Lo sviluppo delle competenze di sviluppatore front-end

Hello!

Alessandro Pasqualini

Full-stack developer, appassionato di programmazione e sviluppo software in generale, amo le serie tv e i gatti

1

Javascript

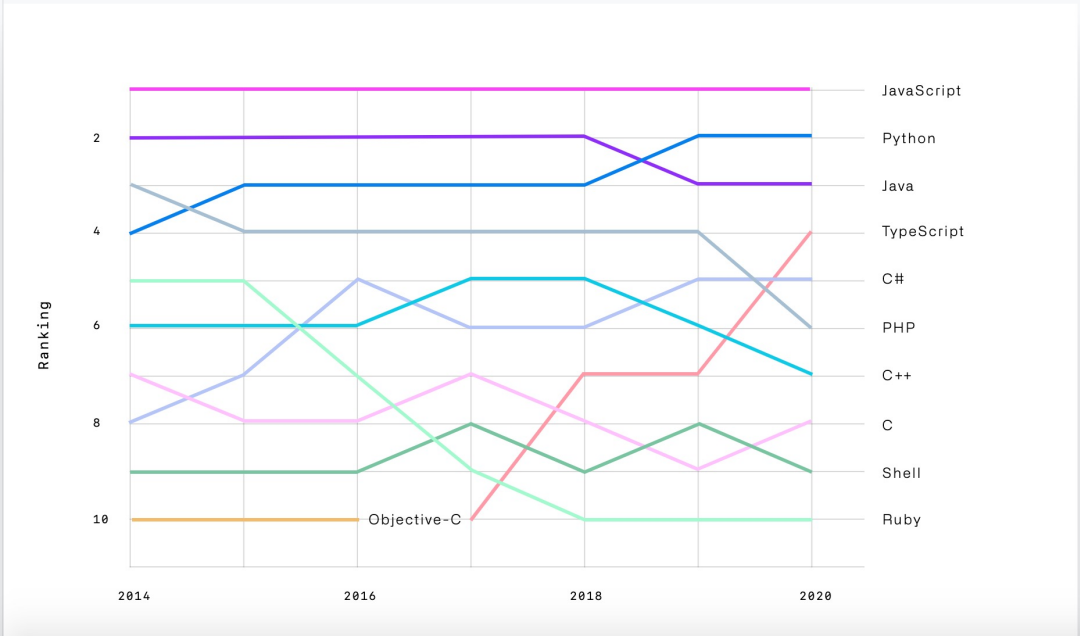
Cos'è Javascript?

Javascript è un **linguaggio di programmazione** che permette di aggiungere interattività e comportamenti personalizzati alla pagina web.

Le tecnologie del web:

- HTML definisce la struttura della pagina
- CSS definisce lo stile della pagina
- JS definisce il comportamento personalizzato della pagina

Cos'è Javascript?



https://octoverse.github.com/

Cos'è Javascript?

01	microsoft/vscode	19.1k
02	MicrosoftDocs/azure-docs	14k
03	flutter/flutter	13k
04	firstcontributions/first-contributions	11.6k
05	tensorflow/tensorflow	9.9k
06	facebook/react-native	9.1k
07	kubernetes/kubernetes	6.9k
08	DefinitelyTyped/DefinitelyTyped	6.9k
09	ansible/ansible	6.8k
10	home-assistant/home-assistant	6.3k

https://octoverse.github.com/

Javascript lato server

Javascript è uno dei linguaggi più usati al mondo.

"Di recente" è stato "trasformato" anche in un linguaggio lato server: **nodejs.**

Nodejs permette di usare le stesse tecnologie nate per il web per realizzare la cosiddetta business logic dell'applicazione, ovvero il backend.

Java is to Javascript what Car is to Carpet

- Chris Heilmann

Javascript vs Java

Java è un linguaggio completamente diverso da Javascript.

L'unica cosa che hanno in comune è una parte del nome.

Javascript ha avuto negli anni una brutta reputazione: era il responsabile di redirect, popup e il fulcro di tantissimi problemi di sicurezza.

I browser moderni controllano e impediscono a JS tutta una serie di "attività" per la sicurezza dell'utente.

Cosa si può fare con JS?

Accedere al contenuto: selezionare un elemento, accedere alle sue proprità, accedere al CSS, accedere al contenuto, etc

Modificare un elemento: modificare il contenuto di un elemento, creare elementi "on the fly", cambiare le sue proprità, cambiare i suoi attributi, cambiare il CSS, etc

Reagire ad eventi: creare del codice che deve essere eseguito se succede qualcosa (evento). Es. il click del mouse, l'hover, drag & drop etc

Per cosa è usato JS?

Validazione di form: validare il contenuto inserito dall'utente prima di inviare il contenuto al server. (c'è sempre bisogno della validazione anche nel server)

Slideshow/animazioni: mostrare contenuto diverso nello stesso spazio, animare **semplicemente** il contento. (Le animazioni posso essere fatte anche con CSS3 ma è più complicato)

Etc...

Per cosa è usato JS?

Ricaricare parte (o tutta) la pagina in modo "asincrono": caricare il contenuto senza che l'utente debba attendere il "refresh della pagina" (Single Page Application)

Filtrare i dati: filtrare i dati nella pagina in base a delle "indicazioni" dell'utente

Testare il browser: ogni browser è diverso e con JS possiamo capire quale browser l'utente sta usando e agire di conseguenza per correggere eventuali problemi

Inserire JS nella pagine

```
<script>
   // Il mio JS
</script>

<script src="…"></script>
```

Javascript può essere includo in due modi:

- **Embedded** nella pagina (similarmente al tag style)
- Inserendo il link del file JS (similarmente al tag link)

Dove includere il JS nella pagina?

```
<! DOCTYPE html>
<html>
   <head>
      <title>Il mio primo JS</title>
      <script>
         // JS
      <script>
   </head>
   <body>
      <script>
         // JS
      <script>
   </body>
</html>
```

Javascript può essere incluso sia nel tag **head**, oppure nel tag **body**.

Non fa (quasi) differenza.

Non fa davvero differenza?

In realtà una pagina web **fatta bene** deve includere il JS come ultima cosa del tag **body**

Il browser interpreta la pagina dall'inizio alla fine. Se il JS richiede operazione pesanti la pagina verrà non verrà renderizzata finchè non è stato interpretato (e scaricato) tutto il JS.

E' più importante che la pagina si veda bene piuttosto che sia "dinamica". I tempi di interpretazione "massimi" del JS dovrebbero comunque attestarsi attorno ad 1s. **Massimi non medi!**

Non fa davvero differenza?

Se il JS è scritto in file esterno (e magari pesa parecchio) se è incluso nel tag head il browser impegnerà risorse per scaricare il JS, privandole ad altre più importanti nell'immediato (es. immagini/CSS)

Se il JS è embedded nella pagina e inserito nel tag head il browser inizierà la sua interpretazione subito appena incontra il tag script, privando risorse al rendering della pagina.

Il JS va sempre inserito nel body

Se il js è esterno oppure embedded ed è posizionato come ultima cosa del body quando il browser lo incontra avrà terminato il rendering della pagina oppure sarà in procinto di farlo e quindi non vengono "sprecate" risorse.

Questa regola generale non si applica ad alcuni script particolari (ad esempio il tracciamento dell'utente o il blocco dei cookies) che richiedono di essere inizializzati prima del contenuto (e del resto del codice JS)

2

Sintassi di base

Esempio JS

```
<script>
    alert('Hello World!');
</script>
```

alert è un comando che permette di visualizzare una finestra popup nel browser per mostrare delle informazioni.

(Non ha niente a che fare con gli alert di bootstrap)

Primo esempio

```
<script>
    alert('Hello World!');
</script>
```

alert è un comando che permette di visualizzare una finestra popup nel browser per mostrare delle informazioni.

(Non ha niente a che fare con gli alert di bootstrap)

Scrivere nella pagina

```
<script>
    document.write('<p>Hello World!</p>');
</script>
```

Hello World!

document.write('..'); ci consente di scrivere del contenuto nella pagina.

