JAVASCRIPT

ITS Tullio Buzzi
Anno scolastico 2020/2021
Prof. Alessandro La Rosa

Un po’ di storia

JavaScript è stato ideato nel 1995 da Netscape e rilasciato inizialmente con il nome **LiveScript.**

Nasce per rendere dinamica la pagina HTML, in base all’interazione dell’utente con il browser (lato client). Questo grazie alle funzionalità di calcolo e di manipolazione dei documenti che era possibile effettuare anche senza coinvolgere il server.

Un po’ di storia

Dopo anni di lento declino, l’avvento della tecnologia Ajax, la possibilità cioè di comunicare in maniera asincrona con il server tramite script, riportò JavaScript ad un livello predominante.

L’evoluzione del linguaggio ha fatto nascere il cosiddetto **Web 2.0** ed ha fatto fiorire numerose librerie con lo scopo di semplificare alcune delle attività più comuni e di superare le differenze che ancora c’erano tra i Browser, favorendo una programmazione unificata e più rapida.

Un po’ di storia

L’avvento di **HTML5** ha ulteriormente amplificato le possibilità applicative del linguaggio, anche al di fuori del semplice Web browser. **JavaScript** può essere utilizzato anche lato server, in applicazioni desktop e mobile. Non si tratta più, come succedeva nelle versioni precedenti, di un semplice collante tra codice HTML e l’utente.

Caratteristiche del linguaggio

- Orientato agli oggetti (OOP)
- Usa prototipi
- Guidato dagli eventi (event driven).
- Interpretato (l'interprete è il browser).
- Blandamente tipizzato (non occorre dichiarare il tipo delle variabili)
- Case sensitive

Strumenti di lavoro

- Editor di testo;
- un interprete;
- un debugger.

Come editor è possibile utilizzare un comune editor di testo (Notepad++, Atom, Visual Studio Code, Webstorm, Sublime text…), ma anche alcuni IDE (Eclipse…).

Alcuni ambienti di sviluppo prevedono un interprete o un compilatore integrato (**JavaScript engine**), ma possiamo sempre sfruttare l'engine di un comune browser.

Da interpretato a compilato

Fino a qualche anno fa JavaScript era un linguaggio esclusivamente interpretato.

L'esigenza di performance sempre maggiori ha poi condotto alla creazione di engine che effettuassero una compilazione in tempo reale (**JIT, Just In Time compilation**) in bytecode o addirittura in codice macchina (**V8** di Google, **Chakra** di Microsoft, **SpiderMonkey** di Mozilla…).

Debug

Gli ambienti di sviluppo che prevedono un engine hanno in genere un debugger integrato che consente di analizzare il codice durante l'esecuzione.

In assenza di un ambiente di sviluppo integrato, possiamo comunque ricorrere a un comune browser, dal momento che i più diffusi prevedono ormai un ambiente di debugging (**Console**).

CODICE JAVASCRIPT IN UNA PAGINA HTML

Metodologie

- Inserire codice **inline**;
- scrivere **blocchi di codice nella pagina**;
- importare **file esterni** con codice JavaScript.

Codice inline

Consente di inserire direttamente le istruzioni **JavaScript** nel codice di un elemento **HTML.**

Assegna quindi un'istruzione ad un attributo che rappresenta un evento.

```
<button type="button" onclick="alert('Questo è un evento generato da un'istruzione JavaScript!')">Clicca qui</button>
```

Codice inline

Abbiamo assegnato all’attributo **onclick** dell’elemento **button** la stringa **alert('Questo è un evento generato da un’istruzione JavaScript!’)**.

L’attributo **onclick** rappresenta l’evento del clic con il pulsante del mouse, quindi in corrispondenza di questo evento verrà analizzato ed eseguito il codice JavaScript assegnato (la visualizzazione di un pop-up con il testo inserito tra parentesi tonde e apici).

Codice inline

Un altro approccio per l'inserimento di codice inline, utilizzabile però soltanto con i link, è quello mostrato nel seguente esempio:

```
<a href="javascript:alert('Questo è un
evento generato da un'istruzione
JavaScript!')">Clicca qui</a>
```

In questo caso indichiamo al browser di interpretare il link come la richiesta di esecuzione del codice JavaScript specificato invece che come un collegamento ad un'altra pagina.

Blocchi di codice nella pagina

In un documento HTML il codice JavaScript viene inserito tra i tag `<script>…</script>`.

È possibile inserire un numero qualsiasi di script, sia nella sezione <body>, che nella sezione <head>.

È consigliabile inserire il codice JavaScript alla fine del body per non rallentare il caricamento della pagina HTML da parte dei browser.

Blocchi di codice nella pagina

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      // codice JavaScript
    </script>
  </head>
  <body>
    <script>
      // codice JavaScript
    </script>
    <script>
      // codice JavaScript
    </script>
  </body>
</html>
```

File esterni

Gli script possono anche essere inseriti in file esterni con estensione **.js**

Essi sono pratici quando lo stesso codice viene utilizzato in diverse pagine web.

Per richiamare uno script esterno basta inserire il nome del file nell'attributo `src` all'interno del tag `<script>`.

```
<script src="fileEsterno.js"> </script>
```

File esterni

L'inserimento di script in file esterni presenta alcuni vantaggi:

- Separa HTML e codice
- Rende HTML e JavaScript più facili da leggere
- I file JavaScript memorizzati nella cache possono velocizzare i caricamenti delle pagine

Per aggiungere più file di script a una pagina, bisogna utilizzare diversi tag di script:

```
<script src="fileEsterno1.js"> </script>
<script src="fileEsterno2.js"> </script>
```

File esterni

È possibile fare riferimento a script esterni con un URL completo o con un percorso relativo alla pagina Web corrente.

```
<script src="https://code.jquery.com/jquery-3.5.1.js"> </script>
```

GESTIRE GLI OUTPUT

Visualizzazioni possibili

È possibile "visualizzare" gli output di JavaScript in diversi modi:

- scrivendo all'interno di un elemento HTML, tramite l'utilizzo di **innerHTML**;
- scrivendo nell'output HTML usando **document.write()**;
- scrivendo in una casella di avviso, utilizzando **window.alert()**;
- scrivendo nella console del browser, utilizzando **console.log()**.

innerHTML

Per accedere a un elemento HTML, JavaScript può utilizzare il metodo **document.getElementById(id)**.

L'attributo **id** definisce l'elemento HTML. La proprietà **innerHTML** definisce il contenuto HTML.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>La mia prima pagina web</h1>
    <p id = "testo"></p>
    <script>
      document.getElementById("testo").innerHTML = "Ciao!";
    </script>
  </body>
</html>
```

document.write()

Un altro metodo i visualizzazione dell'output JavaScript è il metodo **document.write()**.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>La mia prima pagina web</h1>
    <script>
      document.write("Ciao!");
    </script>
  </body>
</html>
```

document.write()

Attenzione! L'utilizzo di **document.write()** dopo il caricamento di un documento HTML **eliminerà tutta la pagina HTML esistente**.
Tale metodo dovrebbe essere utilizzato solo per i test.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>La mia prima pagina web</h1>
    <script>
      document.write("Ciao!");
    </script>
    <br />
    <button type = "button" onclick = "document.write(5 +
    6)">Cliccami</button>
  </body>
</html>
```

window.alert()

Con questo metodo è possibile ottenere una casella di avviso per visualizzare i dati.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>La mia prima pagina web</h1>
    <script>
      window.alert(5 + 6);
    </script>
  </body>
</html>
```

window.alert()

In JavaScript, l'oggetto **window** rappresenta anche **il contesto di esecuzione globale** per JavaScript, cioè l'oggetto all'interno del quale vengono definite variabili e funzioni globali. Ciò significa anche che la specifica della parola chiave **window** è facoltativa.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>La mia prima pagina web</h1>
    <script>
      alert(5 + 6);
    </script>
  </body>
</html>
```

console.log()

Il metodo **console.log()** viene utilizzato per effettuare il debug e per visualizzare i dati nel browser.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>La mia prima pagina web</h1>
    <p>Premi F12 per attivare il debug.</p>
    <p>Selezione "Console" nel menù del debugger.</p>
    <script>
      console.log(5 + 6);
    </script>
  </body>
</html>
```

window.print()

È possibile stampare una pagina web utilizzando un semplice codice in JavaScript.

Il metodo **print()** stampa il contenuto della finestra corrente aprendo la finestra di dialogo "Stampa" che consente di scegliere tra varie opzioni di stampa.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>La mia prima pagina web</h1>
    <button onclick = "window.print()">Stampa la
    pagina</button>
  </body>
</html>
```

SINTASSI

Codice JavaScript

Un programma **JavaScript** (codice JavaScript) è un elenco di **istruzioni** di programmazione.

Le istruzioni JavaScript sono composte da: valori, operatori, espressioni, parole chiave o commenti.

Esse vengono eseguite, una per una, nello stesso ordine in cui sono scritte.

Ogni istruzione è separata dalla successiva con il **;** (punto e virgola).

```
var a, b, c; // Dichiaro 3 variabili
a = 5;       // Assegno il valore 5 alla variabile a
b = 6;       // Assegno il valore 6 alla variabile b
c = a + b;   /* Assegno il risultato della somma di
                a e b alla variabile c */
```

Sono consentite più istruzioni su una stessa riga, se separate da **;**

```
a = 5; b = 6; c = a + b;
```

Codice JavaScript

JavaScript ignora più spazi. È possibile aggiungere uno spazio bianco allo script per renderlo più leggibile.

Le seguenti righe sono equivalenti:

```
var persona = "Giovanni";
var persona="Giovanni";
```

Una buona pratica è quella di inserire degli spazi attorno agli operatori (= + - * /):

```
var x = y + z;
```

Se un'istruzione JavaScript non si adatta a una riga, il posto migliore per interromperla è dopo un operatore:

```
document.getElementById("testo").innerHTML =
"Hello World!";
```

Codice JavaScript

Le istruzioni JavaScript possono essere raggruppate in blocchi di codice, all'interno di parentesi graffe {...}.

Lo scopo dei blocchi di codice è definire le istruzioni da eseguire insieme.

Un esempio lo troviamo nelle funzioni JavaScript:

```
<p id="testo1"></p>
<p id="testo2"></p>

<script>
  function myFunction() {
    document.getElementById("testo1").innerHTML =
    "Hello World!";
    document.getElementById("testo2").innerHTML =
    "How are you?";
  }
</script>
```

Valori fissi

In JavaScript è possibile definire due tipi di valori:

- Valori fissi (chiamati **letterali**)
- Valori variabili (chiamati **variabili**)

I valori fissi prevedono due regole di sintassi:

1. I **numeri** sono scritti con o senza decimali
2. Le **stringhe** sono testo, scritto tra virgolette doppie (") o singole (')

Variabili

Le **variabili** vengono utilizzate per **memorizzare** i valori dei dati.

JavaScript utilizza la parola chiave **var** (fino al 2015) per **dichiarare le** variabili.

Il **segno di uguale** viene utilizzato per **assegnare valori** alle variabili.

Nel seguente esempio, x è definita come una variabile. Quindi, a x viene assegnato il valore 6:

```
<script>
  var x;
  x = 6;
  document.getElementById("demo").innerHTML =
  x;
</script>
```

Operatori

Vengono utilizzati **operatori aritmetici** (+ - * /) per calcolare i valori: (5 + 6) * 10

JavaScript utilizza un **operatore di assegnazione** (=) per **assegnare** valori alle variabili:

```
var x, y;
x = 5;
y = 6;
```

Espressioni

Un'espressione è una combinazione di valori, variabili e operatori, che calcola un valore.

Le espressioni possono anche contenere valori di variabili:

x * 10

I valori possono essere di vari tipi, come numeri e stringhe.

Ad esempio, "Marco" + " " + "Polo", restituisce "Marco Polo":

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

<script>
  document.getElementById("demo").innerHTML =
  "Marco" + " " + "Polo";
</script>
```

Parole chiave

Le **parole chiave** JavaScript vengono utilizzate per identificare le azioni da eseguire.

La parola chiave **var** dice al browser di creare variabili.

È possibile consultare l'elenco di tutte le parole chiave attraverso il sito https://www.w3schools.com/js/js_reserved.asp

Commenti

Non tutte le istruzioni JavaScript vengono "eseguite".

Il codice dopo le doppie barre // o tra /* e */viene considerato un **commento** .

I commenti vengono ignorati e non verranno eseguiti:

```
var x = 5;   //Istruzione eseguita

// var x = 6;  Istruzione non eseguita
```

Commenti

I commenti JavaScript possono essere utilizzati per spiegare il codice e per renderlo più leggibile, ma anche per impedire l'esecuzione, durante la fase di test, di codice alternativo.

Possono essere scritti su una sola riga o su più righe.

- I commenti su una riga iniziano con // (doppio slash). Qualsiasi testo compreso tra // e la fine della riga non verrà eseguito.
- I commenti su più righe iniziano con /* e finiscono con */. Qualsiasi testo compreso tra /* e */ non verrà eseguito.

Commenti

Durante la fase di test è possibile utilizzare i commenti per impedire l'esecuzione del codice.

L'esempio seguente utilizza // per impedire l'esecuzione di una delle righe di codice:

```
//document.getElementById("myH").innerHTML = "My
First Page";
document.getElementById("myP").innerHTML = "My first
paragraph.";
```

Quest'altro esempio utilizza un blocco di commenti per impedire l'esecuzione di più righe:

```
/*
document.getElementById("myH").innerHTML =
"My First Page";
document.getElementById("myP").innerHTML =
"My first paragraph.";
*/
```

Identificatori

Gli identificatori sono nomi che vengono utilizzati per denominare variabili (e parole chiave, funzioni ed etichette).

Le regole per la scelta dei nomi sono più o meno le stesse nella maggior parte dei linguaggi di programmazione.

In JavaScript, il primo carattere deve essere una lettera (per convenzione minuscola), un trattino basso (_) o un segno di dollaro ($).

I caratteri successivi possono essere lettere, cifre, trattini bassi o segni di dollaro.

Bisogna utilizzare la notazione **camel case** (scrivere parole composte o frasi unendo tutte le parole tra loro, ma lasciando le loro iniziali maiuscole).
Essendo JavaScript case sensitive, gli identificatori fanno **distinzione tra maiuscole e minuscole**.

Le variabili lastName e lastname, sono due variabili differenti.

Set di caratteri

Come per l'HTML, JavaScript utilizza il set di caratteri **Unicode**.

Unicode copre (quasi) tutti i caratteri, i segni punteggiatura e i simboli del mondo.

VARIABILI

Variabili

Le variabili JavaScript sono contenitori per l'archiviazione dei valori dei dati.
In questo esempio, x, y, e z, sono variabili, dichiarate con la parola chiave **var**:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Variabili JavaScript</h2>
    <p>In questo esempio, x, y, e z sono variabili.</p>
    <p id="demo"></p>
    <script>
      var x = 5;
      var y = 6;
      var z = x + y;
      document.getElementById("demo").innerHTML =
      "Il valore di z è: " + z;
    </script>
  </body>
</html>
```

const e let

Prima del 2015, l'utilizzo della parola chiave **var** era l'unico modo per dichiarare una variabile JavaScript.

La versione 2015 di JavaScript (ES6) consente l'utilizzo della parola chiave **const** per definire una variabile che non può essere riassegnata e della parola chiave **let** per definire una variabile con ambito limitato.

var

Nella programmazione, proprio come in algebra, usiamo variabili (come prezzo1) per contenere valori e nelle espressioni (totale = prezzo1 + prezzo2).

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Variables</h2>
    <p id="demo"></p>
    <script>
      var prezzo1 = 5;
      var prezzo2 = 6;
      var totale = prezzo1 + prezzo2;
      document.getElementById("demo").innerHTML =
      "Il totale è: " + totale;
    </script>
  </body>
</html>
```

Identificatori

Tutte le **variabili** JavaScript devono essere **identificate** con **nomi univoci** .

Questi nomi univoci sono chiamati **identificatori** .

Gli identificatori possono essere nomi brevi (come x e y) o nomi più descrittivi (età, somma, volume totale).

Le regole generali per la costruzione di nomi per le variabili (identificatori univoci) sono:

- i nomi possono contenere lettere, cifre, trattini bassi e segni di dollaro
- i nomi devono iniziare con una lettera
- i nomi possono anche iniziare con $ e _
- i nomi fanno distinzione tra maiuscole e minuscole (y e Y sono variabili diverse)
- le parole riservate (come le parole chiave JavaScript) non possono essere utilizzate come nomi.

Assegnazione

In JavaScript, il segno di uguale (=) è un operatore di "**assegnazione**", non un operatore di "uguale a".

Questo è diverso dall'algebra.

x = x + 5 non ha senso in algebra. In JavaScript, tuttavia, ha perfettamente senso: assegna il valore di x + 5 alla variabile x (calcola il valore di x + 5 e mette il risultato in x, quindi Il valore di x viene incrementato di 5).

In JavaScript l'operatore "uguale a" è scritto come ==

Tipi di dato

Le variabili JavaScript possono contenere numeri come 100 e valori di testo come "Marco Polo".
Nella programmazione, i valori di testo sono chiamati stringhe di testo.
Le stringhe sono scritte tra virgolette doppie o singole. I numeri sono scritti senza virgolette.
Mettendo un numero tra virgolette, verrà trattato come una stringa di testo.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Variabili JavaScript</h2>
    <p>Le stringhe sono scritte tra virgolette.</p>
    <p>I numeri vengono scritti senza virgolette.</p>
    <p id="demo"></p>
    <script>
      var pi = 3.14;
      var persona = "Marco Polo";
      var città = 'Venezia';
      document.getElementById("demo").innerHTML =
      pi + "<br>" + persona + "<br>" + città;
    </script>
  </body>
</html>
```

Dichiarazione

La creazione di una variabile in JavaScript è chiamata "**dichiarazione**" di una variabile.

```
var carName;
```

Dopo la dichiarazione, la variabile non ha valore (tecnicamente ha come valore undefined).

