addirittura un oggetto, in quanto gli oggetti possono essere annidati in altri oggetti.

Questa è la sintassi letterale per gli oggetti che abbiamo visto nel capitolo precedente:

```
const car = {
 }
```

Possiamo definire la proprietà color in questo modo:

```
const car = {
  color: 'blue'
}
```

Abbiamo l'oggetto car con una proprietà chiamata color, che ha valore blue.

Proprietà degli oggetti

Le etichette possono essere rappresentate da qualsiasi stringa, ma attenzione ai caratteri speciali - se vuoi includere un carattere non valido come nome di una variabile nel nome della proprietà, devi racchiuderlo tra virgolette:

```
const car = {
  color: 'blue',
  'the color': 'blue'
}
```

Tra i caratteri non validi per il nome di variabili ci sono spazi, trattini e altri caratteri speciali. Come puoi vedere, quando abbiamo più proprietà, dobbiamo separarle con una virgola. Possiamo recuperare il valore di una proprietà usando 2 sintassi diverse.

La prima è la **dot notation**:

```
car.color //'blue'
```

Proprietà degli oggetti

La seconda (che è la sola che possiamo utilizzare per le proprietà con nomi non validi) si avvale delle parentesi quadre:

```
car['the color'] //'blue’
```

Se provi ad accedere a una proprietà inesistente, otterrai il valore `undefined`:

```
car.brand //undefined
```

Come detto in precedenza, gli oggetti possono avere oggetti annidati come proprietà:

```
const car = {
  brand: { name: 'Ford' },
  color: 'blue'
}
```

In quest'esempio, puoi avere accesso al nome `name` del `brand` usando:

`car.brand.name` oppure `car['brand']['name']`

Proprietà degli oggetti

Puoi impostare il valore di una proprietà quando definisci l'oggetto.

Ma puoi aggiornarlo in un qualsiasi momento:

```
const car = {
  color: 'blue'
}
car.color = 'yellow'
car['color'] = 'red’
```

E puoi anche aggiungere nuove proprietà a un oggetto:

```
car.model = 'Fiesta'
car.model //'Fiesta'
```

Proprietà degli oggetti

Considerando l'oggetto:

```
const car = {
  color: 'blue',
  brand: 'Ford'
}
```

Puoi eliminare una proprietà dell'oggetto tramite:

```
delete car.brand
```

Metodi per gli oggetti

Abbiamo parlato delle funzioni in un capitolo precedente.

Le funzioni possono essere assegnate alle proprietà di un oggetto e in questo caso vengono chiamate **metodi**.

In quest'esempio, la proprietà start ha una funzione assegnata e possiamo chiamarla usando la sintassi per le proprietà con il punto e le parentesi alla fine:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
  start: function() {
    console.log('Started')
  }
}
car.start()
```

All'interno del metodo definito usando la sintassi `function() {},` abbiamo accesso alle istanze dell'oggetto facendo riferimento a `this`.

Metodi per gli oggetti

Nel prossimo esempio, accediamo ai valori delle proprietà `brand` e `model` usando `this.brand e this.model`:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
  start: function() {
    console.log(`Started
      ${this.brand} ${this.model}`)
  }
}
 car.start()
```

Metodi per gli oggetti

È importante notare la distinzione tra le funzioni ordinarie e le funzioni freccia - non possiamo usare `this` con le funzioni freccia:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
  start: () => {
    console.log(`Started
      ${this.brand} ${this.model}`) //not going to work
  }
}
 car.start()
```

Questo accade perché **le funzioni freccia non sono legate all'oggetto**. Questa è la ragione per cui le funzioni ordinarie vengono spesso usate come metodi per oggetti.

Metodi per gli oggetti

I metodi accettano parametri, come le funzioni ordinarie:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
  goTo: function(destination) {
    console.log(`Going to ${destination}`)
  }
}
 car.goTo('Rome')
```

Le classi

Abbiamo parlato di oggetti, che sono una delle parti più interessanti di JavaScript.

In questo capitolo, saliremo di livello introducendo le classi.

Cosa sono le classi? Sono un modo per definire un pattern comune a più oggetti.

Consideriamo l'oggetto person:

```
const person = {
  name: 'Flavio'
}
```

Creiamo una classe chiamata `Person` (nota la P maiuscola, una convenzione usata per le classi), che possiede una proprietà name:

```
class Person {
  name
}
```

Le classi

Da questa classe, inizializziamo un oggetto flavio, in questo modo:

```
const flavio = new Person()
```

`flavio` è detto *istanza* della classe `Person`.

Possiamo modificare il valore della proprietà name:

```
flavio.name = 'Flavio'
```

e accedervi tramite:

```
flavio.name
```

come facciamo per le proprietà di un oggetto.

Le classi contengono proprietà, come name, e metodi.

Le classi

I metodi vengono definiti in questo modo:

```
class Person {
  hello() {
    return 'Hello, I am Flavio'
  }
}
```

E possiamo invocare un metodo su un'istanza della classe:

```
class Person {
  hello() {
    return 'Hello, I am Flavio'
  }
}
const flavio = new Person()
flavio.hello()
```

Le classi

Esiste un metodo particolare, chiamato constructor() che possiamo usare per inizializzare le proprietà di una classe quando creiamo una nuova istanza di un oggetto.