Nella quotidianità non è molto usata perché "non è abbastanza potente" come altre soluzioni (jQuery)

Scrivere nella pagina

```
<script>
    document.write('<p>Hello World!</p>');
</script>
```

Hello World!

Se visualizziamo il sorgente della pagina non troviamo il contenuto che abbiamo scritto.

Il contenuto è scritto **dinamicamente** dal browser all'interpretazione del codice JS.

Scrivere nella pagina

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Javascript test</title>
</head>
<body>
    <script>
        document.write('<p>Hello World!</p>');
    </script>
</body>
</html>
```

Commenti

In JS esistono due tipi di commento:

- Commenti di una sola riga
- Commenti multiriga

```
<script>
    // Singola riga
</script>
```

```
<script>
    /* Questo commmento
       e' composto da
       più righe */
</script>
```

Commenti a riga singola

```
<script>
    // Singola riga
</script>
```

- I commenti a riga singola si formano preponendo `//` prima del commento.
- Tutto ciò che si trova a destra di `//` fino alla fine della riga è considerato un commento.
- Si usano di solito per spiegazioni brevi o per annotazioni (brevi) sul funzionamento del codice.

Commenti multi riga

```
<script>
    /* Questo commmento
       e' composto da
       più righe */
</script>
```

- I commenti multi riga si aprono scrivendo /* e si chiudono con */
- Tutto ciò che è compreso tra /* e */ è considerato commento e può occupare più linee.
- Il commento a riga singola termina automaticamente quando finisce la riga
- Si usano di solito per spiegazioni molto lunghe

Commentare il codice

Quando si programma esistono delle situazioni in cui è necessario "disabilitare" parte del codice per capire l'origine un bug.

Un modo molto semplice e usatissimo è commentare il codice problematico.

```
<script>
    /*
        console.log('Questo non viene eseguito');
    */
</script>
```

Documentare il codice

```
// Dear programmer:
// When I wrote this code, only god and
// I knew how it worked.
// Now, only god knows it!
//
// Therefore, if you are trying to optimize
// this routine and it fails (most surely),
// please increase this counter as a
// warning for the next person:
//
// total_hours_wasted_here = 254
//
```

I commenti hanno la funzione di spiegare il funzionamento del codice.

Meglio un commento in più che un commento in meno!

Punto e virgola

Javascript non obbliga la separazione delle istruzioni attraverso il punto e virgola ";".

Non c'è una regola, **l'importante è essere consistenti!** Se si decide di metterlo allora è un errore non metterlo, viceversa se si decide di non metterlo è un errore metterlo!

```
<script>
   console.log('Senza ;')
</script>
```

```
<script>
   console.log('Con ;');
</script>
```

Variabili

Una variabile deve essere pensata come un contenitore di qualcosa (un numero, una stringa, etc).

Sono usate per contenere un valore "**temporaneo**" che può variare durante l'esecuzione del codice (da cui il nome variabile)

Le variabili sono identificate da un nome che permette di richiamarle nel codice.

Dare un nome alle variabili

Un nome valido di una variabile:

- **Non deve** coincidere con una parola chiave del linguaggio
- **Non può** iniziare con un numero
- **Non può** contenere caratteri speciali (spazi, lettere accentate, il trattino, etc)
- **Può** contenere un underscore
- **Può contenere** o iniziare con il simbolo del dollaro ($)

Le costanti in JS

Javascript (fino alle versione ES5) non conosce il concetto di costante, ovvero di un valore definito una sola volta e che non può (e non deve) variare durante l'esecuzione del codice.

Ad esempio le costante matematiche non devono variare (pi greco)

Convenzionalmente si è deciso che le costanti sono semplicemente variabili scritte tutte in maiuscolo con le parole eventualmente separate da _ (undescore). Es. PIGRECO, LA_MIA_COSTANTE

Le costanti in JS

```
var PIGRECO = 3.14; // Questa è una costante per convenzione
```

Dalla versione ES6 è stato introdotta la possibilità di dichiarate costanti usando la parola chiave **const**. Naturalmente la convenzione di usare nomi maiuscoli, eventualmente seprarati da _ (undescore) deve essere comunque rispettata: aumenta la leggibilità del codice.

```
const PIGRECO = 3.14; // Questa è una costante per ES6
```

Tutti i nomi sono "buoni nomi"?

Il codice che scrivete deve essere **autoesplicativo**

Il codice si dice essere autoesplicativo se è di facile comprensione e i nomi di variabili, funzioni etc rispecchiano esattamente quello che contengono/fanno.

Questo è di **fondamentale importanza** perché il codice deve essere letto da altri programmatori e più facile capire cosa fa e più è facile apportare modifice/migliorire/correggere problemi.

Tutti i nomi sono "buoni nomi"?

```
var a = 31; // Cosa significa??

var giorni = 31; // Già meglio, stiamo parlando di giorni

var numeroDiGiorniInDicembre = 31; // Ah, adesso si capisce!
```

I tipi di dato: stringe

```
<script>
    var string1 = 'abcd';
    var string2 = "abcd";
    var empty = '';
</script>
```

Le stringe sono sequenze di caratteri.

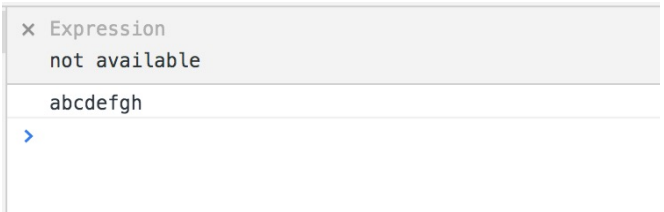
Le stringe sono delimitate da `""` oppure da `''`. Sono validi entrambi e non fanno differenza.

Scegliete pure quello che vi piace ma siate **consistenti: fate le cose sempre nello stesso modo!**

I tipi di dato: stringhe

Le stringhe possono essere concatenate: ovvero "sommare" due stringhe per formarne una composta dalla giustapposizione della prima e della seconda.

```
<script>
    var string1 = 'abcd';
    var string2 = "efgh";
    var string3 = string1 + string2;
    console.log(string3);
</script>
```



I tipi di dato: numeri

```
<script>
    var numero1 = 6;
    var numero2 = 3.14;
</script>
```

Javascript non fa differenza tra numero intero (es 6) e i numeri decimali (es 3.14).

Si usa la notazione americana, quindi il punto al posto della virgola.

Naturalmente sono supportate tutte le principali operazioni sui numeri.

I tipi di dato: bolean

```
<script>
    var vero = true;
    var falso = false;
</script>
```

Javascript possiede valori booleani, ovvero valori che assumono solamente uno di due valori: TRUE o FALSE.

Sono utilissimo soprattutto nella logica condizionale: if, while, etc dove si decidere di eseguire del codice solamente se si verifica una determina condizione.

I tipi di dato: array

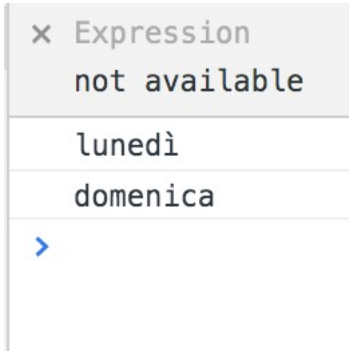
```
<script>
    var giorniDellaSettimana = [
        "lunedì",
        "martedì",
        "mercoledì",
        "giovedì",
        "venerdì",
        "sabato",
        "domenica"
];
</script>
```

Gli array sono un insieme numerato di variabili tutte dello stesso tipo.

E' possibile accedere ad uno specifico elemento attraverso il suo indice (gli indici partono da 0)

I tipi di dato: array

```
var giorniDellaSettimana = [
        "lunedì",
        "martedì",
        "mercoledì",
        "giovedì",
        "venerdì",
        "sabato",
        "domenica"
];
console.log(giorniDellaSettimana[0]);
console.log(giorniDellaSettimana[6]);
```



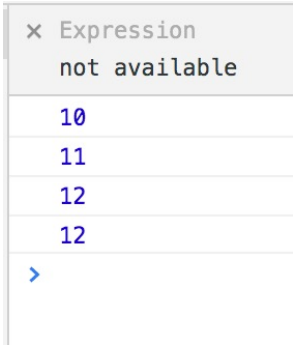
Gli operatori matematici

Operatore	Nome
`+`	addizione
`-`	sottrazione
`/`	divisione
`*`	moltiplicazione
`%`	modulo o resto

Gli operatori matematici unari

Operatore	Nome
`-`	negazione
`++`	incremento
`--`	decremento