Per **assegnare** un valore alla variabile, bisogna usare il segno di uguale:

```
carName = "Volvo";
```

È possibile anche assegnare un valore alla variabile quando viene dichiarata:

```
var carName = "Volvo";
```

Nell'esempio alla slide seguente, creiamo una variabile chiamata carName, le assegniamo il valore "Volvo" ed "inseriamo" il valore all'interno di un paragrafo HTML con id = "demo".

Esempio

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Variables</h2>
    <p>Create a variable, assign a value to it, and
display it:</p>
    <p id="demo"></p>
    <script>
      var carName;
      carName = "Volvo";
      document.getElementById("demo").innerHTML =
      carName;
    </script>
  </body>
</html>
```

Dichiarazioni

È possibile dichiarare molte variabili in un'unica istruzione.
Per fare questo bisogna iniziare l'istruzione con var e separare le variabili con una **virgola**.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Variabili JavaScript</h2>
    <p> Puoi dichiarare molte variabili in un'unica
    istruzione.</p>
    <p id="demo"></p>
    <script>
      var persona = "Marco Polo", carName = "Volvo",
      prezzo = 200;
      document.getElementById("demo").innerHTML = carName;
    </script>
  </body>
</html>
```

Dichiarazioni

Una dichiarazione può estendersi su più righe. L'esempio precedente può anche essere scritto nel modo seguente:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Variabili JavaScript</h2>
    <p> Puoi dichiarare molte variabili in un'unica
    istruzione.</p>
    <p id="demo"></p>
    <script>
      var persona = "Marco Polo",
      carName = "Volvo",
      prezzo = 200;
      document.getElementById("demo").innerHTML =
      carName;
    </script>
  </body>
</html>
```

undefined

Nei programmi per computer, le variabili vengono spesso dichiarate senza valore. Il valore può essere qualcosa che deve essere calcolato o qualcosa che verrà fornito in seguito, come l'input dell'utente.

Una variabile dichiarata senza un valore avrà il valore **undefined**.

La variabile carName avrà il valore undefined dopo l'esecuzione di questa istruzione:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Variabili JavaScript</h2>
    <p> Una variabile dichiarata senza un valore avrà il
    valore undefined.</p>
    <p id="demo"></p>
    <script>
      var carName;
      document.getElementById("demo").innerHTML = carName;
    </script>
  </body>
</html>
```

Ridichiarazione di una variabile

Dichiarando nuovamente una variabile JavaScript, non perderà il suo valore.

La variabile carName avrà ancora il valore "Volvo" dopo l'esecuzione di queste istruzioni:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Variabili JavaScript</h2>
    <p> Se dichiari nuovamente una variabile JavaScript, non
    perderà il suo valore.</p>
    <p id="demo"></p>
    <script>
      var carName = "Volvo";
      var carName;
      document.getElementById("demo").innerHTML = carName;
    </script>
  </body>
</html>
```

Operazioni aritmetiche

Come con l'algebra, è possibile eseguire operazioni aritmetiche con variabili JavaScript, utilizzando operatori come = e +

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Variabili JavaScript</h2>
    <p>Il risultato della somma di 5 + 2 + 3 è:</p>
    <p id="demo"></p>
    <script>
      var x = 5 + 2 + 3;
      document.getElementById("demo").innerHTML = x;
    </script>
  </body>
</html>
```

Somma di stringhe

È anche possibile sommare stringhe, ma le stringhe verranno concatenate:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Variabili JavaScript</h2>
    <p>Il risultato della somma di "Marco" + " " +
    "Polo" è:</p>
    <p id="demo"></p>
    <script>
      var x = "Marco" + " " + "Polo";
      document.getElementById("demo").innerHTML = x;
    </script>
  </body>
</html>
```

Casi particolari

Se inseriamo un numero tra virgolette, il resto dei numeri verrà trattato come stringhe e concatenato.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Variabili JavaScript</h2>
    <p>Il risultato della somma di "5" + 2 + 3
    è:</p>
    <p id="demo"></p>
    <script>
      x = "5" + 2 + 3;
      document.getElementById("demo").innerHTML = x;
    </script>
  </body>
</html>
```

Casi particolari

Le cifre prima della stringa verranno invece trattate come numeri.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Variabili JavaScript</h2>
    <p>Il risultato della somma di 2 + 3 + "5" + 6 +
    7 è:</p>
    <p id="demo"></p>
    <script>
      x = 2 + 3 + "5" + 6 + 7;
      document.getElementById("demo").innerHTML = x;
    </script>
  </body>
</html>
```

$

Ricordiamo che gli identificatori JavaScript devono iniziare con:

- una lettera (AZ o az)
- un segno di dollaro ($)
- un trattino basso (_)

Poiché JavaScript tratta un segno di dollaro come una lettera, gli identificatori che contengono $ sono nomi di variabili validi:

```
var $$$ = "Hello World";
var $ = 2;
var $myMoney = 5;
```

L'utilizzo del simbolo del dollaro non è molto comune in JavaScript, ma i programmatori lo usano spesso come alias per la funzione principale in una libreria JavaScript.

Nella libreria JavaScript jQuery, ad esempio, la funzione principale $ viene utilizzata per selezionare gli elementi HTML. In jQuery **$("p");** significa "seleziona tutti gli elementi p".

Underscore

Poiché JavaScript tratta l'underscore come una lettera, gli identificatori che contengono _ sono nomi di variabili validi:

```
var _lastName = "Polo";
var _x = 2;
var _100 = 5;
```

L'uso del trattino basso non è molto comune in JavaScript, ma una convenzione tra i programmatori è di usarlo come alias per variabili "private (nascoste)".

LET E CONST

Ambito globale ed ambito della funzione

Abbiamo visto che, con la versione ES5 di JavaScript, l'unico modo per identificare le variabili era tramite l'identificatore var.

Fino a quella versione JavaScript aveva solo due ambiti: **ambito globale** e **ambito della funzione**.

Ambito globale

Le variabili dichiarate **globalmente** (al di fuori di qualsiasi funzione) sono valorizzate in **ambito globale**.

```
var x = 5;

// in questa parte del codice è possibile utilizzare la variabile x

function myFunction() {
  // anche in questo blocco di codice è possibile utilizzare la variabile x
}
```

Ambito globale - Esempio

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Ambito globale JavaScript</h2>
    <p>È possibile accedere a una variabile globale da qualsiasi
    script o funzione.</p>
    <p id="demo"></p>
    <script>
      var carName = "Volvo"; /*variabile dichiarata prima
      della funzione */
      myFunction();
      function myFunction() {
        document.getElementById("demo").innerHTML =
        "Posso visualizzare" +  carName;
      }
    </script>
  </body>
</html>.
```

È possibile accedere alle variabili **globali** da qualsiasi punto in un programma JavaScript.

Ambito della funzione

Le variabili dichiarate **localmente** (all'interno di una funzione) hanno validità solo nella funzione stessa.

```
/* in questa parte del codice non è possibile
   utilizzare la variabile y */

function myFunction() {
  var y = 7;
  /* in questa parte del codice è possibile
  utilizzare la variabile y */

}

/* in questa parte del codice non è possibile
   utilizzare la variabile y */
```

Ambito della funzione - Esempio

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Block Scope</h2>
    <p>Al di fuori della funzione carName assume il valore undefined.</p>
    <p id="demo1"></p>
    <p id="demo2"></p>
    <script>
      myFunction();
      function myFunction() {
        var carName = "Volvo"; /*variabile dichiarata nello stesso blocco della
        funzione */
        document.getElementById("demo1").innerHTML =
        typeof carName + " " + carName; /*typeof restituisce il tipo della
        variabile*/
      }
      document.getElementById("demo2").innerHTML = typeof carName;
    </script>
  </body>
</html>
```

È possibile accedere alle variabili **locali** solo dall'interno della funzione in cui sono dichiarate.

Ambito delle var

Abbiamo visto che le variabili dichiarate con **var** hanno validità globale o all'interno della funzione. non possono essere utilizzate in **Block Scope**.

È possibile accedere alle variabili dichiarate all'interno di un blocco **{…}** anche dall'esterno del blocco in cui sono state dichiarate.

```
{
  var x = 2;
}
// x può essere utilizzata anche in qui
```

Block Scope

Con la versione del 2015, ES6 ha introdotto due nuove importanti parole chiave: **let** e **const**.

Queste due parole chiave forniscono variabili (e costanti) definite **Block Scope**. Ciò significa che una volta definite sono disponibili solo all'interno del blocco di istruzioni in cui sono dichiarate.

let

L'ambito delle variabili dichiarate con la parola chiave **let** è Block Scope.

Le variabili dichiarate all'interno di un blocco {…} non sono accessibili dall'esterno del blocco.

```
{
  let x = 2;
}
// x non può essere utilizzata qui
```

Esempio

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Ambiti JavaScript</h2>
    <script>
      {
       {
         var x = 5;
       }
       let y = 7;
      }
      document.write(x);
      document.write(y);
    </script>
  </body>
</html>
```

Ridichiarazione delle variabili

Quando utilizziamo **var**, ridichiarare una variabile può arrecare dei problemi.

La variabile ridichiarata all'interno di un blocco manterrà il valore assegnatole anche all'esterno del blocco.

```
var x = 10;
// in questo blocco x vale 10
{
  var x = 2;
  // in questo blocco x vale 2
}
// in questo blocco x vale 2
```

Esempio

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Dichiarazione di una variabile utilizzando var</h2>
    <p>La variabile x dichiarata all'inizio vale</p>
    <p id="prova"></p>
    <hr />
    <p>La variabile x dichiarata all'interno del blocco vale
    </p>
    <p id="prova1"></p>

    <script>
      var x = 10; // in questo blocco x vale 10
      {
        var x = 2; // in questo blocco x vale 2
        document.getElementById("prova").innerHTML = x;
      }
      // in questo blocco x vale 2
      document.getElementById("prova1").innerHTML = x;
    </script>

</body>
</html>
```

Ridichiarazione delle variabili

Utilizzando la parola chiave **let** è possibile risolvere il precedente problema.

Dichiarando nuovamente una variabile all'interno di un blocco non modificherà la variabile all'esterno del blocco.

```
var x = 10;
// in questo blocco x vale 10
{
  let x = 2;
  // in questo blocco x vale 2
}
// in questo blocco x vale 10
```

Esempio

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Dichiarazione di una variabile utilizzando let</h2>
    <p>La variabile x dichiarata all'inizio vale</p>
    <p id="prova"></p>
    <hr />
    <p>La variabile x dichiarata all'interno del blocco vale
    </p>
    <p id="prova1"></p>

    <script>
      var x = 10; // in questo blocco x vale 10
      {
        let x = 2; // in questo blocco x vale 2
        document.getElementById("prova").innerHTML = x;
      }
      // in questo blocco x vale 10
      document.getElementById("prova1").innerHTML = x;
    </script>

</body>
</html>
```

Ridichiarazione delle variabili

È possibile ridichiarare una variabile JavaScript con **var** in qualsiasi punto del codice.

```
var x = 2;
// il valore di x è 2
var x = 3;
// il valore di x è 3
```

Non è consentito dichiarare nuovamente una **var** con **let** (e viceversa) nello stesso ambito o nello stesso blocco.

```
var x = 2;
let x = 3; // non consentito

{
  var x = 4; // consentito
  let x = 5; // non consentito
}
```

Altro esempio:

```
let x = 2;
var x = 3; // non consentito

{
  let x = 4; // consentito
  var x = 5; // non consentito
}
```

Cicli

Altro problema riscontrabile con l'utilizzo di **var** è riscontrabile nei cicli.

Utilizzando **var**, la variabile dichiarata nel ciclo modifica il valore della variabile al di fuori del ciclo.

Quando viene utilizzato **let** per dichiarare una variabile in un ciclo, la variabile sarà visibile solo all'interno del ciclo.

Esempio

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Ciclo con l'utilizzo di var</h2>
    <p>Il valore della variabile i al termine del ciclo è</p>
    <p id="valore_x_var"></p>

    <script>
      var i = 5;
      for (var i = 0; i < 10; i++) {
        // ...
      }
      document.getElementById("valore_x_var").innerHTML = i;
    </script>

    <hr />
    <h2>Ciclo con l'utilizzo di let</h2>
    <p>Il valore della variabile n al termine del ciclo è</p>
    <p id="valore_x_let"></p>

    <script>
      let n = 5;
      for (let n = 0; n < 10; n++) {
        // ...
      }
      document.getElementById("valore_x_let").innerHTML = n;
    </script>

  </body>
</html>
```

Similitudini

Le variabili dichiarate con **var** e **let** sono abbastanza simili quando dichiarate all'interno di una funzione o all'esterno di un blocco.

Entrambe avranno l'ambito della funzione:

```
function myFunction() {
  var carName = "Volvo"; //Function Scope
}
function myFunction() {
  let carName = "Volvo"; //Function Scope
}
```

Entrambe avranno ambito globale:

```
var x = 2; //Global scope
let x = 2; //Global scope
```

window object

In JavaScript, l'ambito globale è l'ambiente JavaScript.
In HTML, l'ambito globale è l'oggetto finestra.
Le variabili globali definite con la parola chiave **var** appartengono all'oggetto finestra.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Variabili globali in JavaScript</h2>
    <p>In HTML, le variabili globali definite con <b>var</b>, sono
    variabili di finestra.</p>
    <p id="demo"></p>
    <script>
      var carName = "Volvo";
      // in questa parte del codice si può usare window.carName
      document.getElementById("demo").innerHTML = "Casa
      automobilistica " + window.carName;
    </script>
  </body>
</html>
```

window object

Le variabili globali definite con la parola chiave **let** non appartengono all'oggetto finestra.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Variabili globali in JavaScript</h2>
    <p>In HTML, le variabili globali definite con <b>let</b>,
    non sono variabili di finestra.</p>
    <p id="demo"></p>
    <script>
      let carName = "Volvo";
      // non è possibile utilizzare window.carName
      document.getElementById("demo").innerHTML = "Casa
      automobilistica" + window.carName;
    </script>
  </body>
</html>
```

Hoisting

Le variabili definite con **var** possono essere inizializzate in qualsiasi momento. Possiamo usare la variabile prima che venga dichiarata.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Hoisting</h2>
    <p>Con <b>var</b>, posso utilizzare una variabile prima
    che venga dichiarata:</p>
    <p id="demo"></p>
    <script>
      carName = "Volvo";
      document.getElementById("demo").innerHTML =
      "Casa automobilistica" + carName;
      var carName;
    </script>
  </body>
</html>
```

Hoisting

Le variabili definite con **let** vengono "sollevate" all'inizio del blocco, ma non inizializzate.

Significato: il blocco di codice è a conoscenza della variabile, ma non può essere utilizzato fino a quando non è stato dichiarato.

L'uso di una variabile **let** prima che venga dichiarata si tradurrà in un **ReferenceError**.

La variabile si trova in una "zona morta temporale" dall'inizio del blocco fino a quando non viene dichiarata.

Esempio

```
<!DOCTYPE html>
<html>
  <body>

    <h2>JavaScript Hoisting</h2>
    <p>Con <b>let</b>, non posso utilizzare una variabile prima che venga
    dichiarata:</p>
    <p id="demo"></p>
    <script>
      try {
        carName = "Volvo";
        let carName;
        document.getElementById("demo").innerHTML =
        "Casa automobilistica" + carName;
      }
      catch(errore) {
        document.getElementById("demo").innerHTML =
        errore.name + ": " + errore.message;
      }
    </script>
  </body>
</html>
```

const

Le variabili definite con **const** hanno lo stesso comportamento delle variabili **let**, tranne per il fatto che non possono essere riassegnate.

```
const PI = 3.141592653589793;
PI = 3.14; // questo darà un errore
PI = PI + 10; // anche questo darà un errore
```

Esempio

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript const</h2>
    <p>Non è possibile modificare un valore primitivo.</p>
    <p id="demo"></p>
    <script>
    try {
      const PI = 3.141592653589793;
      PI = 3.14;
    }
    catch (errore) {
      document.getElementById("demo").innerHTML = errore;
    }
    </script>
  </body>
</html>
```

const

Il comportamento delle variabili **const** è simile a **let** anche quando si tratta di **Block Scope** .