Funziona in questo modo:

```
class Person {
  constructor(name) {
    this.name = name
  }
 hello() {
    return 'Hello, I am ' + this.name + '.'
  }
}
```

Nota come utilizziamo `this` per accedere all'istanza dell'oggetto.

Le classi

Ora possiamo creare una nuova istanza della classe, passare una stringa e una volta chiamata hello otterremo un messaggio personalizzato:

```
const flavio = new Person('flavio')
flavio.hello() //'Hello, I am flavio.'
```

Quando l'oggetto è inizializzato, il metodo constructor viene chiamato con qualsiasi parametro passato. Di norma i metodi vengono definiti su un'istanza di un oggetto e non sulla classe. Possiamo definire un metodo come `static` per far sì che possa essere eseguito sulla classe:

```
class Person {
  static genericHello() {
    return 'Hello'
  }
}
Person.genericHello() //Hello
```

Ereditarietà

Una classe può estendere un'altra classe e gli oggetti inizializzati usando quella classe ereditano tutti i metodi di entrambe le classi.

Supponiamo di avere una classe Person:

```
class Person {
  hello() {
    return 'Hello, I am a Person'
  }
}
```

Definiamo una nuova classe, Programmer, che estende Person:

```
class Programmer extends Person {
}
```

Se creiamo un'istanza della classe Programmer, avrà accesso al metodo hello():

```
const flavio = new Programmer()
flavio.hello() //'Hello, I am a Person'
```

Ereditarietà

All'interno di una classe figlia, puoi far riferimento alla classe genitrice tramite super():

```
class Programmer extends Person {
  hello() {
    return super.hello() +
      '. I am also a programmer.'
  }
}
const flavio = new Programmer()
flavio.hello()
```

Il programma qui sopra restituisce Hello, I am a Person. I am also a programmer..

Termine della lezione

Grazie per la vostra partecipazione

Esercitazione

Scrivi una funzione di uguaglianza che prenda in input due argomenti e restituisca TRUE se i due argomenti sono IDENTICI, FALSE altrimenti.

Esempi:

Input: n = 2, m = 3
Output: FALSE

Input: n = 2, m = '2'
Output: FALSE

Input: n = 2, m = 2
Output: TRUE

Esercitazione

Scrivi un programma che dato un numero intero compreso tra 1 a 7 visualizzi il corrispondente giorno della settimana. Sapendo che:

1 = lunedì
2 = martedì
3 = mercoledì
...
7 = domenica

Utilizza il costrutto tipo if e prevedi anche il caso in cui il valore immesso non sia valido (nel caso stampare un messaggio di errore a tua scelta).

Esempi:

Input: numero = 6
Output: "Sabato"

Input: numero = 10
Output: "Errore! Giorno della settimana non valido!"

Soluzione

```
// Soluzione1
var numero = prompt("Scrivi in numero il giorno della settimana");
if(numero == 1) alert("Lunedi");
else if(numero == 2) alert("Martedi");
else if(numero == 3) alert("Mercoledi");
else if(numero == 4) alert("Giovedi");
else if(numero == 5) alert("Venerdi");
else if(numero == 6) alert("Sabato");
else if(numero == 7) alert("Domenica");
else alert("Valore errato");

// Soluzione2
let settimana = Array("Lunedi","Martedi","Mercoledi","Giovedi","Venerdi","Sabato","Domenica");
let numero = prompt("Scrivi in numero il giorno della settimana");
if(numero > 0 && numero < 8){
        for(var i=0; i<settimana.length; i++){
                if(i == numero - 1){
                        alert(settimana[i]);
                }
        }
} else {
        alert("Valore errato");
}
```

Esercitazione

Scrivere un documento HTML contenente una form contenente i seguenti campi:

- cognome e nome (casella di testo editabile lunga 50 caratteri)
- sesso (selezionabile tramite due bottoni radio)
- ateneo (da scegliere da una combobox che riporta delle università)
- CAP (casella di testo editabile lunga 5 caratteri)
- studente lavoratore (selezionabile tramite checkbox)
- descrizione del lavoro svolto (casella di testo editabile lunga 80 caratteri)
- bottone di <invio>
- bottone di <reset>

Aggiungere al documento HTML una funzione JavaScript che esegue i seguenti controlli:

- ❑ mentre l'utente edita i campi della form:
  - subito dopo che l'utente ha editato il campo CAP, deve verificare che tale campo sia un numero di 5 cifre;
  - subito dopo che l'utente ha editato il campo nome e cognome, deve verificare che tale campo non sia un numero;
- ❑ al momento dell'invio del messaggio:
  - ✓ verifica che il cognome e nome non sia vuoto;
  - ✓ verifica che il sesso sia stato selezionato;
  - ✓ verifica che sia stato selezionato un ateneo;
  - ✓ se l'utente ha dichiarato di essere uno studente lavoratore, la descrizione del lavoro svolto non può essere vuota.