```
var a = 10;
console.log(a++);
console.log(a);
console.log(++a);
console.log(a);
```



Gli operatori relazionali

Operatore	Nome
`<`	minore
`<=`	minore o uguale
`>`	maggiore
`>=`	maggiore o uguale
`==`	uguale
`!=`	diverso
`===`	strettamente uguale
`!==`	strettamente diverso

Gli operatori logici

Operatore	Nome
`&&`	and
`||`	or
`!`	not

Gli operatori di assegnamento

Forma compatta	Scrittura equivalente
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x %= y`	`x = x % y`

Istruzione alert

```
<script>
    alert('sono una prova');
</script>
```

L'istruzione **alert** ci permette di far apparire un box con in cui mostrare un messaggio di attenzione all'utente. Il box contiene un bottone OK per chiudere la finestra. Sono sconsigliate in un sito ed è meglio usare qualcosa di più carino tipo i modal di bootstrap.

Istruzione console.log

```
<script>
    console.log('test');
</script>
```

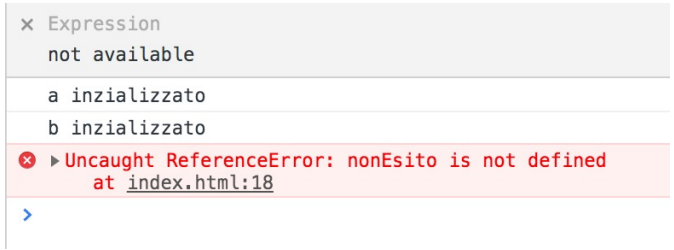
L'istruzione console.log ci permette di mostrare in console delle informazioni. E' perfetta nello sviluppo per stampare a console delle variabili con contenuto "ignoto", frutto di qualche operazione.

Quando il sito va in produzione la console deve essere "pulita", ovvero non devono rimanere nel codice istruzioni di console.log.

Istruzione console.log

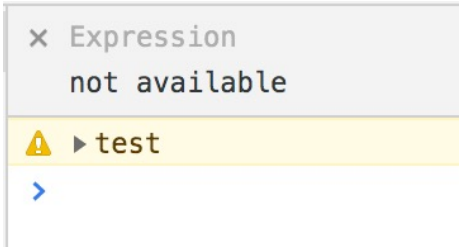
In caso di problemi con il codice può essere usata per capire a che punto il codice si blocca.

```
<script>
    var a = 10;
    console.log('a inzializzato');
    var b = 11;
    console.log('b inzializzato');
    var c = nonEsito(a, b);
    console.log('eseguita la funziona nonEsisto');
</script>
```



Istruzione console.warn

```
<script>
    console.warn('test');
</script>
```

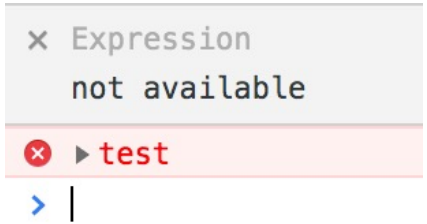


Identica come funzionamento a console.log, ma mostra un messaggio di attenzione.

Deve essere usata per indicare all'utente che qualcosa potrebbe funzionare a dovere ma non è detto. E' un messaggio di attenzione.

Istruzione console.error

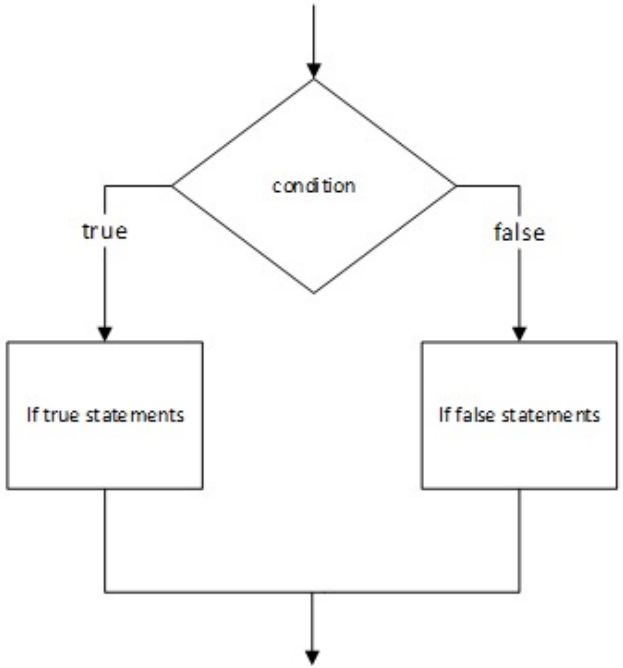
```
<script>
    console.error('test');
</script>
```



Funziona esattamente come console.log e console.warn ma mostra un messaggio di errore.

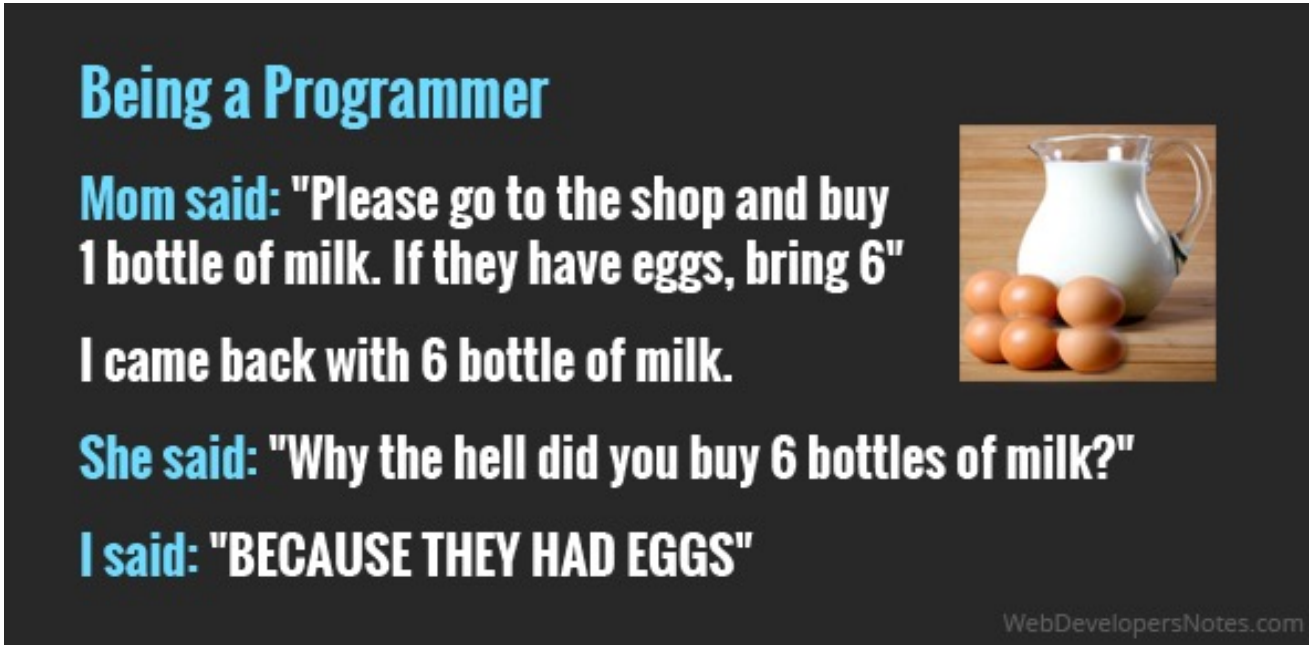
Può essere usata per avvertire l'utente che c'è stato qualche problema e che quindi il sito non funzionerà come voluto.

Controllo di flusso: if



Quando scriviamo il codice capita spesso di dover eseguire operazioni differenti se si verifica una determinata condizione.

Controllo di flusso: if



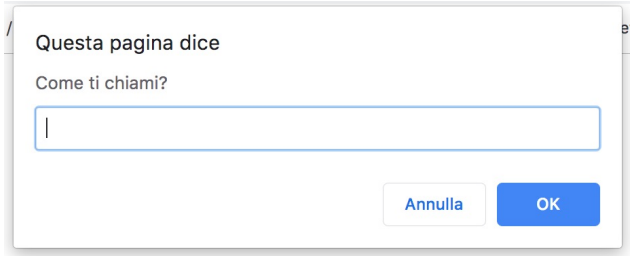
Controllo di flusso: if

```
if (codizione1) {
   // Codice se condizione è vera
} [else if (codizione2) {
   // Codice se condizione1 falsa e condizione2 vera
} [else {
  // Entrambe le condizioni sono false
}]]
```

Prompt

Prompt permette di richiedere all'utente l'immissione di alcune informazioni attraverso una finestra di dialogo.