```
var x = 10;
// x vale 10
{
  const x = 2;
  // x vale 2
}
// x vale 10
```

const

Il valore delle variabili **const** deve essere assegnato in fase di dichiarazione:

```
const PI = 3.14159265359;
```

Scrivere:

```
const PI;
PI = 3.14159265359;
```

è errato.

const

La parola chiave **const** è un po' fuorviante.

NON definisce un valore costante. Definisce un riferimento costante a un valore.

Per questo motivo, non possiamo modificare i valori primitivi costanti, ma possiamo modificare le proprietà degli oggetti costanti.

Come precedentemente visto, se assegniamo un valore primitivo a una costante, non possiamo cambiare il valore primitivo:

```
const PI = 3.141592653589793;
PI = 3.14; // questo darà un errore
PI = PI + 10; // anche questo darà un errore
```

const

È, invece, possibile modificare le proprietà di un oggetto costante.

```
// creo un oggetto const:
const car = {type:"Fiat", model:"500",
color:"bianca"};

// modifico una proprietà:
car.color = "rossa";

// aggiungo una proprietà:
car.owner = "Marco";
```

const

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript const</h2>
    <p>La dichiarazione di un oggetto costante NON rende le proprietà
    dell'oggetto immutabili:</p>
    <p id="demo"></p>
    <script>
      // creo un oggetto:
      const car = {type:"Fiat", model:"500", color:"bianca"};
      // modifico una proprietà:
      car.color = "rossa";
      // aggiungo una proprietà:
      car.owner = "Marco";
      // visualizzo le proprietà:
      document.getElementById("demo").innerHTML =
      "Marca " + car.type + "; Modello " + car.model + ";
      Colore " + car.color + "; Proprietario " + car.owner;
    </script>
  </body>
</html>
```

const

Non è possibile riassegnare un valore ad una costante.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript const</h2>
    <p>Non è è possibile riassegnare un oggetto costante:</p>
    <p id="demo"></p>
    <script>
      try {
        const car = {type:"Fiat", model:"500",
        color:"bianca"};
        car = {type:"Volvo", model:"EX60", color:"rossa"};
      }
      catch (errore) {
        document.getElementById("demo").innerHTML = errore;
      }
    </script>
  </body>
</html>
```

const

È possibile modificare gli elementi di un array costante.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript const</h2>
    <p>La dichiarazione di un array costante NON rende gli elementi
    immutabili:</p>
    <p id="demo"></p>
    <script>
      // creo un Array:
      const cars = ["Saab", "Volvo", "BMW"];
      // modifico il primo elemento:
      cars[0] = "Toyota";
      // aggiungo un elemento:
      cars.push("Audi");
      // visualizzo l'Array:
      document.getElementById("demo").innerHTML = cars;
    </script>
  </body>
</html>
```

const

Come per le variabili costanti, non è possibile riassegnare un array costante.

```
<html>
  <body>
    <h2>JavaScript const</h2>
    <p>Non è possibile riassegnare un array costante:</p>
    <p id="demo"></p>
    <script>
      try {
        const cars = ["Saab", "Volvo", "BMW"];
        cars = ["Toyota", "Volvo", "Audi"];
      }
      catch (errore) {
        document.getElementById("demo").innerHTML = errore;
      }
    </script>
  </body>
</html>
```

Ridichiarazione di una const

Non è consentito ridichiarare o riassegnare una variabile esistente **var** o **let** verso una **const**, nello stesso ambito o nello stesso blocco.

```
var x = 2;    // consentito
const x = 2; // non consentito
{
  let x = 2;    // consentito
  const x = 2; // non consentito
}
```

Ridichiarazione di una const

Non è consentito ridichiarare o riassegnare una **const** esistente nello stesso ambito o nello stesso blocco.

```
const x = 2; // consentito
const x = 3; // non consentito
x = 3;       // non consentito
var x = 3;   // non consentito
let x = 3;   // non consentito

{
  const x = 2; // consentito
  const x = 3; // non consentito
  x = 3;       // non consentito
  var x = 3;   // non consentito
  let x = 3;   // non consentito
}
```

Ridichiarazione di una const

È consentito ridichiarare una **const**, in un altro ambito o in un altro blocco.

```
const x = 2;    // consentito
{
  const x = 3; // consentito
}
{
  const x = 4; // consentito
}
```

Hoisting

Come per le variabili definite con **let**, anche le variabili **const** vengono "sollevate" all'inizio del blocco, ma non inizializzate.

Significato: il blocco di codice è a conoscenza della variabile, ma non può essere utilizzato fino a quando non è stato dichiarato.

La variabile si trova in una "zona morta temporale" dall'inizio del blocco fino a quando non viene dichiarata.

L'uso di una variabile **const** prima che venga dichiarata è un errore di sintassi, quindi il codice non verrà eseguito.

Hoisting

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Hoisting</h2>
    <p>Con <b>const</b>, non è possibile utilizzare
    una variabile prima che venga dichiarata.</p>
    <p id="demo">testo non modificato</p>
    <script>
      carName = "Volvo";
      const carName;
      document.getElementById("demo").innerHTML =
      carName;
    </script>
  </body>
</html>
```

OPERATORI ARITMETICI

Assegnazione =

Gli operatori di assegnazione attribuiscono valori alle variabili.

Operatore	esempio	equivale a
=	x = y	x = y
+=	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
**=	x **= y	x = x ** y

Assegnazione

L'operatore di **assegnazione dell'addizione** (+=) aggiunge un valore a una variabile.

```
<script>
  var x = 10;
  x += 5;
  document.getElementById("demo").innerHTML
  = x;
</script>
```

Assegnazione

Può essere utilizzato anche per aggiungere (concatenare) stringhe:

```
<script>
  txt1 = "Oggi è ";
  txt1 += "una bella giornata";
  document.getElementById("demo").innerHTML =
  txt1;
</script>
```

Ma anche:

```
<script>
  var txt1 = "Marco";
  var txt2 = "Polo";
  document.getElementById("demo").innerHTML =
  txt1 + " " + txt2;
</script>
```

Quando viene utilizzato su stringhe, l'operatore + è chiamato operatore di concatenazione.

Assegnazione

L'addizione di due numeri restituirà la somma, ma l'addizione di un numero e una stringa restituirà una stringa.

```
var x = 5 + 5;       // restituisce 10
var y = "5" + 5;     // restituisce 55
var z = "Ciao" + 5; // restituisce Ciao5
```

Operatori aritmetici

Gli operatori aritmetici eseguono operazioni aritmetiche sui numeri (letterali o variabili).

Operatore	Descrizione
+	Addizione
-	Sottrazione
*	Moltiplicazione
**	Potenza (ES2016)
/	Divisione
%	Modulo (Resto)
++	Incremento
--	Decremento

TIPI DI DATI

Tipi di dati

Le variabili JavaScript possono contenere molti **tipi di dati**: numeri, stringhe, oggetti e altro.

```
var length = 16;                // Numero
var lastName = "Johnson";       // Stringa
var x = {nome:"Marco",
cognome:"Polo"};                // Oggetto
```

Tipi di dati

Nella programmazione, i tipi di dati sono un concetto importante. Per poter operare sulle variabili, è importante sapere qualcosa sul tipo. Senza specificare i tipi di dati, un computer non può risolvere questo problema in modo sicuro:

```
var x = 16 + "Volvo";
```

JavaScript tratterà l'esempio sopra come:

```
var x = "16" + "Volvo";
```

Tipi di dati

JavaScript valuta le espressioni da sinistra a destra.
Diverse sequenze possono produrre risultati diversi:

```
var x = 16 + 4 + "Volvo"; // 20Volvo
var x = "Volvo" + 16 + 4; // Volvo164
```

Nel primo esempio, JavaScript tratta 16 e 4 come numeri, finché non raggiunge "Volvo".
Nel secondo esempio, poiché il primo operando è una stringa, tutti gli operandi vengono trattati come stringhe.

Tipi dinamici

JavaScript tratta i tipi di dato dinamicamente. Ciò significa che la stessa variabile può essere utilizzata per contenere diversi tipi di dato:

```
var x;          // x è undefined
x = 5;          // x è un numero
x = "Marco"; // x è una stringa
```

Stringhe

Una stringa (o una stringa di testo) è una serie di caratteri come "Marco Polo".
Le stringhe sono scritte tra virgolette singole o doppie:

```
var car1 = "Volvo"; // virgolette doppie
var car2 = 'Volvo'; // virgolette singole
```

È possibile usare le virgolette all'interno di una stringa, a condizione che non corrispondano alle virgolette che racchiudono la stringa:

```
var answer1 = "Un'altra persona";
var answer2 = "Si chiama 'Marco'";
var answer3 = 'Si chiama "Marco"';
```

Numeri

In JavaScript esiste un solo tipo di numeri. I numeri possono essere scritti con o senza decimali:

```
var x1 = 34.00;
var x2 = 34;
```

Numeri molto grandi o molto piccoli possono essere scritti con notazione scientifica (esponenziale):

```
var y = 123e5;  // 12300000
var z = 123e-5; // 0.00123
```

Booleani

I booleani possono avere solo due valori: **true** o **false**.

```
<script>
   var x = 5;
   var y = 5;
   var z = 6;
   document.write((x == y) + "<br />" +
   (x == z));
</script>
```

Visualizzeremo:
true
false

Array

In JavaScript gli array sono scritti con parentesi quadre.
Gli elementi della matrice sono separati da virgole.
Il codice seguente dichiara (crea) un array chiamato cars, contenente tre elementi (nomi di auto):

```
var cars = ["Saab", "Volvo", "BMW"];
```

Gli indici della matrice sono a base zero, il che significa che il primo elemento è [0], il secondo è [1] e così via.

Oggetti

Gli oggetti JavaScript sono scritti con parentesi graffe {}. Le proprietà degli oggetti vengono scritte come coppie **nome:valore**, separate da virgole.

```
<script>
  var persona = {
    nome : "Marco",
    cognome  : "Polo",
    anni     : 50,
    altezza  : 175
  };
  document.write (persona.nome + " ha " +
  persona.anni + " anni.");
</script>
```

typeof

È possibile utilizzare l'operatore **typeof** per conoscere il tipo di una variabile JavaScript.
L'operatore **typeof** restituisce il tipo di una variabile o di un'espressione:

```
<script>
  var x = 50
  document.write (typeof "Marco" + "<br />" +
  typeof x + "<br />" +
  typeof (50**2));
</script>
```

Risultato:

```
string
number
number
```

undefined

In JavaScript, una variabile senza valore assume il valore undefined. Anche il tipo sarà undefined.

```
<script>
  var car;
  document.write(car + "<br />" +
  typeof car);
</script>
```

Risultato:

```
undefined
undefined
```

undefined

Qualsiasi variabile può essere svuotata, impostando il valore su undefined.

```
<script>
  var car = "Volvo";
  car = undefined;
  document.write(car + "<br />" +
  typeof car);
</script>
```

Risultato:

```
undefined
undefined
```

Empty Values

Un valore vuoto non non è un valore undefined.
Una stringa vuota ha sia un valore che un tipo.

```
<script>
  var car = "";
  document.write("Il valore è: " + car +
  "<br />" + "Il tipo è: " + typeof car);
</script>
```

Risultato:

```
Il valore è:
Il tipo è: string
```

null

In JavaScript **null** è un oggetto.
È possibile svuotare un oggetto impostandolo su null…

```
<script>
  var persona = {nome:"Marco",
  cognome:"Polo", anni:50, altezza:175};
  persona = null;
  document.write(typeof persona);
  var x = 50;
  x = null;
  document.write("<br />" + typeof x);
</script>
```

Risultato:

```
object
object
```

Svuotare un oggetto

È possibile svuotare un oggetto impostandolo su null.

```
<script>
  var persona = {
    nome      : "Marco",
    cognome   : "Polo",
    anni      : 50,
    altezza   : 175
  };
  persona = null;
  document.write (typeof persona);
</script>
```

Risultato:

```
object
```

Svuotare un oggetto

…o impostandolo su undefined.

```
<script>
  var persona = {
    nome      : "Marco",
    cognome   : "Polo",
    anni      : 50,
    altezza   : 175
  };
  persona = undefined;
  document.write (typeof persona);
</script>
```

Risultato:

```
undefined
```

Differenza tra undefined e null

Undefined e **null** sono di valore uguale ma sono di tipo diverso.

```
<script>
  document.write(
  typeof undefined + "<br />" +
  typeof null + "<hr />" +
  (null === undefined) + //ugual valore e
stesso tipo
  "<br>" +
  (null == undefined));
</script>
```

Risultato:

```
undefined
object
```

```
false
true
```

Dati primitivi

In **JavaScript**, ci sono 4 tipi di **dati primitivi**: string, number, boolean, undefined.

```
<script>
  document.write(
  typeof "john" + "<br />" +
  typeof 3.14 + "<br />" +
  typeof true + "<br />" +
  typeof false + "<br />" +
  typeof x);
</script>
```

Risultato:

```
string
number
boolean
boolean
undefined
```

Dati complessi

L'operatore typeof può restituire uno di due tipi complessi:
- object
- Function

Restituisce "object" per oggetti, array e null.
Restituisce "function" per le funzioni.

```
<script>
  document.write(
  typeof {nome:'Marco', anni:34} + "<br />" +
  typeof [1,2,3,4] + "<br />" +
  typeof null + "<br />" +
  typeof function myFunc(){});
</script>
```

Risultato:

```
object
object
object
function
```

FUNZIONI

Funzioni

Una funzione è un blocco di codice progettato per eseguire una particolare attività.
In JavaScript una funzione viene eseguita quando "qualcosa" la invoca (la chiama).

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Functions</h2>
    <p>Questo esempio richiama una funzione che esegue un
    calcolo e restituisce il risultato:</p>
    <p id="demo"></p>
    <script>
      function myFunction(p1, p2) {
        return p1 * p2;
      }
      document.getElementById("demo").innerHTML =
      myFunction(4, 3);
    </script>
  </body>
</html>
```

Sintassi

Una funzione è definita con la parola chiave function, seguita da un **nome**, seguito da parentesi **()**.
I nomi delle funzioni possono contenere lettere, cifre, trattini bassi e segni di dollaro (stesse regole delle variabili).
Le parentesi possono includere nomi di parametri separati da virgole:
(*parametro1, parametro2, ...*)
Il codice della funzione da eseguire è posto tra parentesi graffe: **{}**

```
function name(p1, p2, p3) {
  // codice da eseguire
}
```

Sintassi

I **parametri** della funzione sono elencati tra parentesi () nella definizione della funzione.
Gli **argomenti** della funzione sono i **valori** ricevuti dalla funzione quando viene richiamata.
All'interno della funzione, gli argomenti (i parametri) si comportano come variabili locali.
Una funzione è molto simile a una procedura o una subroutine, in altri linguaggi di programmazione.

Invocazione

Il codice all'interno della funzione verrà eseguito quando "qualcosa" **invoca** (chiama) la funzione:

- Quando si verifica un evento (quando un utente fa clic su un pulsante)
- Quando viene invocato (chiamato) dal codice JavaScript
- Automaticamente (auto invocato)

return

Quando il codice JavaScript raggiunge un'istruzione **return**, la funzione interromperà l'esecuzione.
Le funzioni spesso calcolano un **valore da restituire**. Il valore viene "restituito" al "chiamante":

```
<script>
  var x = myFunction(4, 3);
  document.write("x = " + x);
  function myFunction(a, b) {
    return a * b;
  }
</script>
```

Utilizzo

È possibile utilizzare lo stesso codice molte volte con argomenti diversi, per produrre risultati diversi: definisci il codice una volta e usalo molte volte.

```
<script>
  function toCelsius(fahrenheit) {
    return (5/9) * (fahrenheit - 32);
  }
  document.write( toCelsius(86) );
</script>
```

L'operatore ()

Utilizzando l'esempio precedente, **toCelsius** si riferisce all'oggetto della funzione e **toCelsius()** si riferisce al risultato della funzione.
L'accesso a una funzione senza **()** restituirà l'oggetto funzione invece del risultato della funzione.

```
<script>
  function toCelsius(f) {
    return (5/9) * (f-32);
  }
  document.write(toCelsius);
</script>
```

Risultato:

```
function toCelsius(f){ return (5/9) * (f-32); }
```

Funzioni utilizzate come valori variabili

Le funzioni possono essere utilizzate allo stesso modo delle variabili, in tutti i tipi di formule, assegnazioni e calcoli. Invece di utilizzare una variabile per memorizzare il valore restituito di una funzione:

```
var x = toCelsius(77);
var text = "La temperatura è " + x + "° Celsius";
```

è possibile utilizzare la funzione direttamente, come valore variabile:

```
var text = "La temperatura è " + toCelsius(77) + "° Celsius";
```

Variabili locali

Le variabili dichiarate all'interno di una funzione JavaScript diventano **LOCALI** per la funzione.
È possibile accedere alle variabili locali solo dall'interno della funzione.

```
/* in questa parte del codice non è
possibile utilizzare la variabile carName */
function myFunction() {
  var carName = "Volvo";
   /* in questa parte del codice è possibile
   utilizzare la variabile carName */
}
/* in questa parte del codice non è
possibile utilizzare la variabile carName */
```

Variabili locali

Poiché le variabili locali vengono riconosciute solo all'interno delle loro funzioni, le variabili con lo stesso nome possono essere utilizzate in funzioni diverse.
Le variabili locali vengono create all'avvio di una funzione e cancellate quando la funzione viene completata.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Funzioni JavaScript</h2>
    <p> Al di fuori di myFunction() carName non è definito.</p>
    <p id="demo1"></p>
    <p id="demo2"></p>
    <script>
      myFunction();
      function myFunction() {
        var carName = "Volvo";
        document.getElementById("demo1").innerHTML = typeof carName +
        " " + carName;
      }
      document.getElementById("demo2").innerHTML = typeof carName;
    </script>
  </body>
</html>
```

OGGETTI

Oggetti

Nella vita reale, un'auto è un **oggetto** .
Un'auto ha **proprietà** come il peso e il colore e **metodi** come l'avvio e l'arresto.

Oggetto	Proprietà	Metodi
	car.nome = Fiat	car.start()
	car.modello = 500	car.drive()
	car.peso = 850kg	car.brake()
	car.colore = bianco	car.stop()

Tutte le auto hanno le stesse **proprietà**, ma i **valori** delle proprietà differiscono da auto a vettura.
Tutte le auto hanno gli stessi **metodi**, ma i metodi vengono eseguiti **in momenti diversi**.

Oggetti

Abbiamo visto che le variabili sono contenitori per i valori dei dati.

```
var car = "Fiat";
```

Anche gli oggetti sono variabili. Ma gli oggetti possono contenere molti valori.

```
var car = {type:"Fiat", model:"500",
color:"white"};
```

I valori vengono scritti come coppie **nome:valore** (nome e valore separati da due punti).

Definizione dell'oggetto

Definiamo (e creiamo) un oggetto JavaScript con un oggetto letterale:

```
var persona = {nome:"Marco", cognome:"Polo",
anni:50, coloreOcchi:"blu"};
```

Gli spazi e le interruzioni di riga non sono importanti. Una definizione di oggetto può estendersi su più righe:

```
var persona = {
  nome:"Marco",
  cognome:"Polo",
  anni:50,
  coloreOcchi:"blu"
};
```

Proprietà

Le coppie **nome:valore** negli oggetti JavaScript sono chiamate **proprietà**.

Proprietà	Valore della proprietà
nome	Marco
cognome	Polo
anni	50
colore degli occhi	blu

Accesso alle proprietà

È possibile accedere alle proprietà degli oggetti in due modi:

- *objectName.propertyName*

```
// visualizza alcuni dati dell'oggetto:
document.write(persona.nome + " " +
persona.anni);
```

- *objectName["propertyName"]*

```
// visualizza alcuni dati dell'oggetto:
document.write(persona["cognome"] + " " +
persona["coloreOcchi"]);
```

Metodi

Gli oggetti possono anche avere dei **metodi**.

I metodi sono **azioni** che possono essere eseguite sugli oggetti.

I metodi vengono memorizzati nelle proprietà come **definizioni di funzioni**.

Proprietà	Valore della proprietà
nome	Marco
cognome	Polo
anni	50
colore degli occhi	blu
nome e cognome	function() {return this.nome + " " + this.cognome;}

Un metodo è una funzione memorizzata come proprietà.

Esempio

```
var persona = {
   nome: "Marco",
   cognome: "Polo",
   id: 5566,
   fullName : function(){
      return this.nome + " " + this.cognome;
   }
};
```

this

Nella definizione di una funzione, **this** si riferisce al "proprietario" della funzione.

Nell'esempio sopra, **this** è l'**oggetto persona** che "possiede" la funzione fullName.

In altre parole, this.nome indica la proprietà **nome** di **questo oggetto**.

Accesso ai metodi degli oggetti

Si accede a un metodo dell'oggetto con la seguente sintassi:
objectName.methodName()

```
<script>
  // creo un oggetto:
  var persona = {
    nome: "Marco",
    cognome: "Polo",
    id: 5566,
    fullName: function() {
      return this.nome + " " + this.cognome;
    }
  };
  // visualizzo i dati di un oggetto:
  document.write(persona.fullName());
</script>
```

Accesso ai metodi degli oggetti

Accedendo a un metodo **senza** le parentesi (), verrà restituita la **definizione** della **funzione**:

```
<script>
  // creo un oggetto:
  var persona = {
    nome: "Marco",
    cognome: "Polo",
    id: 5566,
    fullName: function() {
      return this.nome + " " + this.cognome;
    }
  };
  // visualizzo i dati di un oggetto:
  document.write(persona.fullName);
</script>
```

Risultato:

```
fullName: function() { return this.nome + " " +
this.cognome; }
```

Stringhe, numeri e booleani

Quando una variabile viene dichiarata con la parola chiave "**new**", la variabile viene creata come oggetto:

```
var x = new String(); /* Dichiara x come un oggetto String */
var y = new Number(); /* Dichiara x come un oggetto Number */
var z = new Boolean(); /* Dichiara x come un oggetto Boolean */
```

È meglio evitare oggetti String, Number e Boolean in quanto complicano il codice e rallentano la velocità di esecuzione.

EVENTI

Eventi

Gli eventi HTML sono **"cose"** che accadono agli elementi HTML.
Quando viene utilizzato JavaScript nelle pagine HTML, il codice JavaScript può **"reagire"** a questi eventi.
Un evento HTML può essere qualcosa che fa il browser o qualcosa che fa un utente.
Alcuni esempi di eventi HTML sono:

- È terminato il caricamento di una pagina Web HTML
- È stato modificato un campo di input HTML
- È stato fatto clic su un pulsante HTML

Spesso, quando accadono degli eventi, potresti voler fare qualcosa.
JavaScript consente di eseguire il codice quando vengono rilevati eventi.

Eventi

HTML consente di aggiungere attributi del gestore eventi, **con codice JavaScript**, agli elementi HTML.

Con virgolette singole:

```
<element event='some JavaScript'>
```

Con virgolette doppie:

```
<element event="some JavaScript">
```

Esempio

Nell'esempio seguente, un attributo onclick (con codice) viene aggiunto a un elemento <button>:

```
<!DOCTYPE html>
<html>
  <body>
    <button onclick =
    "document.getElementById('demo').innerHTML =
    Date()">The time is?</button>
    <p id="demo"></p>
  </body>
</html>
```

Il codice JavaScript modifica il contenuto dell'elemento p con id = "demo".

Esempio

In quest'altro esempio, il codice modifica il contenuto del proprio elemento (utilizzando **this**.innerHTML):

```
<!DOCTYPE html>
<html>
  <body>
    <button onclick =
    "this.innerHTML =
    Date()">The time is?</button>
  </body>
</html>
```

Sintassi

Il codice JavaScript è spesso lungo diverse righe.
È comune vedere gli attributi degli eventi che chiamano funzioni:

```
<!DOCTYPE html>
<html>
  <body>
    <p>Clicca il bottone per visualizzare la data.
    </p>
    <button onclick="displayDate()">Oggi è?</button>
    <script>
      function displayDate() {
        document.getElementById("demo").innerHTML =
        Date();
      }
    </script>
    <p id="demo"></p>
  </body>
</html>
```

Eventi comuni

Di seguito è riportato un elenco di alcuni eventi HTML comuni:

Evento	Descrizione
onchange	È stato modificato un elemento HTML
onclick	L'utente fa clic su un elemento HTML
onmouseover	L'utente sposta il mouse su un elemento HTML
onmouseout	L'utente allontana il mouse da un elemento HTML
onkeydown	L'utente preme un tasto della tastiera
onload	Il browser ha terminato il caricamento della pagina

Cosa può fare JavaScript

I gestori di eventi possono essere utilizzati per gestire e verificare l'input dell'utente, le azioni dell'utente e le azioni del browser:

- Cose da fare ogni volta che viene caricata una pagina
- Cose che dovrebbero essere fatte quando la pagina è chiusa
- Azione da eseguire quando un utente fa clic su un pulsante
- Contenuto che dovrebbe essere verificato quando un utente inserisce dati
- E altro ancora ...

È possibile utilizzare molti metodi diversi per consentire a JavaScript di funzionare con gli eventi:

- Gli attributi dell'evento HTML possono eseguire direttamente il codice JavaScript
- Gli attributi degli eventi HTML possono chiamare funzioni JavaScript
- È possibile assegnare le proprie funzioni del gestore di eventi agli elementi HTML
- È possibile impedire che gli eventi vengano inviati o gestiti
- E altro ancora ...

STRINGHE

Stringhe

Le stringhe vengono utilizzate per memorizzare e manipolare il testo. In JavaScript una stringa è composta da zero o più caratteri scritti tra virgolette.

```
var x = "Questa è una stringa";
```

Si possono usare virgolette singole o doppie:

```
var carName1 = "Volvo XC60";  // virgolette doppie
var carName2 = 'Volvo XC60';  // virgolette singole
```

È possibile usare le virgolette all'interno di una stringa, a condizione che non corrispondano alle virgolette che circondano la stringa:

```
var answer1 = "It's alright";
var answer2 = "Si chiama 'Marco'";
var answer3 = 'Si chiama "Marco"';
```

lenght

Per ricavare la lunghezza di una stringa, bisogna utilizzare la proprietà incorporata **length**:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Proprietà di String in JavaScript</h2>
    <p>La proprietà length restituisce la
    lunghezza di una stringa:</p>
    <p id="demo"></p>
    <script>
      var testo = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
      var stringLenghtn = testo.length;
      document.getElementById("demo").innerHTML =
      stringLenghtn;
    </script>
  </body>
</html>
```

Carattere di uscita \

Poiché le stringhe devono essere scritte tra virgolette, JavaScript fraintenderà la stringa:

```
var x = "Noi siamo i cosiddetti "Vichinghi" del nord.";
```

La stringa verrà tagliata in "Noi siamo i cosiddetti".
La soluzione per evitare questo problema è utilizzare il **carattere di uscita ** (**backslash**).
Il carattere di uscita backslash (\) trasforma i caratteri speciali in caratteri stringa.

Codice	Risultato	Descrizione
\'	'	Virgolette singole
\"	"	Virgolette doppie
\\	\	Backslash

Carattere di uscita \

La sequenza **\"** inserisce una virgoletta doppia in una stringa:

```
var x = "Noi siamo i cosiddetti
\"Vichinghi\" del nord.";
```

La sequenza **\'** inserisce una virgoletta singola in una stringa:

```
var x = 'Si puo\' fare!';
```

La sequenza **** inserisce una barra rovesciata in una stringa:

```
var x = "Il carattere \\ si chiama
backslash.";
```

Linee troppo lunghe

Per una migliore leggibilità, i programmatori preferiscono evitare le righe di codice più lunghe di 80 caratteri.
Se un'istruzione JavaScript non si adatta a una riga, il posto migliore per interromperla è dopo un operatore:

```
document.getElementById("demo").innerHTML =
"Hello Dolly!";
```

Si può anche suddividere una riga di codice **all'interno di una stringa di testo** con una singola barra rovesciata:

```
document.write( "Hello \
Dolly!";
```

Tale metodo non è il preferito. Potrebbe non avere un supporto universale. Alcuni browser non consentono spazi dietro il carattere \.
Un modo più sicuro per suddividere una stringa è quello di usare l'aggiunta di stringhe:

```
document.write( "Hello " +
"Dolly!";
```

Non è possibile suddividere una riga di codice con una barra rovesciata:

```
document.getElementById("demo").innerHTML = \
"Hello Dolly!";
```

Stringhe come oggetti

Normalmente, le stringhe JavaScript sono valori primitivi, creati da letterali:

```
var nome = "Marco";
```

Ma le stringhe possono anche essere definite come oggetti con la parola chiave **new**:

```
var nome = new String("Marco");
```

Creare stringhe come oggetti rallenta la velocità di esecuzione.

Stringhe come oggetti

La parola chiave **new** complica il codice. Questo può produrre alcuni risultati inaspettati.
Quando si utilizza l'operatore **==**, le stringhe con lo stesso contenuto sono uguali:

```
<script>
  var x = "Marco"; // x è una stringa
  var y = new String("Marco");
  // y è un oggetto
  document.write(x==y);
</script>
```

Risultato:

```
true
```

Stringhe come oggetti

Quando si utilizza l'operatore **===**, i valori con lo stesso contenuto potrebbero non essere uguali, perché l'operatore === si aspetta l'uguaglianza sia nel tipo di dati che nel valore:

```
<script>
  var x = "Marco"; // x è una stringa
  var y = new String("Marco");
  // y è un oggetto
  document.write(x===y);
</script>
```

Risultato:

```
false
```

Stringhe come oggetti

Può andare anche peggio. Gli oggetti non possono essere confrontati:

```
<script>
   var x = new String("Marco");
   var y = new String("Marco");
   document.write ((x==y) + "<br /> " +
   (x===y));
</script>
```

Risultato:

```
false
false
```

METODI DELLE STRINGHE

Metodi delle stringhe

Tutti i metodi stringa restituiscono una nuova stringa. Non modificano la stringa originale.
Le stringhe sono immutabili: non possono essere cambiate, ma solo sostituite.

Metodi delle stringhe: lenght

I valori primitivi, come "Marco Polo", non possono avere proprietà o metodi (perché non sono oggetti).
Ma con JavaScript, metodi e proprietà sono disponibili anche per i valori primitivi, perché JavaScript tratta i valori primitivi come oggetti durante l'esecuzione di metodi e proprietà.

Abbiamo già visto la proprietà **lenght** che restituisce la lunghezza di una stringa.

```
<script>
  var txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  var sln = txt.length;
  document.write( sln );
</script>
```

indexOf()

Il metodo **indexOf()** restituisce l'indice (della posizione) della pima occorrenza di un testo specificato in una stringa:

```
<script>
  var str = "Individua il punto in cui si
  trova la parola 'punto'!";
  var pos = str.indexOf("punto");
  document.write( pos );
</script>
```

Risultato:

```
13
```

JavaScript conta le posizioni da zero. In una stringa 0 è la prima posizione, 1 è la seconda, 2 è la terza...

lastIndexOf()

Il metodo **lastIndexOf()** restituisce l'indice dell'ultima occorrenza di un testo specificato in una stringa:

```
<script>
  var str = "Individua il punto in cui si
  trova la parola 'punto'!";
  var pos = str.lastIndexOf("punto");
  document.write( pos );
</script>
```

Risultato:

```
46
```

Sia **indexOf()** che **lastIndexOf()** restituiscono -1 se il testo non viene trovato.

Parametro aggiuntivo

Entrambi i metodi accettano un secondo parametro come posizione iniziale per la ricerca:

```
<script>
  var str = "Individua il punto in cui si
  trova la parola 'punto'!";
  var pos = str.indexOf("punto", 14);
  document.write( pos );
</script>
```

Risultato:

```
46
```

La ricerca inizia dalla 14ª posizione.

Parametro aggiuntivo

Il metodo lastIndexOf() ricerca all'indietro (dalla fine all'inizio), ovvero: se il secondo parametro è 16, la ricerca inizia dalla posizione 16 e cerca fino all'inizio della stringa.

```
<script>
  var str = "Individua il punto in cui si
  trova la parola 'punto'!";
  var pos = str.lastIndexOf("punto", 16);
  document.write( pos );
</script>
```

Risultato:

```
13
```

La ricerca inizia dalla 16ª posizione verso l'inizio della stringa.

search()

Il metodo **search()** cerca una stringa per un valore specificato e restituisce la posizione della corrispondenza:

```
<script>
  var str = "Individua il punto in cui si
  trova la parola 'punto'!";
  var pos = str.search("punto");
  document.write( pos );
</script>
```

Risultato:

```
13
```

Quali sono le differenze tra i due metodi, indexOf() e search()?

- Il metodo search() non può accettare un secondo argomento della posizione iniziale.
- Il metodo indexOf() non può accettare valori di ricerca potenti (espressioni regolari).

slice()

Il metodo **slice()** estrae una parte di una stringa e restituisce la parte estratta in una nuova stringa.
Il metodo accetta 2 parametri: la posizione iniziale e la posizione finale (fine non inclusa).
Questo esempio estrae una porzione di una stringa dalla posizione 6 alla posizione 11 (12-1):

```
<script>
  var str = "Mela, Banana, Kiwi";
  var res = str.slice(6,12);
  document.write( res );
</script>
```

Risultato:

```
Banana
```

slice()

Se un parametro è negativo, la posizione viene contata dalla fine della stringa.