```
<script>
    var name = prompt('Come ti chiami?');
    console.log(name);
</script>
```




```
× Expression
  not available
  Alessandro
>
```

3

Esercizio 2

Costruire una calcolatrice

Creare una calcolatrice dove all'utente è richiesto l'inserimento di un operando, dell'operatore e del secondo operando e mostrare nella pagina il risultato dentro un div di colore rosso se il risultato è negativo e verde se positivo.

Se l'operazione scelta è una divisione e il secondo operando è 0 mostrare un alert con scritto che il non è possibile fare divisioni per 0.

4

Esercizio 3

Controllo di flusso

Modificare l'esercizio precedente in modo che uno dei due operatori è nullo mostrare un alert indicante l'errore.

Iterazioni

Nella programmazione a volte è necessario ripetere del codice per un predeterminato numero di volte oppure finchè una condizione non diventa falsa.

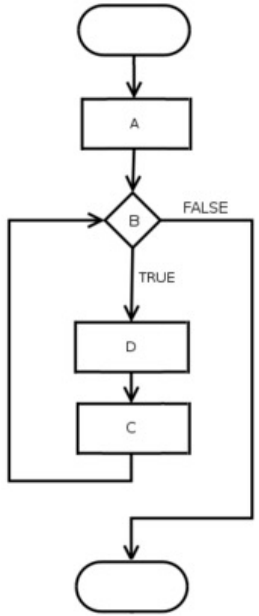
Javascript (così come gli altri linguaggi) possiede due costrutti: **for** e **while**

for: ripete il codice per un determinato numero di volte

while: ripete il codice finchè una condizione non diventa false

Iterazioni: for

```
for(A;B;C)
  D;
```



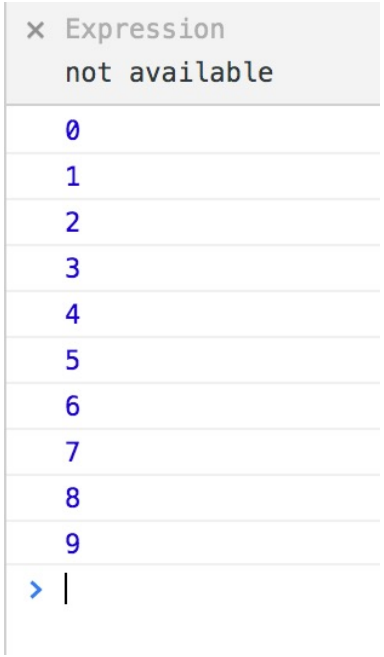
A è un istruzione di inizializzazione: inizializiamo una variabile contatore al suo valore minimo.

B è una condizione: finchè è valida esegui il codice D.

C è un'istruzione di incremento: incremento la variabile contatore

Iterazioni: for

```
<script>
    for (var i = 0; i < 10; i++) {
        console.log(i);
    }
</script>
```



Iterazioni: for

```
for (init; condizione; incremento) {
    // Codice da ripetere
}
```

Da notare che init, condizione e incremento sono separati da ;

Questo perché tutte e 3 sono istruzioni

Iterazioni: while

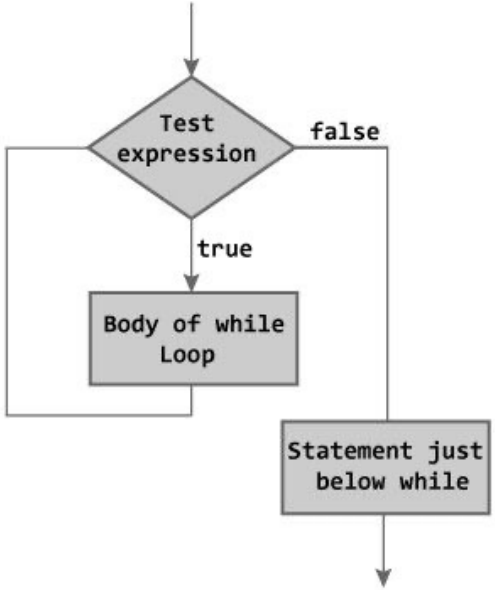


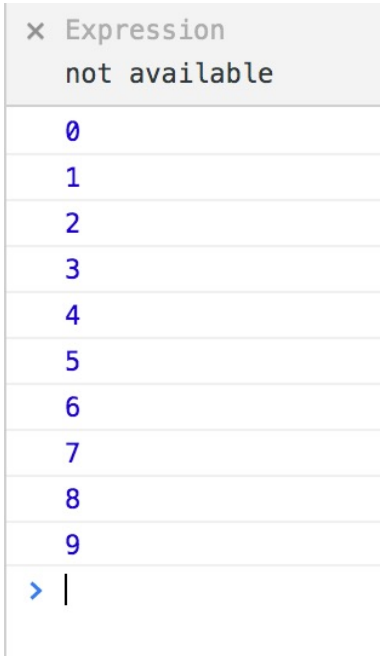
Figure: Flowchart of while Loop

Test expression è la nostra condizione.

"Funziona similarmente ad un if", ovvero finchè la condizione è vera, allora il codice viene ripetuto.

Iterazioni: while

```
<script>
    var i = 0;
    while (i < 10) {
        console.log(i);
        i++;
    }
</script>
```



Iterazioni: while

```
while (condizione) {
    // Codice da ripetere
}
```

for vs while

I costrutti per fare i cicli sono completamente equivalenti.

Il fatto che ne esistano diversi è solamente per aiutare il programmatore a scrivere codice più leggibile.

```
<script>
   for (var i = 0; i < 10; i++) {
      console.log(i);
   }
</script>
```

```
<script>
   var i = 0;
   while (i < 10) {
      console.log(i);
      i++;
   }
</script>
```

onclick

E' possibile eseguire del codice JS in risposta ad un evento.

Il più semplice è l'evento onclick, dove è possibile eseguire una funzione in risposta al click dell'evento.

```
<button onclick="eseguiClick()">Test</button>
<script>
    function eseguiClick() {
        alert('Hai premuto il tasto');
    }
</script>
```

Modificare il contenuto

JS permette di indentificare un elemento attraverso il suo id e modificare il suo contenuto.

```
<button id="test1" onclick="eseguiClick()">Premi</button>
<script>
    function eseguiClick() {
        var el = document.getElementById('test1');
        el.innerHTML = "Sono stato premuto";
    }
</script>
```

Aggiungere/Rimuovere classi

JS permette di aggiungere una o più classi di un elemento.

```
<button id="test1" onclick="eseguiClick()">Premi</button>
<script>
    var el = document.getElementById('test1');
    el.classList.add("my-class");
</script>
```

```
<button id="test1" onclick="eseguiClick()" class="my-class">Premi</button>
```

Aggiungere/Rimuovere classi

JS permette di rimuovere una o più classi di un elemento.