Questo esempio estrae una porzione di una stringa dalla posizione -12 alla posizione -6:

```
<script>
  var str = "Mela, Banana, Kiwi";
  var res = str.slice(-12,-6);
  document.write( res );
</script>
```

Risultato:

```
Banana
```

slice()

Omettendo il secondo parametro, il metodo taglierà il resto della stringa:

```
<script>
  var str = "Mela, Banana, Kiwi";
  var res = str.slice(6);
  document.write( res );
</script>
```

Risultato:

```
Banana, Kiwi
```

Ma anche:

```
<script>
  var str = "Mela, Banana, Kiwi";
  var res = str.slice(-12);
  document.write( res );
</script>
```

Risultato:

```
Banana, Kiwi
```

substring ()

Il metodo **substring()** è simile a slice().
La differenza è che substring() non può accettare indici negativi:

```
<script>
  var str = "Mela, Banana, Kiwi";
  var res = str.substring(6,12);
  document.write( res );
</script>
```

Risultato:

```
Banana
```

Anche per substring() omettendo il secondo parametro verrà estratto il resto della stringa.

substr ()

Anche il metodo **substr()** è simile a slice().

In substr() il secondo parametro specifica la lunghezza della parte estratta.

```
<script>
  var str = "Mela, Banana, Kiwi";
  var res = str.substr(6,6);
  document.write( res );
</script>
```

Risultato:

```
Banana
```

Omettendo il secondo parametro, substr() estrarrà il resto della stringa. Se il primo parametro è negativo, la posizione verrà presa dalla fine della stringa.

replace()

Il metodo **replace()** sostituisce un valore specificato con un altro valore in una stringa:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript String Methods</h2>
    <p>Modifica "Gramsci" con "Buzzi" nel paragrafo
    sottostante:</p>
    <button onclick="myFunction()">Cliccami</button>
    <p id="demo">Visita il Gramsci!</p>
    <script>
      function myFunction() {
        var str = document.getElementById("demo").innerHTML;
        var txt = str.replace("Gramsci","Buzzi");
        document.getElementById("demo").innerHTML = txt;
      }
    </script>
  </body>
</html>
```

Il metodo replace() non cambia la stringa su cui è chiamato. Restituisce una nuova stringa.

replace()

Per impostazione predefinita, il metodo **replace()** sostituisce solo la prima corrispondenza:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript String Methods</h2>
    <p>Modifica "Gramsci" con "Buzzi" nel paragrafo
    sottostante:</p>
    <button onclick="myFunction()">Cliccami</button>
    <p id="demo">Visita il Gramsci ed il Gramsci</p>
    <script>
      function myFunction() {
        var str = document.getElementById("demo").innerHTML;
        var txt = str.replace("Gramsci","Buzzi");
        document.getElementById("demo").innerHTML = txt;
      }
    </script>
  </body>
</html>
```

replace()

Il metodo **replace()** distingue tra maiuscole e minuscole. La scrittura di GRAMSCI (con lettere maiuscole) non funzionerà.
Per sostituire senza distinzione tra maiuscole e minuscole, bisogna utilizzare **un'espressione regolare** con un flag /i (insensibile):

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript String Methods</h2>
    <p>Modifica "Gramsci" con "Buzzi" nel paragrafo
    sottostante:</p>
    <button onclick="myFunction()">Cliccami</button>
    <p id="demo">Visita il Gramsci!</p>
    <script>
      function myFunction() {
        var str = document.getElementById("demo").innerHTML;
        var txt = str.replace(/GRAMSCI/i, "Buzzi");
        document.getElementById("demo").innerHTML = txt;
      }
    </script>
  </body>
</html>
```

replace()

Per sostituire tutte le corrispondenze, si deve utilizzare **un'espressione regolare** con un flag /g (corrispondenza globale):

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript String Methods</h2>
    <p>Modifica "Gramsci" con "Buzzi" nel paragrafo
    sottostante:</p>
    <button onclick="myFunction()">Cliccami</button>
    <p id="demo">Visita il Gramsci ed il Gramsci!</p>
    <script>
      function myFunction() {
        var str = document.getElementById("demo").innerHTML;
        var txt = str.replace(/Gramsci/g,"Buzzi");
        document.getElementById("demo").innerHTML = txt;
      }
    </script>
  </body>
</html>
```

toUpperCase() e toLowerCase()

Una stringa viene convertita in maiuscolo con **toUpperCase()**:

```
<!DOCTYPE html>
<html>
  <body>
    <p>Convertire una string in maiuscolo:</p>
    <button onclick="myFunction()">Cliccami</button>
    <p id="demo">Hello World!</p>
    <script>
      function myFunction() {
        var text =
        document.getElementById("demo").innerHTML;
        document.getElementById("demo").innerHTML =
        text.toUpperCase();
      }
    </script>
  </body>
</html>
```

Il metodo **toLowerCase()** converte una stringa in minuscolo.

concat()

Il metodo **concat()** unisce due o più stringhe:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript String Methods</h2>
    <p>The concat() method joins two or more strings:</p>
    <p id="demo"></p>
    <script>
      var text1 = "Hello";
      var text2 = "World!";
      var text3 = text1.concat(" ",text2);
      document.getElementById("demo").innerHTML = text3;
    </script>
  </body>
</html>
```

concat() può essere utilizzato al posto dell'operatore più.
Scrivere:

```
var text = "Hello" + " " + "World!";
```

oppure:

```
var text = "Hello".concat(" ", "World!");
```

restituisce il medesimo risultato

trim()

Il metodo **trim()** rimuove gli spazi bianchi da entrambi i lati di una stringa:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>String.trim()</h2>
    <pCliccare sul pulsante per visualizzare la
    stringa con gli spazi vuoti rimossi.</p>
    <button onclick =
    "myFunction()">Cliccami</button>
    <script>
      function myFunction() {
        var str = "     Hello World!     ";
        alert(str.trim());
      }
    </script>
  </body>
</html>
```

padStart() e padEnd()

ECMAScript 2017 ha aggiunto altri due metodi String: **padStart()** e **padEnd()** per supportare il riempimento all'inizio e alla fine di una stringa:

```
<!DOCTYPE html>
<html>
  <body>
  <p>I metodi padStart() e padEnd() riempiono una stringa con altri
  caratteri:</p>
    <p id="demo1"></p>
    <p id="demo2"></p>
    <script>
      let str = "5";
      str = str.padStart(4,0);
      document.getElementById("demo1").innerHTML = str;
      str = str.padEnd(7,"a");
      document.getElementById("demo2").innerHTML = str;
    </script>
  </body>
</html>
```

Il primo parametro indica di quanti caratteri dovrà essere composta la stringa, il secondo fornisce il carattere da inserire. Risultato:

```
Il metodo padStart() riempie una stringa con altri caratteri:
0005
0005aaa
```

charAt()

Il metodo **charAt()** restituisce il carattere , all'interno di una stringa, attraverso un indice (posizione) specificato:

```
<!DOCTYPE html>
<html>
  <body>
    <p>Il metodo charAt() restituisce il carattere
di una
    data posizione in una stringa:</p>
    <p id="demo"></p>
    <script>
      var str = "HELLO WORLD";
      document.getElementById("demo").innerHTML =
      str.charAt(6);
    </script>
  </body>
</html>
```

Risultato:

```
W
```

charCodeAt ()

Il metodo **charCodeAt()** restituisce l'unicode del carattere in corrispondenza di un indice specificato in una stringa:

```
<!DOCTYPE html>
<html>
  <body>
    <p>Il metodo charCodeAt() restituisce l'unicode del
    carattere di una data posizione in una stringa:</p>
    <p id="demo"></p>
    <script>
      var str = "HELLO WORLD";
      document.getElementById("demo").innerHTML =
      str.charCodeAt(2);
    </script>
  </body>
</html>
```

Risultato:

```
76
```

Il metodo restituisce un codice UTF-16 (un numero intero compreso tra 0 e 65535).

split()

Una stringa può essere convertita in un array con il metodo **split()**:

```
<!DOCTYPE html>
<html>
  <body>
    <p>Clicca su "Prova" per visualizzare il primo
    elemento dell'array, dopo una divisione di
    stringa.</p>
    <button onclick="myFunction()">Prova</button>
    <p id="demo"></p>
    <script>
      function myFunction() {
        var str = "a,b,c,d,e,f";
        var arr = str.split(",");
        document.getElementById("demo").innerHTML =
        arr[3];
      }
    </script>
  </body>
</html>
```

Risultato:

d

split()

Se il separatore è "", l'array restituito sarà un array di singoli caratteri:

```
<!DOCTYPE html>
<html>
  <body>
    <p id="demo"></p>
    <script>
      var str = "Hello";
      var arr = str.split("");
      var text = "";
      var i;
      for (i = 0; i < arr.length; i++) {
        text += arr[i] + "<br />"
      }
      document.getElementById("demo").innerHTML = text;
    </script>
  </body>
</html>
```

Risultato:

H
e
l
l
o

NUMERI

Numeri

In JavaScript esiste un solo tipo di numeri. I numeri possono essere scritti con o senza decimali:

```
var x1 = 34.00;
var x2 = 34;
```

Numeri molto grandi o molto piccoli possono essere scritti con notazione scientifica (esponenziale):

```
var y = 123e5;  // 12300000
var z = 123e-5; // 0.00123
```

Numeri

In JavaScript i numeri sono sempre a **virgola mobile a 64 bit**.

A differenza di molti altri linguaggi di programmazione, JavaScript non definisce diversi tipi di numeri, come numeri interi, brevi, lunghi, virgola mobile ecc.

I numeri vengono sempre memorizzati come numeri in virgola mobile a doppia precisione, seguendo lo standard internazionale IEEE 754.
Questo formato memorizza i numeri in 64 bit, dove il numero (la frazione) è memorizzato nei bit da 0 a 51, l'esponente nei bit da 52 a 62 e il segno nel bit 63:

Valore	Esponente	Segno
52 bit (0-51)	11 bit (52-62)	1 bit (63)

Precisione

I numeri interi (numeri senza punto o notazione esponente) sono precisi fino a 15 cifre.

```
var x = 999999999999999; // x sarà 999999999999999
var y = 9999999999999999;// y sarà 10000000000000000
```

Il numero massimo di decimali è 17, ma l'aritmetica in virgola mobile non è sempre precisa al 100%:

```
var x = 0.2 + 0.1; // x sarà 0.30000000000000004
```

Per risolvere il problema sopra descritto, basta moltiplicare e dividere per 10:

```
var x = (0.2 * 10 + 0.1 * 10) / 10; // x sarà 0.3
```

Somma

JavaScript utilizza l'operatore **+** sia per l'addizione che per la concatenazione.
I numeri vengono sommati. Le stringhe sono concatenate.
Sommando due numeri, il risultato sarà un numero:

```
var x = 10;
var y = 20;
var z = x + y; // z sarà 30 (un numero)
```

Sommando due stringhe, il risultato sarà una concatenazione di stringhe:

```
var x = "10";
var y = "20";
var z = x + y; // z sarà 1020 (una stringa)
```

Somma

Sommando un numero e una stringa, il risultato sarà una concatenazione di stringhe:

```
var x = 10;
var y = "20";
var z = x + y; // z sarà 1020 (una stringa)
```

Analogamente:

```
var x = "10";
var y = 20;
var z = x + y; // z sarà 1020 (una stringa)
```

Somma

Scrivendo:

```
var x = 10;
var y = 20;
var z = "Il risultato è: " + x + y;
```

il valore di z sarà:

```
Il risultato è: 1020
```

Scrivendo:

```
var x = 10;
var y = 20;
var z = "30";
var result = x + y + z;
```

Il valore di reslut sarà:

```
3030
```

Stringhe numeriche

Le stringhe possono avere contenuto numerico:

```
var x = 100;    // x è un numero
var y = "100"; // y è una stringa
```

JavaScript prova a convertire le stringhe in numeri in tutte le operazioni numeriche, tranne che nella somma.

Stringhe numeriche

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      var x = "100";
      var y = "10";
      var z = x * y; // z sarà un numero
      document.write(z + "<br />");
      var z = x / y; // z sarà un numero
      document.write(z + "<br />");
      var z = x - y; // z sarà un numero
      document.write(z + "<br />");
      var z = x + y; // z sarà una stringa
      document.write(z);
    </script>
  </body>
</html>
```

Risultato:

```
1000
10
90
10010
```

JavaScript utilizza l'operatore + per concatenare le stringhe.

NaN

NaN è una parola riservata in JavaScript che indica che un numero non è un numero legale.
Il tentativo di eseguire operazioni aritmetiche con una stringa non numerica restituirà NaN (Not a Number):

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Numeri</h2>
    <p>Un numero diviso per una stringa non numerica
    restituisce NaN (Not a Number):</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      100 / "Apple";
    </script>
  </body>
</html>
```

Risultato:

```
NaN
```

isNaN()

Per scoprire se un valore è un numero è possibile utilizzare la funzione globale **isNaN()**:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Numberi</h2>
    <p>È possibile utilizzare la funzione globale
    isNaN() per scoprire se un valore è un numero:</p>
    <p id="demo"></p>
    <script>
      var x = 100 / "Apple";
      document.getElementById("demo").innerHTML =
      isNaN(x);
    </script>
  </body>
</html>
```

Risultato:

```
true
```

NaN

Usando **NaN** in un'operazione matematica, il risultato sarà anch'esso NaN:

```
var x = NaN;
var y = 5;
var z = x + y; // z sarà NaN
```

Oppure il risultato potrebbe essere una concatenazione:

```
var x = NaN;
var y = "5";
var z = x + y; // z sarà NaN5
```

NaN è un numero. **typeof NaN** restituisce number.

Infinity

Infinity (o **-Infinity**) è il valore restituito da JavaScript se si calcola un numero al di fuori del numero più grande possibile.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Numberi</h2>
    <p>Infinity viene restituito se si calcola un numero al
    di fuori del numero più grande possibile:</p>
    <p id="demo"></p>
    <script>
      var myNumber = 2;
      var txt = "";
      while (myNumber != Infinity) {
        myNumber = myNumber * myNumber;
        txt = txt + myNumber + "<br />";
      }
      document.getElementById("demo").innerHTML = txt;
    </script>
  </body>
</html>
```

Infinity

Anche la divisione per 0 (zero) genera **Infinity**:

```
var x =  2 / 0; // x sarà Infinity
var y = -2 / 0; // y sarà -Infinity
```

Infinity è un numero: **typeof Infinity** restituisce number.

Esadecimale

Quando le costanti numeriche sono preceduta da **0x** vengono interpretate come esadecimali.

```
var x = 0xFF; // x sarà 255
```

Per impostazione predefinita, JavaScript visualizza i numeri come decimali in **base 10**.
È possibile usare il metodo **toString()** per produrre numeri dalla **base 2** alla **base 36**.

```
<script>
  var myNumber = 32;
  document.write( "32 = " + "<br />" +
  " Decimale " + myNumber.toString(10) + "<br />" +
  " Esadecimale " + myNumber.toString(16) + "<br />" +
  " Ottale " + myNumber.toString(8) + "<br />" +
  " Binario " + myNumber.toString(2));
</script>
```

Numeri come oggetti

Normalmente i numeri sono valori primitivi creati da letterali:

```
var x = 123;
```

Ma i numeri possono anche essere definiti come oggetti con la parola chiave new:

```
var y = new Number(123);
```

```
var x = 123; // typeof x restituirà number
var y = new Number(123); /* typeof y
restituirà object */
```

Numeri come oggetti

Quando si utilizza l'operatore **==**, i numeri uguali sono uguali:

```
var x = 500;
var y = new Number(500);
/* (x == y) è true perchè x e y anno lo stesso valore */
```

Quando si utilizza l'operatore **===**, i numeri uguali non sono uguali, perché l'operatore **===** si aspetta l'uguaglianza sia nel tipo che nel valore.

```
var x = 500;
var y = new Number(500);
/* (x === y) è false perchè x e y sono di diverso tipo */
```

O anche peggio. Gli oggetti non possono essere confrontati:

```
var x = new Number(500);
var y = new Number(500);
/* (x == y) è false gli oggetti non possono essere confrontati */
```

METODI NUMERICI

Metodi numerici

I metodi numerici ci aiutano a lavorare con i numeri.

I valori primitivi (come 3.14 o 2014), non possono avere proprietà e metodi (perché non sono oggetti).