```
<button id="test1" onclick="eseguiClick()" class="my-class">Premi</button>
<script>
    var el = document.getElementById('test1');
    el.classList.remove("my-class");
</script>
```

```
<button id="test1" onclick="eseguiClick()" class>Premi</button> == $0
```

Ottenere il valore di un input

JS permette di ottenere il contenuto di un elemento di input

```
<input id="test1" value="test123">
<script>
    var el = document.getElementById('test1');
    var value = el.value;
    console.log(value);
</script>
```

Modificare il valore di un input

JS permette di ottenere il contenuto di un elemento di input

```
<input id="test1" value="">
<script>
    var el = document.getElementById('test1');
    el.value = '1234';
</script>
```

1234

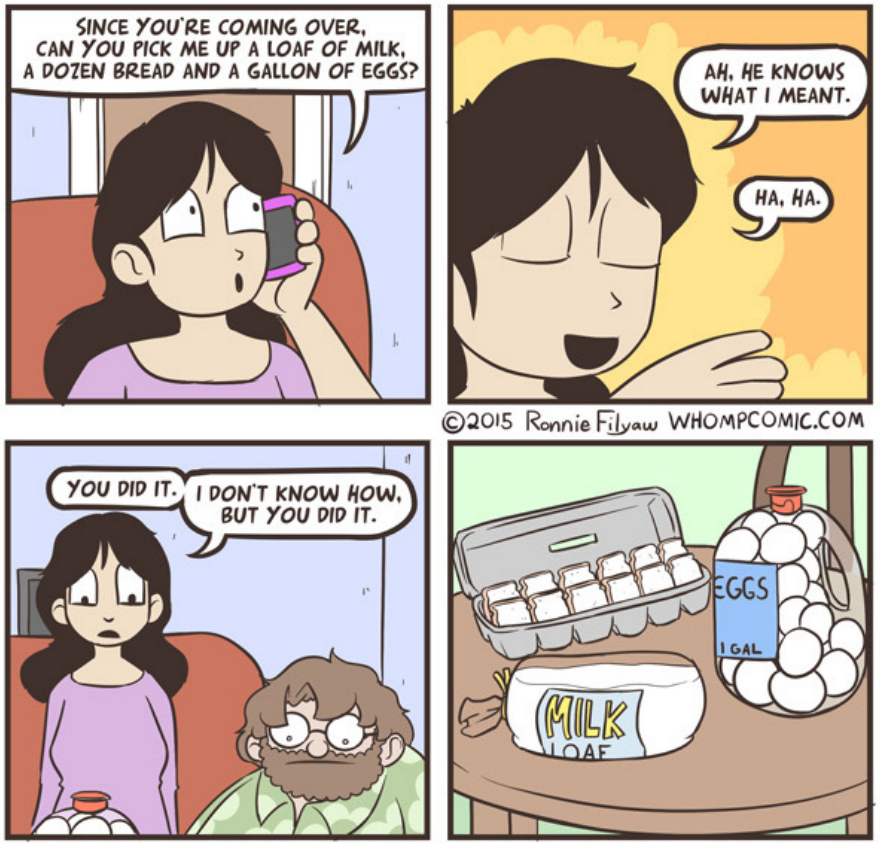
Esercizio 4

Calcolatrice 2

Creare una calcolatrice con 4 pulsanti, uno per operazione e due input per inserire gli operandi.

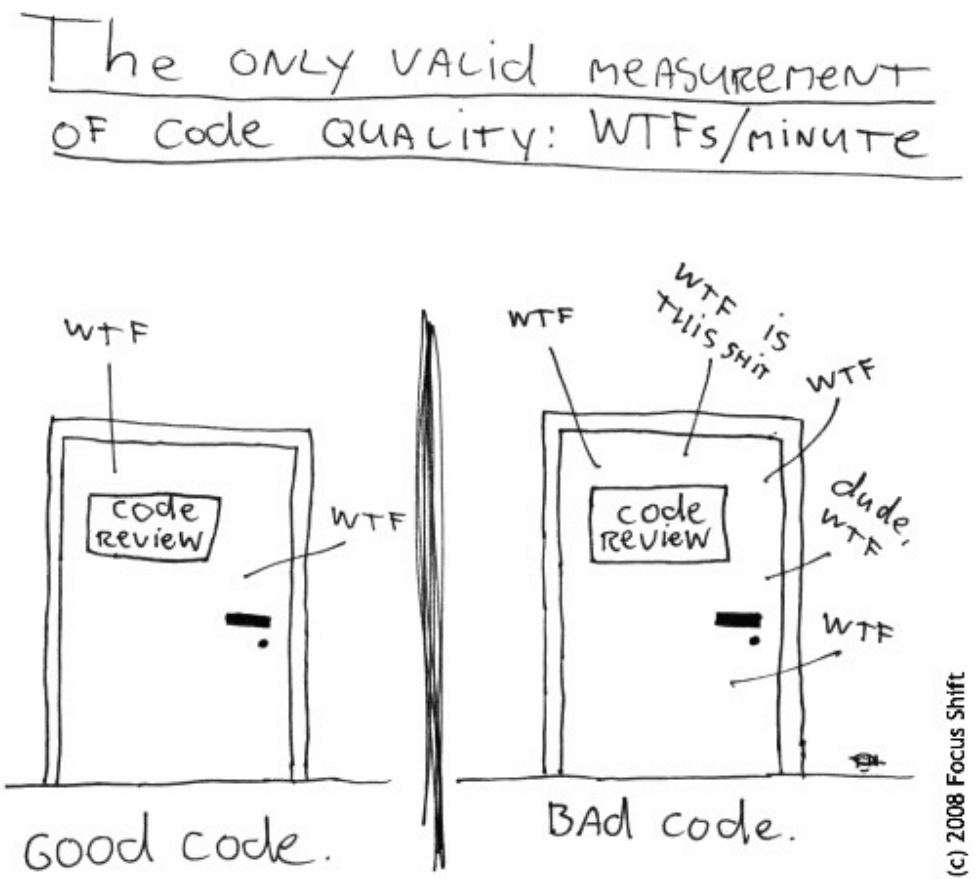
Creare un div per scrivere il risultato dell'operazione.

Se il risultato è positivo mettere il testo in verde, se è negativo in rosso.


SINCE YOU'RE COMING OVER, CAN YOU PICK ME UP A LOAF OF MILK, A DOZEN BREAD AND A GALLON OF EGGS?
AH, HE KNOWS WHAT I MEANT.
HA, HA.
©2015 Ronnie Filyaw WHOMPCOMIC.COM
YOU DID IT.
I DON'T KNOW HOW, BUT YOU DID IT.
EGGS
1 GAL
MILK

6

Indentazione e coding style


The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE
WTF
code review
WTF
Good code.
WTF
WTF IS THIS SHIT
WTF
code review
dude, WTF
WTF
Bad code.
(c) 2008 Focus Shift

You know you are working on clean code when each routine you read turns out to be pretty much what you expected. You can call it beautiful code when the code also makes it look like the language was made for the problem.

- Ward Cunningham

Perchè "clean code"?

La principale attività del programmatore è **leggere e comprendere codice**.

E' di fondamentale importanza scrivere codice comprensibile e facile da leggere. Più è facile da leggere meno tempo "perdiamo" nel comprendere il codice e più tempo possiamo dedicare ad attività "più produttive".

Coding style

```
var x=10;if(x>10)for(var i=0;i<10;i++)console.log(i);else
console.log('boh');
```

Coding style

```
var x=10;
if(x>10) {
for(var i=0;i<10;i++) {
console.log(i);
}
}else{
console.log('boh');
}
```

Coding style

```
var x = 10;

if (x > 10) {
    for (var i = 0; i < 10; i++) {
        console.log(i);
    }
} else {
    console.log('boh');
}
```

Documentare il codice

I commenti svolgono la funzione di documentazione del codice.

Il codice scritto bene ha bisogno di commenti che descrivono il comportamento di parti complicate del codice.

I commenti inutili o che dicono cose "ovvie" sono peggio di non avere commenti.

Nomi significativi

Il codice che scrivete deve essere autoesplicativo

Il codice si dice essere autoesplicativo se è di facile comprensione e i nomi di variabili, funzioni etc rispecchiano esattamente quello che contengono/fanno.

E' molto probabile che il vostro codice venga letto da altri programmatori (anche voi) dopo qualche tempo e l'unico modo di ricostruire il funzionamento è che il codice sia semplice, pulito e ordinato.

7

Istruzione switch

Istruzione switch

```
switch (variabile) {
   case '1':
      console.log('1');
      break;
   case '2':
      console.log('2');
      break;

   default:
      console.log('default');
}
```

Istruzione switch

```
switch (variabile) {
   case '1':
      console.log('1');
      break;
   case '2':
      console.log('2');
      break;

   default:
      console.log('default');
}
```

```
if (variabile == '1') {
   console.log('1');
} else if (variabile == '2') {
   console.log('2');
} else {
   console.log('default');
}
```

Istruzione switch

Switch è un costrutto che permette di sostituire una cascata di if e di migliorare la leggibilità del codice.

E' meno potente del costrutto if e ogni **case** deve obbligatoriamente essere "interrotto" da un istruzione **break;**

Istruzione switch (con break!)