Ma con JavaScript, metodi e proprietà sono disponibili anche per i valori primitivi, perché JavaScript tratta i valori primitivi come oggetti durante l'esecuzione di metodi e proprietà.

toString()

Il metodo **toString()** converte un numero come stringa.
I metodi numerici possono essere utilizzati su qualsiasi tipo di numero (letterali, variabili o espressioni):

```
var x = 123;
x.toString(); /* restituisce 123 dalla variabile x */
(123).toString(); /* restituisce 123 dal letterale 123 */
(100 + 23).toString(); /* restituisce 123 dall'espressione 100 + 23 */
```

toExponential()

toExponential() restituisce una stringa, con un numero arrotondato e scritto utilizzando la notazione esponenziale. Un parametro definisce il numero di caratteri dopo il punto decimale:

```
var x = 9.656;
x.toExponential(2); // restituisce 9.66e+0
x.toExponential(4); // restituisce 9.6560e+0
x.toExponential(6); // restituisce 9.656000e+0
```

Il parametro è facoltativo. Se non lo specifichi, JavaScript non arrotonderà il numero.

```
x.toExponential(); // restituisce 9.656e+0
```

toFixed()

Il metodo **toFixed()** restituisce una stringa, con un numero di decimali specificato come parametro:

```
var x = 9.656;
x.toFixed(0); // restituisce 10
x.toFixed(2); // restituisce 9.66
x.toFixed(4); // restituisce 9.6560
x.toFixed(6); // restituisce 9.656000
```

toFixed(2) è perfetto per lavorare con il denaro.

toPrecision()

Il metodo **toPrecision()** restituisce una stringa, il parametro specifica il numero delle cifre da considerare:

```
var x = 9.656;
x.toPrecision();  // restituisce 9.656
x.toPrecision(2); // restituisce 9.7
x.toPrecision(4); // restituisce 9.656
x.toPrecision(6); // restituisce 9.65600
```

Metodi globali JavaScript

In JavaScript esistono 3 metodi che possono essere utilizzati per convertire le variabili in numeri:

Metodo	Descrizione
Number()	restituisce un numero, convertito dall'argomento
parseFloat()	analizza l'argomento e restituisce un numero in virgola mobile
parseInt()	analizza l'argomento e restituisce un numero intero

Questi metodi non sono metodi **numerici**, ma metodi JavaScript **globali**.

I metodi globali JavaScript possono essere utilizzati su tutti i tipi di dati JavaScript.

Number()

Il metodo **Number()** può essere utilizzato per convertire le variabili in numeri:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Metodi Globali JavaScript</h2>
    <p>Il metodo Number() converte le variabili in numeri:</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
        Number(true) + "<br />" +
        Number(false) + "<br />" +
        Number("10") + "<br />" +
        Number("  10") + "<br />" +
        Number("10  ") + "<br />" +
        Number(" 10  ") + "<br />" +
        Number("10.33") + "<br />" +
        Number("10,33") + "<br />" +
        Number("10 33") + "<br />" +
        Number("Marco");
    </script>
  </body>
</html>
```

Se il numero non può essere convertito, viene restituito NaN (Not a Number).

Number() utilizzato nelle date

Il metodo **Number()** può anche convertire una data in un numero:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Metodi Globali JavaScript</h2>
    <p>Il metodo Number() converte una data in un
    numero:</p>
    <p id="demo"></p>
    <script>
      var x = new Date("2017-09-30");
      document.getElementById("demo").innerHTML =
      Number(x);
    </script>
  </body>
</html>
```

Risultato:

```
1616371200000
```

Restituisce il numero di millisecondi dall'1.1.1970.

parseInt()

Il metodo **parseInt()** analizza una stringa e restituisce un numero intero. Gli spazi sono consentiti e viene restituito solo il primo numero:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Metodi Globali JavaScript</h2>
    <p>La funzione globale parseInt() converte le stringhe in
    numeri:</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
        parseInt("10") + "<br />" +
        parseInt("10.33") + "<br />" +
        parseInt("10 6") + "<br />" +
        parseInt("10 anni") + "<br />" +
        parseInt("anni 10");
    </script>
  </body>
</html>
```

Se il numero non può essere convertito, viene restituito NaN.

parseFloat()

Il metodo **parseFloat()** analizza una stringa e restituisce un numero in virgola mobile. Gli spazi sono consentiti e viene restituito solo il primo numero:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Metodi Globali JavaScript</h2>
    <p>La funzione globale parseFloat() converte le stringhe
    in numeri:</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
        parseFloat("10") + "<br />" +
        parseFloat("10.33") + "<br />" +
        parseFloat("10 6") + "<br />" +
        parseFloat("10 anni") + "<br />" +
        parseFloat("anni 10");
    </script>
  </body>
</html>
```

Se il numero non può essere convertito, viene restituito NaN.

Proprietà dei numeri

Proprietà	Descrizione
MAX_VALUE	restituisce il numero più grande possibile
MIN_VALUE	restituisce il numero più basso possibile
POSITIVE_INFINITY	rappresenta l'infinito (restituito in caso di overflow)
NEGATIVE_INFINITY	rappresenta l'infinito negativo (restituito in caso di overflow)
NaN	rappresenta un valore "Not-a-Number"

MIN_VALUE e MAX_VALUE

MAX_VALUE restituisce il numero più grande possibile in JavaScript:

```
var x = Number.MAX_VALUE;
//1.7976931348623157e+308
```

MIN_VALUE restituisce il numero più basso possibile in JavaScript:

```
var x = Number.MIN_VALUE;
//5e-324
```

POSITIVE_INFINITY e NEGATIVE_INFINITY

POSITIVE_INFINITY

```
var x = Number.POSITIVE_INFINITY; //Infinity
```

Viene restituito in caso di overflow:

```
var x = 1 / 0; //Infinity
```

NEGATIVE_INFINITY

```
var x = Number.NEGATIVE_INFINITY;//-Infinity
```

Viene restituito in caso di overflow:

```
var x = -1 / 0; //-Infinity
```

NaN

NaN è una parola riservata a JavaScript la quale indica che un numero non è un numero legale.
Il tentativo di eseguire operazioni aritmetiche con una stringa non numerica risulterà NaN (Not a Number):

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Numeri</h2>
    <p>Un numero diviso per una stringa non numerica
    restituirà NaN (Not a Number):</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      100 / "Patate";
    </script>
  </body>
</html>
```

Risultato:

```
NaN
```

Proprietà dei numeri

Le proprietà Number appartengono al wrapper dell'oggetto numero di JavaScript denominato **Number** .
È possibile accedere a queste proprietà solo tramite Number.MAX_VALUE.
L'utilizzo di *myNumber*.MAX_VALUE, dove *myNumber* è una variabile, un'espressione o un valore, restituirà undefined:

```
var x = 6;
document.write( x.MAX_VALUE);
//restituisce undefined
```

ARRAY

Creazione di un array

L'utilizzo di un array letterale è il modo più semplice per creare un array JavaScript.
Sintassi:

```
var array_name = [item1, item2, ...];
var cars = ["Saab", "Volvo", "BMW"];
```

Gli spazi e le interruzioni di riga non sono importanti. Una dichiarazione può estendersi su più righe:

```
var cars = [
  "Saab",
  "Volvo",
  "BMW"
];
```

new

È possibile creare un array utilizzando la parola chiave **new**:

```
var cars = new Array("Saab", "Volvo", "BMW");
```

Per semplicità, leggibilità e velocità di esecuzione, è meglio evitare di utilizzare new Array e preferire il metodo letterale.

Accesso agli elementi

Si accede ad un elemento dell'array facendo riferimento al **numero di indice** .
Questa istruzione accede al valore del primo elemento dell'array cars:

```
var name = cars[0];
```

Gli indici degli array iniziano con 0.
[0] è il primo elemento; [1] è il secondo elemento; ecc.

Modifica di un elemento

Per cambiare il valore del primo elemento in cars basta scrivere:

```
cars[0] = "Opel";
```

Accesso all'array completo

In JavaScript, è possibile accedere all'intero array facendo riferimento al nome dell'array:

```
var cars = ["Saab", "Volvo", "BMW"];
document.write ( cars );
```

Risultato:

```
Saab,Volvo,BMW
```

typeof

Gli array sono un tipo speciale di oggetti. L'operatore typeof restituisce "oggetto" per gli array.
Tuttavia, gli array JavaScript sono meglio descritti come array.
Gli array utilizzano **numeri** per accedere ai suoi "elementi".
In questo esempio, person[1] restituisce Polo:

```
var person = ["Marco", "Polo", 46];
```

Gli oggetti utilizzano **nomi** per accedere ai propri "membri".
In questo esempio, person.firstName restituisce Marco:

```
var person = {firstName:"Marco",
lastName:"Polo", age:46}
```

Variabili come oggetti

Le variabili in JavaScript possono essere oggetti. Gli array sono tipi speciali di oggetti.
Per questo motivo è possibile avere variabili di tipi diversi nello stesso array:

- È possibile avere oggetti in un array.
- È possibile avere funzioni in un array.
- È possibile avere array in un array.

```
myArray[0] = Date.now;
myArray[1] = myFunction;
myArray[2] = myCars;
```

La proprietà length

Il vero punto di forza degli array JavaScript sono le proprietà e i metodi degli array incorporati.
La proprietà **lenght** restituisce la lunghezza di un array (il numero di elementi dell'array):

```
var frutti = ["Banana", "Arancia", "Mela"];
frutti.length; // la lunghezza di frutta è 3
```

La proprietà lenght restituisce sempre l'indice della matrice più alto aumentato di 1.

Accesso all'ultimo elemento dell'array

Sfruttando la proprietà lenght è possibile accedere immediatamente all'ultimo elemento dell'array:

```
frutti = ["Banana", "Arancia", "Mela"];
var last = frutti[frutti.length - 1];
```

Ciclo for

Il modo più sicuro per scorrere un array è usare un ciclo for:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Array</h2>
    <p>Il modo migliore per scorrere un array è utilizzare un
    ciclo for:</p>
    <p id="demo"></p>
    <script>
      var frutti, text, fLen, i;
      frutti = ["Banana", "Arancia", "Mela", "Mango"];
      fLen = frutti.length;
      text = "<ul>";
      for (i = 0; i < fLen; i++) {
        text += "<li>" + frutti[i] + "</li>";
      }
      text += "</ul>";
      document.getElementById("demo").innerHTML = text;
    </script>
  </body>
</html>
```

forEach()

Per scorrere un array è possibile utilizzare la funzione **forEach()**:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Array</h2>
    <p>Array.forEach() chiama una funzione per ogni elemento
    dell'array.</p>
    <p id="demo"></p>
    <script>
      var frutti, text;
      frutti = ["Banana", "Arancia", "Mela", "Mango"];
      text = "<ol>";
      frutti.forEach(myFunction);
      text += "</ol>";
      document.getElementById("demo").innerHTML = text;
      function myFunction(elemento) {
        text += "<li>" + elemento + "</li>";
      }
    </script>
  </body>
</html>
```

Aggiungere un nuovo elemento

Il metodo **push()** è modo più semplice per aggiungere un nuovo elemento a un array:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Array</h2>
    <p>Il metodo push aggiunge un nuovo elemento a un
    array.</p>
    <button onclick="myFunction()">Prova</button>
    <p id="demo"></p>
    <script>
      var frutti = ["Banana", "Arancia", "Mela", "Mango"];
      document.getElementById("demo").innerHTML = frutti;
      function myFunction() {
        frutti.push("Pera");
        document.getElementById("demo").innerHTML = frutti;
      }
    </script>
  </body>
</html>
```

Aggiungere un nuovo elemento

È inoltre possibile aggiungere un nuovo elemento a un array utilizzando la proprietà lenght:

```
var frutti = ["Banana", "Arancia", "Mela"];
frutti[frutti.length] = "Pera";
//aggiunge un nuovo elemento (Pera) a frutti
```

Aggiungere un nuovo elemento

L'aggiunta di elementi con indici alti può creare "buchi" non definiti in un array:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Array</h2>
    <p>L'aggiunta di elementi con indici alti può creare
    "buchi" indefiniti in un array.</p>
    <p id="demo"></p>
    <script>
      var frutti, text, fLen, i;
      frutti = ["Banana", "Arancia", "mela", "Mango"];
      frutti[6] = "Pera";
      fLen = frutti.length;
      text = "";
      for (i = 0; i < fLen; i++) {
        text += frutti[i] + "<br />";
      }
      document.getElementById("demo").innerHTML = text;
    </script>
  </body>
</html>
```

Array associativi

Molti linguaggi di programmazione supportano array con indici denominati.
Gli array con indici denominati sono chiamati array associativi (o hash).
JavaScript **non** supporta array con indici denominati.
In JavaScript, gli array utilizzano sempre **indici numerici**.

```
var persona = [];
persona[0] = "Marco";
persona[1] = "Giuseppe";
persona[2] = 46;
var x = persona.length; /* persona.length
restituirà 3 */
var y = persona[0]; /* persona[0] restituirà
"Marco" */
```

Array associativi

Se si utilizzano indici denominati, JavaScript ridefinirà l'array in un oggetto standard.
Questo comporterà che alcuni metodi e proprietà degli array produrranno **risultati errati**.

```
var persona = [];
persona["nome1"] = "Marco";
persona["nome2"] = "Giuseppe";
persona["anni"] = 46;
var x = persona.length; /* persona.length
restituirà 0 */
var y = persona[0]; /* persona[0] restituirà
undefined*/
```

Differenza tra array e oggetti

In JavaScript, gli **array** utilizzano **indici numerati**.
In JavaScript, gli **oggetti** utilizzano **indici denominati**.
Gli array sono un tipo speciale di oggetti, con indici numerati.

JavaScript non supporta gli array associativi.

- Bisogna usare gli **oggetti** quando vogliamo che i nomi degli elementi siano **stringhe (testo)**.
- Bisogna usare gli **array** quando vogliamo che i nomi degli elementi siano **numeri**.

Riconoscere una matrice

Come facciamo a sapere se una variabile è un array?

Il problema è che l'operatore JavaScript typeof restituisce "object":

```
var frutti = ["Banana", "Arancia", "Mela"];
typeof frutti; // restituisce object
```

L'operatore typeof restituisce object perché, come già detto, un array JavaScript è un oggetto.

Riconoscere una matrice

ECMAScript 5 definisce un nuovo metodo **Array.isArray()**:

```
Array.isArray(frutti); // restituisce true
```

Il problema con questa soluzione è che ECMAScript 5 non è supportato nei browser meno recenti.

Per risolvere questo problema è possibile creare una propria funzione **isArray()**:

```
function isArray(x) {
  return x.constructor.toString().indexOf("Array") > -
1;
}
```

La funzione sopra restituisce sempre true se l'argomento è un array.

O più precisamente: restituisce true se il prototipo dell'oggetto contiene la parola "Array".

Riconoscere una matrice

Il metodo migliore per capire se una variabile è un array è utilizzando l'operatore **instanceof.** Restituisce true se un oggetto viene creato da un determinato costruttore:

```
var frutti = ["Banana", "Arancia", "Mela"];
frutti instanceof Array; // resituisce true
```

METODI DELLE MATRICI

Conversione di array in stringhe

Il metodo **toString()** converte un array in una stringa formata dai valori dell'array (separati da virgole).

```
var frutti = ["Banana", "Arancia", "Mela"];
document.write ( frutti.toString() );
```

Risultato:

```
Banana,Arancia,Mela
```

Conversione di array in stringhe

Anche il metodo **join()** unisce tutti gli elementi dell'array in una stringa.

Si comporta esattamente come **toString()**, ma in aggiunta è possibile specificare il separatore:

```
var frutti = ["Banana", "Arancia", "Mela"];
document.write ( frutti.join(" - ") );
```

Risultato:

```
Banana - Arancia - Mela
```

Rimuovere elementi

Il metodo **pop()** rimuove l'ultimo elemento da un array:

```
var frutti = ["Banana", "Arancia", "Mela"];
frutti.pop(); /* rimuove l'ultimo elemento
("Mela") da frutti */
```

Il metodo **pop()** restituisce il valore che viene rimosso:

```
var frutti = ["Banana", "Arancia", "Mela"];
var x = frutti.pop(); /* il valore di x è
"Mela" */
```

Aggiungere elementi

Il metodo **push()** aggiunge un nuovo elemento a un array (alla fine):

```
var frutti = ["Banana", "Arancia", "Mela"];
frutti.push("Pera"); /* aggiunge un elemento
("Pera") a frutti */
```

Il metodo **push()** restituisce la nuova lunghezza dell'array:

```
var frutti = ["Banana", "Arancia", "Mela"];
var x = frutti.push("Pera"); /* il valore di
x è 4 */
```

shift()

Shifting equivale a popping, lavorando sul primo elemento invece che sull'ultimo.
Il metodo **shift()** rimuove il primo elemento della matrice e "sposta" tutti gli altri elementi su un indice inferiore:

```
var frutti = ["Banana", "Arancia", "Mela"];
frutti.shift(); /* rimuove il primo elemento ("Banana") da frutti */
```

Il metodo **shift()** restituisce la stringa che è stata rimossa:

```
var frutti = ["Banana", "Arancia", "Mela"];
var x = frutti.shift(); /* il valore di x è ("Banana") */
```

unshift()

Il metodo **unshift()** aggiunge un nuovo elemento a un array (all'inizio):

```
var frutti = ["Banana", "Arancia", "Mela"];
frutti.unshift("Pera"); /* aggiunge
l'elemento "Pera" a frutti */
```

Il metodo **unshift()** restituisce la nuova lunghezza dell'array:

```
var frutti = ["Banana", "Arancia", "Mela"];
var x = frutti.unshift("Pera"); /*
restituisce 4 */
```

Eliminare elementi

Poiché gli array JavaScript sono oggetti, gli elementi possono essere eliminati utilizzando l'operatore **delete**:

```
<script>
  var frutti = ["Banana", "Arancia", "Mela"];
  document.write (
  "Il 1° frutto è: " + frutti[0] + "<br />" );
  delete frutti[0];
  document.write (
  "Il 1° frutto è: " + frutti[0] );
</script>
```

Risultato:

```
Il 1° frutto è: Banana
Il 1° frutto è: undefined
```

L'uso di **delete** può lasciare buchi indefiniti nell'array. È meglio utilizzare pop() o shift().

splice()

Il metodo **splice()** può essere utilizzato per aggiungere nuovi elementi a un array:

```
var frutti = ["Banana", "Arancia", "Mela"];
frutti.splice(2, 0,"Pera", "Uva");
```

Il primo parametro (2) definisce la posizione in cui devono essere **inseriti** i nuovi elementi (giuntati).

Il secondo parametro (0) definisce quanti elementi devono essere **rimossi**.

Il resto dei parametri ("Pera", "Uva") definiscono i nuovi elementi da **aggiungere**.

splice()

```
<!DOCTYPE html>
<html>
  <body>
    <h2>splice()</h2>
    <p>Il metodo splice() aggiunge nuovi elementi all'array e, qualora
ci fossero, restituisce un array con gli elementi eliminati.</p>
    <button onclick="myFunction()">Prova</button>
    <p id="demo1"></p>
    <p id="demo2"></p>
    <p id="demo3"></p>
    <script>
      var frutti = ["Banana", "Arancia", "Mela", "Mango"];
      document.getElementById("demo1").innerHTML =
      "Array originale:<br /> " + frutti;
      function myFunction() {
        var removed = frutti.splice(2, 2, "Pera", "Uva");
        document.getElementById("demo2").innerHTML =
        "Nuovo Array:<br />" + frutti;
        document.getElementById("demo3").innerHTML =
        "Elementi rimosso:<br /> " + removed;
      }
    </script>
  </body>
</html>
```

splice()

Combinando adeguatamente i parametri, è possibile utilizzare **splice()** per rimuovere elementi senza lasciare "buchi" nell'array:

```
var frutti = ["Banana", "Arancia", "Mela"];
frutti.splice(0, 1); /* rimuove il primo elemento di frutti */
```

Dato che il primo parametro (0) definisce la posizione in cui devono essere aggiunti nuovi elementi, che il secondo parametro (1) definisce quanti elementi devono essere rimossi e che gli altri parametri vengono omessi, non verranno aggiunti nuovi elementi.

concat()

Il metodo **concat()** crea un nuovo array unendo (concatenando) gli array esistenti:

```
var myGirls = ["Cecilie", "Lone"];
var myBoys = ["Emil", "Tobias", "Linus"];
var myChildren = myGirls.concat(myBoys);
// Concatena(unisce) myGirls e myBoys
```

Il metodo **concat()** non modifica gli array esistenti. Restituisce sempre un nuovo array.

concat()

Il metodo **concat()** accetta un numero qualsiasi di argomenti di matrice:

```
var arr1 = ["Cecilie", "Lone"];
var arr2 = ["Emil", "Tobias", "Linus"];
var arr3 = ["Robin", "Morgan"];
var myChildren = arr1.concat(arr2, arr3);
// concatena arr1 con arr2 e arr3
```

Il metodo **concat()** può anche accettare stringhe come argomenti:

```
var arr1 = ["Emil", "Tobias", "Linus"];
var myChildren = arr1.concat("Peter");
```

slice()

Il metodo **slice()** suddivide una parte di un array in un nuovo array.
Questo esempio taglia una parte di un array a partire dall'elemento 1 ("Arancia"):

```
var frutti =
["Banana", "Arancia", "Limone", "Mela"];
var frutti2 = frutti.slice(1);
```

Il metodo **slice()** crea un nuovo array. Non rimuove alcun elemento dall'array di origine.

slice()

Il metodo **slice()** accetta due argomenti.
Il metodo seleziona quindi gli elementi dall'argomento iniziale (incluso) fino all'argomento finale (escluso).

```
var frutti =
["Banana", "Arancia", "Limone", "Mela",
"Pera"];
var agrumi = frutti.slice(1, 3);
```

sort()

Il metodo **sort()** ordina alfabeticamente un array:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Ordinamento array</h2>
    <p>Il metodo sort() ordina un array in ordine
    alfabetico.</p>
    <button onclick="myFunction()">Prova</button>
    <p id="demo"></p>
    <script>
      var frutti = ["Banana", "Arancia", "Mela", "Mango"];
      document.getElementById("demo").innerHTML = frutti;
      function myFunction() {
        frutti.sort();
        document.getElementById("demo").innerHTML = frutti;
      }
    </script>
  </body>
</html>
```

reverse()

Il metodo **reverse()** inverte gli elementi in un array.
Si può utilizzare per ordinare un array in ordine decrescente:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Ordine inverso degli array</h2>
    <p>Il metodo reverse () inverte gli elementi in un array.</p>
    <p>Combinando sort() e reverse() ordiniamo un array in ordine decrescente.
    </p>
    <button onclick="myFunction()">Prova</button>
    <p id="demo"></p>
      <script>
        // Crea e visualizza un array:
        var frutti = ["Banana", "Arancia", "Mela", "Mango"];
        document.getElementById("demo").innerHTML = frutti;
        function myFunction() {
          // Ordina l'array
          frutti.sort();
          // Inverte l'ordine degli elementi:
          frutti.reverse();
          document.getElementById("demo").innerHTML = frutti;
        }
    </script>
  </body>
</html>
```

Ordinamento numerico

Per impostazione predefinita, la funzione **sort()** ordina i valori come **stringhe** .
Funziona bene per le stringhe ("Arancia" viene prima di "Banana").
Tuttavia, se i numeri sono ordinati come stringhe, "25" è maggiore di "100", perché "2" è maggiore di "1".
Per questo motivo, il metodo **sort()** produrrà risultati errati durante l'ordinamento dei numeri.

Ordinamento numerico

Possiamo effettuare un ordinamento numerico usando una **funzione di confronto**:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Ordinamento array</h2>
    <p>Clicca sul pulsante per ordinare la matrice in ordine
    crescente.</p>
    <button onclick="myFunction()">Prova</button>
    <p id="demo"></p>
    <script>
      var punti = [40, 100, 1, 5, 25, 10];
      document.getElementById("demo").innerHTML = punti;
      function myFunction() {
        punti.sort(
          function(a, b){return a - b}
        );
        document.getElementById("demo").innerHTML = punti;
      }
    </script>
  </body>
</html>
```

Ordinamento numerico

Allo stesso modo possiamo effettuare un ordinamento decrescente:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Ordinamento array</h2>
    <p>Clicca sul pulsante per ordinare la matrice in ordine
    decrescente.</p>
    <button onclick="myFunction()">Prova</button>
    <p id="demo"></p>
    <script>
      var punti = [40, 100, 1, 5, 25, 10];
      document.getElementById("demo").innerHTML = punti;
      function myFunction() {
        punti.sort(
          function(a, b){return b - a}
        );
        document.getElementById("demo").innerHTML = punti;
      }
    </script>
  </body>
</html>
```

Funzione di confronto

Lo scopo della funzione di confronto è definire un **ordinamento alternativo**.
La funzione di confronto dovrebbe restituire un valore negativo, zero o positivo, a seconda degli argomenti:

```
function(a, b){return a - b}
```

Quando la funzione **sort()** confronta due valori, invia i valori alla funzione di confronto e ordina i valori in base al valore restituito (negativo, zero, positivo):

- Se il risultato è negativo a viene ordinato prima b.
- Se il risultato è positivo b viene ordinato prima a.
- Se il risultato è 0, non vengono apportate modifiche all'ordinamento dei due valori.

Funzione di confronto

Nell'esempio precedente la **funzione di confronto** confronta tutti i valori nell'array, due valori alla volta (a, b).

Quando si confrontano 40 e 100, il metodo **sort()** chiama la **funzione di confronto(40, 100)**.

La funzione calcola 40-100 (a - b) e poiché il risultato è negativo (-60), la funzione di ordinamento ordinerà 40 prima di 100.

Esempio

Questo frammento di codice effettua l'ordinamento numerico o alfabetico:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Ordinamento array</h2>
    <p>Cliccare sui pulsanti per ordinare la matrice alfabeticamente o
    numericamente.</p>
    <button onclick="myFunction1()">Ordinamento alfabetico</button>
    <button onclick="myFunction2()">Ordinamento numerico</button>
    <p id="demo"></p>
    <script>
      var punti = [40, 100, 1, 5, 25, 10];
      document.getElementById("demo").innerHTML = punti;
      function myFunction1() {
        punti.sort();
        document.getElementById("demo").innerHTML = punti;
      }
      function myFunction2() {
        punti.sort(function(a, b){return a - b});
        document.getElementById("demo").innerHTML = punti;
      }
    </script>
  </body>
</html>
```

Ordine casuale

Il metodo per effettuare un ordine casuale in un array si chiama Fisher Yates, ed è stato introdotto nella scienza dei dati già nel 1938!
In JavaScript il metodo può essere tradotto in questo:

```
var punti = [40, 100, 1, 5, 25, 10];
for (i = punti.length -1; i > 0; i--) {
  j = Math.floor(Math.random() * i)
  k = punti[i]
  punti[i] = punti[j]
  punti[j] = k
}
```

Esempio

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Il metodo Fisher Yates</h3>
    <p>Clicca il pulsante per ordinare l'array in ordine casuale.</p>
    <button onclick="myFunction()">Prova</button>
    <p id="demo"></p>
    <script>
      var punti = [40, 100, 1, 5, 25, 10];
      document.getElementById("demo").innerHTML = punti;
      function myFunction() {
        var i, j, k;
        for (i = punti.length -1; i > 0; i--) {
          j = Math.floor(Math.random() * i)
          k = punti[i]
          punti[i] = punti[j]
          punti[j] = k
        }
        document.getElementById("demo").innerHTML = punti;
      }
    </script>
  </body>
</html>
```

Massimo e minimo

Non ci sono funzioni incorporate per trovare il valore massimo o minimo in un array.
Tuttavia, dopo aver ordinato un array, è possibile utilizzare l'indice per ottenere i valori più alti e più bassi.
Ordinamento crescente:

```
var punti = [40, 100, 1, 5, 25, 10];
punti.sort(function(a, b){return a - b});
// punti[0] contiene il valore minimo
/* punti[punti.length-1] contiene il valore massimo */
```

Ordinare un intero array è un metodo molto inefficiente se si desidera trovare solo il valore più alto (o più basso).

Math.max.apply

È possibile usare **Math.max.apply** per trovare il numero più alto in un array:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Valore massimo</h2>
    <p>Il numero più alto è:
    <spanid="demo"></span>.</p>
    <script>
      var punti = [40, 100, 1, 5, 25, 10];
      document.getElementById("demo").innerHTML =
      myArrayMax(punti);
        function myArrayMax(arr) {
        return Math.max.apply(null, arr);
      }
    </script>
  </body>
</html>
```

Math.min.apply

Allo stesso modo è possibile usare **Math.min.apply** per trovare il numero più basso in un array:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Valore minimo</h2>
    <p>Il numero più basso è:
    <span id="demo"></span>.</p>
    <script>
      var punti = [40, 100, 1, 5, 25, 10];
      document.getElementById("demo").innerHTML =
      myArrayMin(punti);
        function myArrayMin(arr) {
        return Math.min.apply(null, arr);
      }
    </script>
  </body>
</html>
```

Trova massimo

La soluzione più veloce per trovare il valore massimo e minimo in un array è utilizzando un metodo "fatto in casa". Creiamo una funzione che scorre un array confrontando ogni valore con il valore più alto trovato:

```
function trovaMassimo(arr) {
  var lunghezza = arr.length;
  var massimo = -Infinity;
  while (lunghezza--) {
    if (arr[lunghezza] > massimo) {
      massimo = arr[lunghezza];
    }
  }
return massimo;
}
```

Trova minimo

Allo stesso modo creiamo una funzione che scorre un array confrontando ogni valore con il valore più basso trovato:

```
function trovaMinimo(arr) {
  var lunghezza = arr.length;
  var minimo = Infinity;
  while (lunghezza--) {
    if (arr[lunghezza] < minimo) {
    minimo = arr[lunghezza];
    }
  }
return minimo;
}
```

Trova massimo e trova minimo

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Valore massimo e valore minimo</h2>
    <p>Il valore minimo è: <span id="demo1"></span>.</p>
    <p>Il valore massimo è: <span id="demo2"></span>.</p>
    <script>
      var punti = [40, 100, 1, 5, 25, 10];
      document.getElementById("demo1").innerHTML =
      trovaMinimo(punti);
      function trovaMinimo(arr) {
        var lunghezza = arr.length;
        var minimo = Infinity;
        while (lunghezza--) {
          if (arr[lunghezza] < minimo) {
          minimo = arr[lunghezza];
          }
        }
      return minimo;
      }
      document.getElementById("demo2").innerHTML =
      trovaMassimo(punti);
      function trovaMassimo(arr) {
        var lunghezza = arr.length;
        var massimo = -Infinity;
        while (lunghezza--) {
          if (arr[lunghezza] > massimo) {
            massimo = arr[lunghezza];
          }
        }
      return massimo;
      }
    </script>
  </body>
</html>
```

Ordinare di oggetti

Spesso gli array JavaScript contengono oggetti:

```
var cars = [
  {tipo:"Volvo", anno:2016},
  {tipo:"Saab", anno:2001},
  {tipo:"BMW", anno:2010}
];
```

Anche se gli oggetti hanno proprietà con diversi tipi di dati, il metodo **sort()** può essere utilizzato per ordinare l'array. La soluzione è scrivere una funzione di confronto per confrontare i valori delle proprietà:

```
cars.sort(function(a, b){return a.year - b.year});
```

Ordinare gli oggetti

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Ordinamento Array di oggetti</h2>
    <p>Clicca il pulsante per ordinare gli oggetti in base all'anno.</p>
    <button onclick="myFunction()">Ordina</button>
    <p id="demo"></p>
    <script>
      var cars = [
        {tipo:"Volvo", anno:2016},
        {tipo:"Saab", anno:2001},
        {tipo:"BMW", anno:2010}
      ];
      displayCars();
      function myFunction() {
        cars.sort(function(a, b){return a.anno - b.anno});
        displayCars();
      }
      function displayCars() {
        document.getElementById("demo").innerHTML =
        cars[0].tipo + " " + cars[0].anno + "<br />" +
        cars[1].tipo + " " + cars[1].anno + "<br />" +
        cars[2].tipo + " " + cars[2].anno;
      }
    </script>
  </body>
</html>
```

Ordinare gli oggetti

Il confronto delle proprietà delle stringhe è più complesso:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Ordinamento Array di oggetti</h2>
    <p>Cliccare il pulsante per ordinare gli oggetti in base al tipo.</p>
    <button onclick="myFunction()">Ordina</button>
    <p id="demo"></p>
    <script>
      var cars = [
        {tipo:"Volvo", anno:2016},
        {tipo:"Saab", anno:2001},
        {tipo:"BMW", anno:2010}
      ];
      displayCars();
      function myFunction() {
        cars.sort(function(a, b){
          var x = a.tipo.toLowerCase();
          var y = b.tipo.toLowerCase();
          if (x < y) {return -1;}
          if (x > y) {return 1;}
          return 0;
        });
        displayCars();
      }
      function displayCars() {
        document.getElementById("demo").innerHTML =
        cars[0].tipo + " " + cars[0].anno + "<br />" +
        cars[1].tipo + " " + cars[1].anno + "<br />" +
        cars[2].tipo + " " + cars[2].anno;
      }
    </script>
  </body>
</html>
```

METODI DI ITERAZIONE

Array.forEach()

Il metodo **forEach()** chiama una funzione (una funzione di callback) una volta per ogni elemento dell'array.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Array.forEach()</h2>
    <p>Chiama una funzione una volta per ogni elemento
    dell'array.</p>
    <p id="demo"></p>
    <script>
      var txt = "";
      var numbers = [45, 4, 9, 16, 25];
      numbers.forEach(myFunction);
      document.getElementById("demo").innerHTML = txt;
      function myFunction(value, index, array) {
        txt = txt + value + "<br />";
      }
    </script>
  </body>
</html>
```

Nota che la funzione accetta 3 argomenti:

- Il valore dell'elemento
- L'indice degli elementi
- La matrice stessa

Array.forEach()

L'esempio precedente utilizza solo il parametro value.
Possiamo quindi riscriverlo come segue:

```
var txt = "";
var numbers = [45, 4, 9, 16, 25];
numbers.forEach(myFunction);

function myFunction(value) {
  txt = txt + value + "<br />";
}
```

Array.map()

Il metodo **map()** crea un nuovo array (non modifica l'array originale) eseguendo una funzione su ogni elemento dell'array.

Il metodo **map()** non esegue la funzione per gli elementi della matrice senza valori.

La funzione accetta 3 argomenti:

- Il valore dell'elemento
- L'indice degli elemento
- La matrice stessa

Array.map()

Questo esempio moltiplica ogni valore della matrice per 2:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Array.map()</h2>
    <p>Crea un nuovo array eseguendo una funzione su ogni elemento
    dell'array.</p>
    <p>La matrice di partenza è: <span id="demo1"></span>.</p>
    <p>Matrice con valori doppi: <span id="demo2"></span>.</p>
    <script>
      var numbers1 = [45, 4, 9, 16, 25];
      document.getElementById("demo1").innerHTML = numbers1;
      var numbers2 = numbers1.map(myFunction);
      document.getElementById("demo2").innerHTML = numbers2;
      function myFunction(value, index, array) {
        return value * 2;
      }
    </script>
  </body>
</html>
```

Quando una funzione di callback utilizza solo il parametro value, i parametri index e array possono essere omessi.