```
var variabile = '1';

switch (variabile) {
   case '1':
      console.log('1');
      break;

   case '2':
      console.log('2');
      break;

   default:
      console.log('default');
}
```

```
× Expression
  not available

  1
> |
```

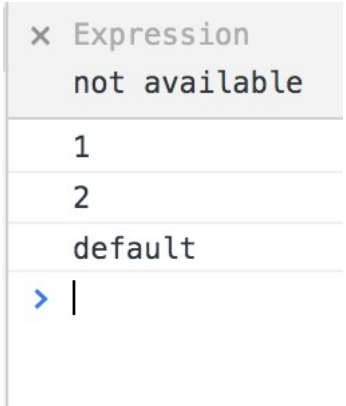
Istruzione switch (senza break!)

```
var variabile = '1';

switch (variabile) {
   case '1':
      console.log('1');

   case '2':
      console.log('2');

   default:
      console.log('default');
}
```



8

Esercizio

Esercizio

Creare un ciclo che scorra i primi 10 numeri naturali (1, 2, 3…) e dire se sono pari o dispari usando solamente switch.

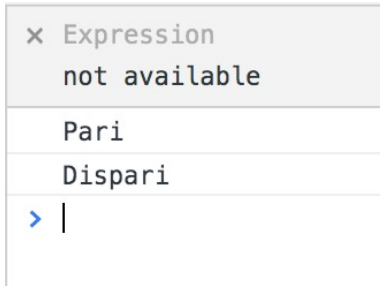
No if!

9

Le funzioni

Le funzioni

```
function isEven(number) {
    if (number % 2 == 0) {
        console.log('Pari');
    } else {
        console.log('Dispari');
    }
}
isEven(2);
isEven(3);
```



Le funzioni

Le funzioni sono un modo per raggruppare insieme del codice in modo da poterlo rieseguire senza doverlo riscrivere.

Le funzioni derivano dal concetto matematico di funzione dove dati determinati parametri di ingresso ci vengono ritornati uno o più parametri "calcolati" dalla funzione.

Sono utilizzate anche per raggruppare logicamente il codice.

Le funzioni

Le funzioni in Javascript posseggono (quasi sempre) un nome e dei parametri di ingresso e restituiscono un valore.

Se la funzione non ha un nome si chiama **funzione anonima**.

Le funzioni anonime sono molto usate per raggruppare il codice e le useremo con jQuery.

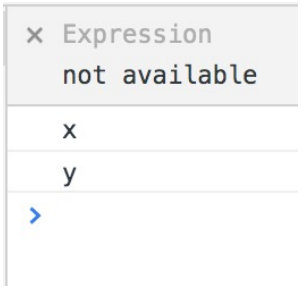
Nomi delle funzioni

I nomi delle funzioni, così come i nomi delle variabili, devono essere **univoci all'interno di un blocco di codice**.

Per Javascript un blocco è il codice definito all'interno di una coppia di parentesi graffe.

Nomi delle funzioni

```
{
    function a() {
        console.log('x');
    }
    a();
}
{
    function a() {
        console.log('y');
    }
    a();
}
```



Nomi delle funzioni

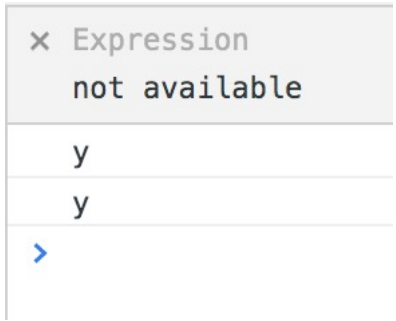
```
function a() {
    console.log('x');
}
a();

function a() {
    console.log('y');
}
a();
```

Nomi delle funzioni

```
function a() {
    console.log('x');
}
a();

function a() {
    console.log('y');
}
a();
```



Parametri di ingresso

I parametri di ingresso non sono altro che delle variabili che vengono passate alla funzione.

Sono usate per passare dei valori necessari all'esecuzione del codice della funzione.

Possono essere in qualsiasi numero e di qualsiasi tipo e numero.

Parametri di ingresso

```
function test(p1, p2, p3) {
    console.log(typeof p1);
    console.log(typeof p2);
    console.log(typeof p3);
}

test(1,2,3);
console.log('-----');
test(1);
```

```
× Expression
  not available
number
number
number
-----
number
undefined
undefined
>
```

Parametri di ingresso

Se un parametro non viene passato durante l'invocazione della funzione, Javascript gli assegna il tipo speciale **undefined**

Undefined significa che la variabile/parametro non è definito.

Prestare attenzione a questi "casi particolari" e passare sempre tutti i parametri alla funzione. E' bad practise passare meno parametri a meno che non siano definiti dei valori di default.

Parametri di ingresso

```
function test(parametro = false) {
    console.log(parametro);
}

test(true);
test();
```

```
× Expression
  not available
  true
  false
>
```

L'istruzione return

L'istruzione **return** è la funzione che permette alla funzione di ritornare un valore al termine della funzione.

Può ritornare un valore unico (gli array sono comunque un valore unico).

Può essere usata per interrompere l'esecuzione della funzione se usata da sola.

Le funzioni

```
function isOdd(number) {
    if (number % 2 != 0) {
        return true;
    }

    return false
}

console.log(isOdd(2));
console.log(isOdd(3));
```

L'istruzione return

Le funzioni ritornano sempre un valore, anche se non è specificata l'istruzione return oppure è vuota.

Questo valore è **undefined** ed è il valore di default di javascript.

Significa letteralmente **non definito**.

Variabili locali e variabili globali

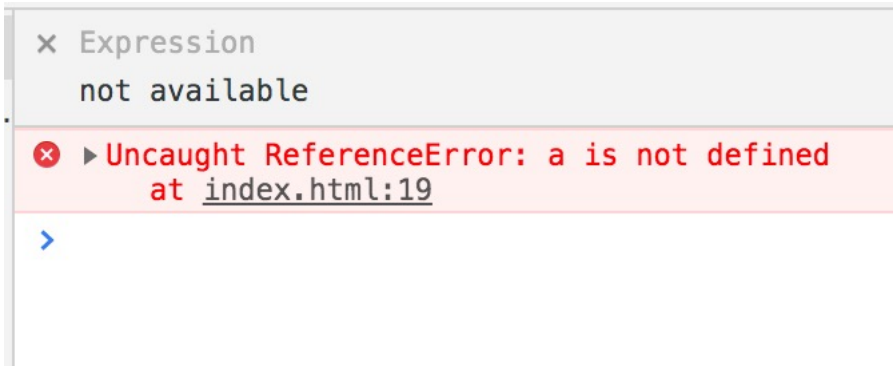
Le variabili definite all'esterno delle funzioni sono chiamate variabili globali.

Le variabili definite all'interno delle funzioni sono chiamate variabili locali.

La differenza sostanziale è che le variabili globali esistono globalmente, ovvero sono accessibili anche all'interno della funzione, mentre quelle locali hanno vita solamente durante l'esecuzione della funzione. Quando la funzione termina vengono distrutte e non sono più accessibili (perdono il contenuto)

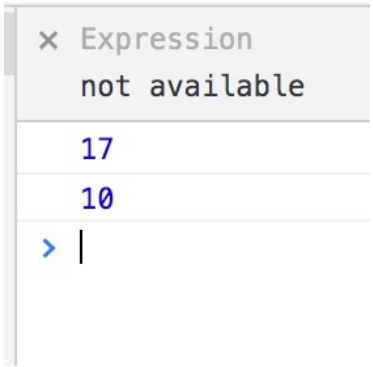
Variabili locali e variabili globali

```
function test(number) {
    var a = number;
}
var b = 10;
test(b);
console.log(a);
console.log(b);
```



Scoping delle variabili

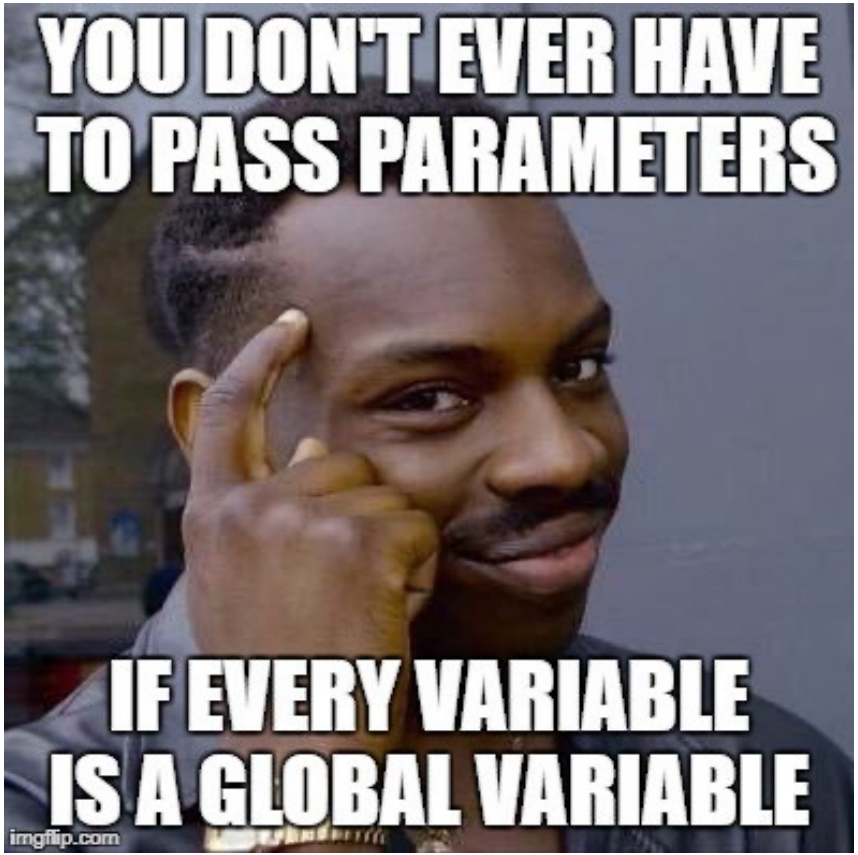
```
function test(number) {
    var a = number;
    console.log(a);
}
var a = 10;
test(17);
console.log(a);
```



Scoping delle variabili

```
function test(number) {
    a = number;
    console.log(a);
}
var a = 10;
test(17);
console.log(a);
```

```
✕ Expression
  not available
  17
  17
>
```


YOU DON'T EVER HAVE TO PASS PARAMETERS
IF EVERY VARIABLE IS A GLOBAL VARIABLE
imgflip.com

10

Introduzione oggetti JS

Oggetti

Un oggetto è una collezione di dati e/o funzionalità correlati.

Nella programmazione ad oggetti si "uniscono" dati e funzioni (chiamate metodi).

In javascript sono molto utilizzati anche come "configurazione" di alcune librerie. Es. slick

```
$('.one-time').slick({
  dots: true,
  infinite: true,
  speed: 300,
  slidesToShow: 1,
  adaptiveHeight: true
});
```

Proprietà degli oggetti

Gli oggetti sono variabili e quindi si istanziano (creano):

```
var persona = {};
```

Javascript assegna il tipo *object* alla variabile che contiene un oggetto.

Gli oggetti possono essere visti come un insieme di associazioni *chiave-valore* chiamate *proprietà.* Le proprietà sono delle variabili e quindi possono contenere tutto quello che una variabile può contenere.

Proprietà degli oggetti

```
var persona = {
    nome: 'John Smith',
    anni: 33
};

console.log(persona);
```

```
▼ {nome: "John Smith", anni: 33}
    nome: "John Smith"
    anni: 33
  ▶ __proto__: Object
```

Naturalmente è possibile accedere ad ogni proprietà singolarmente usando la notazione `oggetto.proprieta`

```
var persona = {
    nome: 'John Smith',
    anni: 33
};

console.log(persona.nome);
```

```
John Smith
```

Metodi degli oggetti

Gli oggetti sono collezioni di dati e funzionalità. I dati sono rappresentati dalle proprietà, mentre le funzionalità dai metodi.

I metodi sono delle funzioni definite all'interno dell'oggetto e possono usare *le proprietà del loro oggetto*.

```
var persona = {
   nome: 'John Smith',
   saluta: function () {
      console.log('Ciao');
   }
};
```

Metodi degli oggetti

Per usare un metodo è necessario invocarlo (chiamarlo).

```
var persona = {
   nome: 'John Smith',
   saluta: function () {
      console.log('Ciao');
   }
};

persona.saluta();
```

```
Ciao
```

Metodi degli oggetti

I metodi possono accedere anche alle proprietà dell'oggetto in cui sono definite.

```
var persona = {
   nome: 'John Smith',
   saluta: function () {
      console.log(this.nome);
   }
};

persona.saluta();
```

```
John Smith
```

Metodi degli oggetti

La parola chiave this può essere usata all'interno dei metodi per indicare l'oggetto in cui sono definti.

Viene usata se il metodo deve accedere ad una proprietà dell'oggetto.

```
var persona = {
   nome: 'John Smith',
   saluta: function () {
      console.log(this.nome);
   }
};

persona.saluta();
```

```
John Smith
```

11

JSON (JavaScript Object Notation)

JSON

JSON è un formato per il trasporto dei dati e deriva dalla notazione per gli oggetti di Javascript.

JSON è indipendente dal linguaggio usato. Si può usare anche con altri linguaggi di programmazione, ad esempio PHP, Java, etc

JSON sta sostituendo XML (un formato "simile" ad HTML) per il trasporto dei dati perchè è molto leggero (poca formattazione) ed è molto facile da leggere anche per gli umani.

XML VS JSON

```json
{
  "id": 123,
  "title": "Object Thinking",
  "author": "David West",
  "published": {
    "by": "Microsoft Press",
    "year": 2004
  }
}
```

JSON

```xml
<?xml version="1.0"?>
<book id="123">
  <title>Object Thinking</title>
  <author>David West</author>
  <published>
    <by>Microsoft Press</by>
    <year>2004</year>
  </published>
</book>
```

XML

La sintassi

JSON deriva dalla sintassi degli oggetti di Javascript e quindi è molto simile alla loro dichiarazione.

```
{
   "prop1": valore1,
   "prop2": valore2,
   …
}
```

JSON supporta stringhe, numeri, array, oggetti, boolean, null (è un tipo speciale)

12

Callback

Callback

```
var laMiaFunzione = function () {
   console.log('La mia funzione');
}

laMiaFunzione(); // Stampa 'La mia funzione'
```

Quindi è possibile dichiarare una funzione e salvarla all'interno di una variabile. Successivamente è possibile invocarla (eseguirla) semplicemente utilizzando il nome della variabile al posto del nome della funzione.

Callback

```
var laMiaFunzione = function () {
   console.log('La mia funzione');
}

laMiaFunzione();
```

```
function laMiaFunzione() {
   console.log('La mia funzione');
}

laMiaFunzione();
```

Entrambe le funzioni si comportano in maniera completamente identica e possono essere invocate allo stesso modo.

L'unica differenza tra le due dichiarazioni è che **la seconda esplicita il nome della funzione**, mentre **la prima è implicitamente il nome della variabile**.

Callback

Se Javascript permette di salvare le funzioni all'interno di una variabile cosa ci impedisce di passarle come argomenti di una funzione?

Un **callback** è una funzione passata come argomento di un'altra funzione.

Callback

```
// Dichiaro una funzione che accetta un callback
function funzioneDiTest(callback) {
   callback();
}


// Dichiaro una funzione all'interno di una variabile
var laMiaFunzione = function () {
   console.log('La mia funzione');
}


// Chiamo la prima funzione
funzioneDiTest(laMiaFunzione);
```

Callback

```
// Dichiaro una funzione che accetta un callback
function funzioneDiTest(callback) {
   callback();
}


// Dichiaro una funzione all'interno di una variabile
var laMiaFunzione = function () {
   console.log('La mia funzione');
}


// Chiamo la prima funzione
funzioneDiTest(laMiaFunzione);
```

```
La mia funzione
>
```

Callback

A volte è necessario dichiarare una funzione "temporanea", ovvero una funzione che serve solo una volta e non ci interessa salvarla da qualche parte: non la useremo mai più!

Allora al posto di dichiarare una variabile in una funzione e successivamente passare questa variabile ad una funzione come callback possiamo saltare questo passaggio: passiamo direttamente la funzione.

I parametri di una funzione (funtion (paramtro1, parametro2, ..) {...}) sono variabili!

Callback

```
// Dichiaro una funzione che accetta un callback
function funzioneDiTest(callback) {
   callback();
}


// Chiamo la prima funzione passando direttamente la
// funzione come argomento
funzioneDiTest(function () {
   console.log('La mia funzione');
});
```

Callback

```
// Dichiaro una funzione che accetta un callback
function funzioneDiTest(callback) {
   callback();
}


// Chiamo la prima funzione passando direttamente la
// funzione come argomento
funzioneDiTest(function () {
   console.log('La mia funzione');
});
```

```
La mia funzione
>
```

A cosa servono i callback?

Possiamo passare una funzione a come argomento ad un'altra funzione ma perché farlo?

Il codice Javascript che scriviamo viene eseguito principalmente al verificarsi di qualche evento: l'utente clicca su un bottone, la richiesta ajax è finita, la pagina è pronta etc

Quindi non è possibile conoscere a priori il flusso che il programma Javascript seguirà ma saranno gli eventi (o l'utente) che determineranno l'esatta sequenza di esecuzione.

A cosa servono i callback?

Visto che non possiamo conoscere a priori la sequenza (e quando) il nostro codice verrà eseguito dobbiamo prevedere un meccanismo capace di garantire comunque la corretta esecuzione del codice.

Questo meccanismo è fornito dai callback!

Attraverso i callback possiamo prevedere del codice che verrà eseguito solo se un certo evento si presenta.

A cosa servono i callback?

```
$('#pulsante').click(function () {
   console.log('Sono un callback!');
});
```

Quando definiamo un event handler (il codice da eseguire quanto un evento si verifica) stiamo in realtà definendo un callback.

Quello che stiamo dicendo a Javascript di fare è: quando l'utente clicca sul pulsante allora esegui la funzione che ti passo come argomento, ovvero esegui il callback!

Callback

Una variabile è un "contenitore" usato per contenere qualche informazione:

In Javascript è possibile memorizzare molte informazioni differenti:

- Numeri interi (number)
- Numeri a virgola mobile (number)
- Un carattere (string)
- Una stringa, ovvero un insieme di caratteri (string)
- Un valore booleano, ovvero **vero o falso** (boolean)
- Un insieme di variabili (array)
- Un oggetto (object)
- **Una funzione (function)**

```
var variabile;
```

Callback

```
var laMiaFunzione = function () {
   console.log('La mia funzione');
}

laMiaFunzione(); // Stampa 'La mia funzione'
```

Quindi è possibile dichiarare una funzione e salvarla all'interno di una variabile. Successivamente è possibile invocarla (eseguirla) semplicemente utilizzando il nome della variabile al posto del nome della funzione.

Callback

```
var laMiaFunzione = function () {
   console.log('La mia funzione');
}

laMiaFunzione();
```

```
function laMiaFunzione() {
   console.log('La mia funzione');
}

laMiaFunzione();
```

Entrambe le funzioni si comportano in maniera completamente identica e possono essere invocate allo stesso modo.

L'unica differenza tra le due dichiarazioni è che **la seconda esplicita il nome della funzione**, mentre **la prima è implicitamente il nome della variabile**.

Callback

Se Javascript permette di salvare le funzioni all'interno di una variabile cosa ci impedisce di passarle come argomenti di una funzione?

Un callback è una funzione passata come argomento di un'altra funzione.

Callback

```
// Dichiaro una funzione che accetta un callback
function funzioneDiTest(callback) {
   callback();
}


// Dichiaro una funzione all'interno di una variabile
var laMiaFunzione = function () {
   console.log('La mia funzione');
}


// Chiamo la prima funzione
funzioneDiTest(laMiaFunzione);
```

Callback

```
// Dichiaro una funzione che accetta un callback
function funzioneDiTest(callback) {
   callback();
}


// Dichiaro una funzione all'interno di una variabile
var laMiaFunzione = function () {
   console.log('La mia funzione');
}


// Chiamo la prima funzione
funzioneDiTest(laMiaFunzione);
```

```
La mia funzione
>
```

Callback

A volte è necessario dichiarare una funzione "temporanea", ovvero una funzione che serve solo una volta e non ci interessa salvare da qualche parte: non la useremo mai più!

Allora al posto di dichiarare una variabile in una funzione e successivamente passare questa variabile ad una funzione come callback possiamo saltare questo passaggio: passiamo direttamente la funzione.

I parametri di una funzione (funtion (paramtro1, parametro2, ..) {...}) sono variabili!

Callback

```
// Dichiaro una funzione che accetta un callback
function funzioneDiTest(callback) {
   callback();
}


// Chiamo la prima funzione passando direttamente la
// funzione come argomento
funzioneDiTest(function () {
   console.log('La mia funzione');
});
```

Callback

```
// Dichiaro una funzione che accetta un callback
function funzioneDiTest(callback) {
   callback();
}


// Chiamo la prima funzione passando direttamente la
// funzione come argomento
funzioneDiTest(function () {
   console.log('La mia funzione');
});
```

```
La mia funzione
>
```

Parametro vs argomento

Un **parametro** è un valore che una funzione prevede che gli venga passato. Quindi è incluso nella sua dichiarazione:

```
function laMiaFunzione(parametro1, parametro2) {

}
```

Un **argomento** è l'effettivo valore passato alla funzione, ovvero il valore che inseriamo nella sua chiamata:

```
laMiaFunzione(argomento1, argomento2);
```

A cosa servono i callback?

Possiamo passare una funzione a come argomento ad un'altra funzione ma perché farlo?

Il codice Javascript che scriviamo viene eseguito principalmente al verificarsi di qualche evento: l'utente clicca su un bottone, la richiesta ajax è finita, la pagina è pronta etc

Quindi non è possibile conoscere a priori il flusso che il programma Javascript seguirà ma saranno gli eventi (o l'utente) che determineranno l'esatta sequenza di esecuzione.

A cosa servono i callback?

Visto che non possiamo conoscere a priori la sequenza (e quando) il nostro codice verrà eseguito dobbiamo prevedere un meccanismo capace di garantire comunque la corretta esecuzione del codice.

Questo meccanismo è fornito dai callback!

Attraverso i callback possiamo prevedere del codice che verrà eseguito solo se un certo evento si presenta.

A cosa servono i callback?

```
$('#pulsante').click(function () {
   console.log('Sono un callback!');
});
```

Quando definiamo un event handler (il codice da eseguire quanto un evento si verifica) stiamo in realtà definendo un callback.

Quello che stiamo dicendo a Javascript di fare è: quando l'utente clicca sul pulsante allora esegui la funzione che ti passo come argomento, ovvero esegui il callback!

1

Approfondimento JS

L'operatore ternario

L'operatore ternario (chiamato anche istruzione condizionale) è una "scorciatoia" dell'if in alcuni casi:

```
var variabile;
if (condizione) {
   variabile = 'Condizione vera';
} else {
   variabile = 'Condizione falsa';
}
```

```
var variabile = condizione ? 'Condizione vera' : 'condizione falsa';
```

Thanks!

Any questions?

Per qualsiasi domanda contattatemi:
alessandro.pasqualini.1105@gmail.com