Array.filter()

Il metodo **filter()** crea un nuovo array con gli elementi dell'array che superano un dato valore.
Questo esempio crea un nuovo array formato dagli elementi che hanno un valore maggiore di 18:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Array.filter()</h2>
    <p>Crea un nuovo array con tutti gli elementi che
    superano un test.</p>
    <p id="demo"></p>
    <script>
      var numbers = [45, 4, 9, 16, 25];
      var over18 = numbers.filter(myFunction);
      document.getElementById("demo").innerHTML = over18;
      function myFunction(value, index, array) {
        return value > 18;
      }
    </script>
  </body>
</html>
```

Anche in questo caso index ed array possono essere omessi.

Array.reduce()

Il metodo **reduce()** esegue una funzione su ogni elemento dell'array per produrre (ridurlo a) un singolo valore, non modificando l'array originale.
Il metodo **reduce()** scorre l'array da sinistra a destra.
Questo esempio trova la somma di tutti i numeri in un array:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Array.reduce()</h2>
    <p>Questo esempio effettua la somma di tutti i valori di
    un array:</p>
    <p id="demo"></p>
    <script>
      var numbers = [45, 4, 9, 16, 25];
      var somma = numbers.reduce(myFunction);
      document.getElementById("demo").innerHTML =
      "La somma è " + somma;
      function myFunction(total, value, index, array) {
        return total + value;
      }
    </script>
  </body>
</html>
```

Array.reduce()

La funzione array.reduce() accetta 4 argomenti:

- Il totale
- Il valore dell'elemento
- L'indice degli elementi
- La matrice stessa

L'esempio precedente non utilizza l'indice e i parametri dell'array. Possiamo riscriverlo:

```
var numbers1 = [45, 4, 9, 16, 25];
var somma = numbers1.reduce(myFunction);

function myFunction(total, value) {
  return total + value;
}
```

Array.reduce()

Il metodo **reduce()** può accettare un valore iniziale che verrà sommato al totale:

```
var numbers1 = [45, 4, 9, 16, 25];
var somma =
numbers1.reduce(myFunction, 100);

function myFunction(total, value) {
  return total + value;
}
```

Risultato:

```
La somma è 199
```

Array.reduceRight()

Il metodo **reduceRight()** esegue una funzione su ogni elemento dell'array per produrre (ridurlo a) un singolo valore, non modificando l'array originale.
A differenza di reduce() scorre la matrice da destra a sinistra.
Anche la funzione array.reduceRight() accetta 4 argomenti:

- Il totale
- Il valore dell'elemento
- L'indice degli elementi
- La matrice stessa

Come per il metodo **reduce()**, può accettare un valore iniziale che verrà sommato al totale.

Array.every()

Il metodo **every()** controlla se tutti i valori dell'array superano un test.
Questo esempio controlla se tutti i valori dell'array sono maggiori di 18:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Array.every()</h2>
    <p>Il metodo every() controlla se tutti i valori
    dell'array superano un test.</p>
    <p id="demo"></p>
    <script>
      var numbers = [45, 4, 9, 16, 25];
      var allOver18 = numbers.every(myFunction);
      document.getElementById("demo").innerHTML =
      "Tutti maggiori di 18 è " + allOver18;
      function myFunction(value, index, array) {
        return value > 18;
      }
    </script>
  </body>
</html>
```

Array.every()

La funzione accetta 3 argomenti:

- Il valore dell'elemento
- L'indice degli elementi
- La matrice stessa

Quando una funzione di callback utilizza solo il primo parametro (valore), gli altri parametri possono essere omessi:

```
var numbers = [45, 4, 9, 16, 25];
var allOver18 = numbers.every(myFunction);

function myFunction(value) {
  return value > 18;
}
```

Array.some()

Il metodo **some()** controlla se alcuni valori dell'array superano un test.
Questo esempio controlla se alcuni valori dell'array sono maggiori di 18:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Array.some()</h2>
    <p>Il metodo some() controlla se alcuni valori
    dell'array superano un test.</p>
    <p id="demo"></p>
    <script>
      var numbers = [45, 4, 9, 16, 25];
      var someOver18 = numbers.some(myFunction);
      document.getElementById("demo").innerHTML =
      "Alcuni valori maggiori di 18 è " + someOver18;
      function myFunction(value, index, array) {
        return value > 18;
      }
    </script>
  </body>
</html>
```

Anche in questo caso La funzione accetta 3 argomenti:

- Il valore dell'elemento
- L'indice degli elementi
- La matrice stessa

Array.indexOf()

Il metodo **indexOf()** cerca un elemento per valore e ne restituisce la posizione.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Array.indexOf()</h2>
    <p id="d"></p>
    <script>
      var frutti =
      ["Mela", "Arancia", "Mela", "Mango"];
      var a = frutti.indexOf("Mela");
      document.getElementById("d").innerHTML =
      "Mela si trova nella posizione " + a;
    </script>
  </body>
</html>
```

Array.indexOf()

Il metodo **indexOf()** accetta due parametri:
`array.indexOf(item, start)`

- item: l'elemento da cercare (necessario).
- start: da dove iniziare la ricerca (opzionale). I valori negativi inizieranno dalla posizione data cercheranno fino alla fine dell'array.

Array.indexOf() restituisce -1 se l'elemento non viene trovato.
Se l'elemento è presente più di una volta, restituisce la posizione della prima occorrenza.

Array.lastIndexOf ()

Array.lastIndexOf() è uguale ad Array.indexOf(), ma restituisce la posizione dell'ultima occorrenza dell'elemento specificato.

Anche il metodo **lastIndexOf()** accetta due parametri:

```
array.lastIndexOf(item, start)
```

Array.find()

Il metodo **find()** restituisce il valore del primo elemento dell'array che supera una funzione di test.
Questo esempio trova il primo elemento maggiore di 18:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Array.find()</h2>
    <p id="demo"></p>
    <script>
      var numbers = [4, 9, 16, 25, 29];
      var first = numbers.find(myFunction);
      document.getElementById("demo").innerHTML =
      "Il primo valore maggiore di 18 è " + first;
      function myFunction(value, index, array) {
        return value > 18;
      }
    </script>
  </body>
</html>
```

Array.find()

La funzione accetta 3 argomenti:

- Il valore dell'elemento
- L'indice degli elementi
- La matrice stessa

Array.findIndex()

Il metodo **findIndex()** restituisce l'indice del primo elemento dell'array che supera una funzione di test.
Questo esempio trova l'indice del primo elemento maggiore di 18:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Array.findIndex()</h2>
    <p id="demo"></p>
    <script>
      var numbers = [4, 9, 16, 25, 29];
      var first = numbers.findIndex(myFunction);
      document.getElementById("demo").innerHTML =
      "Il primo valore maggiore di 18 ha indice " +
      first;
      function myFunction(value, index, array) {
        return value > 18;
      }
    </script>
  </body>
</html>
```

Array.findIndex()

La funzione accetta 3 argomenti:

- Il valore dell'elemento
- L'indice degli elementi
- La matrice stessa

DATA

Output data

Per impostazione predefinita, JavaScript utilizzerà il fuso orario del browser e visualizzerà una data sotto forma di stringa di testo:

```
Tue Apr 06 2021 08:19:24 GMT + 0200 (Ora legale dell'Europa centrale)
```

new Date()

Gli oggetti data vengono creati con il costruttore **new Date()**.
Esistono 4 modi per creare un nuovo oggetto data:

```
new Date()
new Date(year, month, day, hours, minutes, seconds, milliseconds)
new Date(milliseconds)
new Date(date string)
```

new Date() crea un nuovo oggetto data con la data e l'ora correnti:

```
var d = new Date();
```

new Date(*year, month, ...*)

new Date(*year, month, ...*) crea un nuovo oggetto data con una data e un'ora specificate.

7 parametri specificano anno, mese, giorno, ora, minuti, secondi e millisecondi (in quest'ordine):

```
var d =
new Date(2021, 03, 12, 08, 02, 30, 10);
```

Risultato:

```
Mon Apr 12 2021 08:02:30 GMT+0200 (Ora
legale dell’Europa centrale)
```

JavaScript conta i mesi da 0 a 11

new Date(*year, month, ...*)

6 parametri specificano anno, mese, giorno, ora, minuti e secondi:

```
var d = new Date(2021, 03, 12, 08, 02, 30);
```

Risultato:

```
Mon Apr 12 2021 08:02:30 GMT+0200 (Ora legale dell'Europa centrale)
```

5 parametri specificano anno, mese, giorno, ora e minuti:

```
var d = new Date(2021, 03, 12, 08, 02);
```

Risultato:

```
Mon Apr 12 2021 08:02:00 GMT+0200 (Ora legale dell'Europa centrale)
```

Ecc…

new Date(*year, month, ...*)

Non è possibile omettere il mese. Fornendo un solo parametro, verrà trattato come millisecondi.

```
var d = new Date(2021);
```

Risultato:

```
Thu Jan 01 1970 01:00:02 GMT+0200 (Ora standard dell'Europa centrale)
```

JavaScript memorizza le date come millisecondi partebndo dal 1 gennaio 1970, 00:00:00 UTC (Universal Time Coordinated).

L'ora zero è il 01 gennaio 1970 00:00:00 UTC.

new Date(*year, month, ...*)

Gli anni a una e due cifre vengono interpretati come appartenenti al secolo scorso:

```
var d = new Date(99, 11, 24);
```

Risultato:

```
Fri Dec 24 1999 00:00:00 GMT+0200 (Ora
standard dell'Europa centrale)
```

```
var d = new Date(9, 11, 24);
```

Risultato:

```
Fri Dec 24 1909 00:00:00 GMT+0200 (Ora
standard dell'Europa centrale)
```

new Date(dateString)

new Date(dateString) crea un nuovo oggetto data da una stringa:

```
var d =
new Date("October 13, 2014 11:13:00");
```

Risultato:

```
Mon Oct 13 2014 11:13:00 GMT+0200 (Ora
legale dell'Europa centrale)
```

new Date(*milliseconds*)

new Date(*milliseconds*) crea un nuovo oggetto data formato dal tempo di base più i millisecondi forniti come parametro.
Il 1 ° gennaio 1970 più 10000000000 di millisecondi corrisponde approssimativamente al 03 marzo 1973:

```
var d = new Date(100000000000);
```

Risultato:

```
Sat Mar 03 1973 10:46:40 GMT+0200 (Ora
standard dell'Europa centrale)
```

new Date(*milliseconds*)

Il 1 ° gennaio 1970 meno 10000000000 di millisecondi corrisponde approssimativamente al 31 ottobre 1966:

```
var d = new Date(-10000000000);
```

Risultato:

```
Mon Oct 31 1966 15:13:20 GMT+0200 (Ora standard dell'Europa centrale)
```

Un giorno corrisponde a 86400000 millisecondi:

```
var d = new Date(-10000000000);
```

Risultato:

```
Fri Jan 02 1970 01:00:00 GMT+0200 (Ora standard dell'Europa centrale)
```

Visualizzazione della data

Per impostazione predefinita JavaScript produrrà (come abbiamo visto) le date in formato stringa di testo completo:

```
Tue Apr 06 2021 09:01:01 GMT+0200 (Ora legale dell'Europa centrale)
```

Quando si visualizza un oggetto data in HTML, viene automaticamente convertito in una stringa, con il metodo toString().

```
d = new Date();
document.getElementById("demo").innerHTML = d;
```

Corrisponde a:

```
d = new Date();
document.getElementById("demo").innerHTML =
d.toString();
```

toUTCString()

Il metodo **toUTCString()** converte una data in una stringa UTC (uno standard di visualizzazione della data).

```
var d = new Date();
document.write( d.toUTCString() );
```

Risultato:

```
Tue, 06 Apr 2021 09:05:14 GMT
```

toDateString()

Il metodo **toDateString()** converte una data in un formato più leggibile:

```
var d = new Date();
document.write( d.toDateString() );
```

Risultato:

```
Tue Apr 06 2021
```

toISOString()

Il metodo **toISOString()** converte una una stringa, utilizzando il formato standard ISO:

```
var d = new Date();
document.write( d.toISOString() );
```

Risultato:

```
2021-04-06T09:10:30.745Z
```

Formati di input

Esistono generalmente 3 formati di input della data JavaScript:

Formato	Esempio
Data ISO	"2015-03-25" (Standard internazionale)
Data breve	"03/25/2015"
Data lunga	"Mar 25 2015" o "25 Mar 2015"

In JavaScript il formato ISO segue uno standard rigoroso. Gli altri formati non sono così ben definiti e potrebbero essere specifici del browser.

Standard ISO 8601

ISO 8601 è lo standard internazionale per la rappresentazione di date e orari.
La sintassi ISO 8601 (AAAA-MM-GG) è anche il formato della data preferito da JavaScript:

```
var d = new Date("2015-03-25");
document.write( d );
```

Risultato:

```
Wed Mar 25 2015 01:00:00 GMT+0200 (Ora standard dell’Europa centrale)
```

Data ISO

Le date ISO possono essere scritte senza specificare il giorno (AAAA-MM):

```
var d = new Date("2015-03");
```

Risultato

```
Sun Mar 01 2015 01:00:00 GMT+0200 (Ora standard dell'Europa centrale)
```

Possono anche essere scritte senza mese e giorno (AAAA):

```
var d = new Date("2015");
```

Risultato

```
Thu Jan 01 2015 01:00:00 GMT+0200 (Ora standard dell'Europa centrale)
```

Data ISO

Le date ISO possono essere scritte aggiungendo ore, minuti e secondi (AAAA-MM-GGTHH: MM: SSZ):

```
var d = new Date("2015-03-25T12:00:00Z");
```

Risultato:

```
Wed Mar 25 2015 13:00:00 GMT+0200 (Ora standard dell’Europa centrale)
```

La data e l'ora sono separate dalla T maiuscola.
L'ora UTC è definita con una lettera maiuscola Z.

Se si vuole modificare l'ora relativa all'UTC, si deve rimuovere la Z e aggiungere +HH:MM o -HH:MM

```
document.write( new Date("2015-03-25T12:00:00-02:00") );
```

Risultato:

```
Wed Mar 25 2015 15:00:00 GMT+0200 (Ora standard dell’Europa centrale)
```

Fusi orari

UTC (Universal Time Coordinated) è lo stesso di GMT (Greenwich Mean Time).

L'omissione di T o Z in una stringa data-ora può fornire risultati diversi in browser diversi.

Quando si imposta una data, senza specificare il fuso orario, JavaScript utilizzerà il fuso orario del browser.

Quando si ottiene una data, senza specificare il fuso orario, il risultato viene convertito nel fuso orario del browser.

In altre parole: se una data/ora viene creata in GMT (Greenwich Mean Time), la data/ora verrà convertita in CDT (Central US Daylight Time) se un utente naviga dagli Stati Uniti centrali.

Data breve

Le date brevi sono scritte con la sintassi "MM/GG/AAAA":

```
var d = new Date("03/25/2015");
document.write( d );
```

Risultato:

```
Wed Mar 25 2015 00:00:00 GMT+0200 (Ora standard dell'Europa centrale)
```

In alcuni browser, mesi o giorni senza zeri iniziali possono produrre un errore.
Il comportamento di "AAAA/MM/GG" e "GG/MM/AAAA" non è definito. Alcuni browser tenteranno di indovinare il formato, altri restituiranno NaN.

Data lunga

Le date lunghe sono spesso scritte con la sintassi "MMM GG AAAA":

```
var d = new Date("Mar 25 2015");
document.write( d );
```

Risultato:

```
Wed Mar 25 2015 00:00:00 GMT+0200 (Ora standard dell'Europa centrale)
```

Il mese e il giorno possono essere scritti in qualsiasi ordine, il mese può essere scritto per intero (March), o abbreviato (Mar):

```
var d = new Date("25 March 2015");
document.write( d );
```

Le virgole vengono ignorate ed nomi non fanno distinzione tra maiuscole e minuscole:

```
var d = new Date("MARCH, 25, 2015");
```

Date.parse()

Se si dispone di una stringa data valida, è possibile utilizzare il metodo **Date.parse()** per convertirla in millisecondi.

Date.parse() restituisce il numero di millisecondi tra il 1 gennaio 1970 e la data fornita:

```
var msec = Date.parse("March 21, 2012");
document.write( msec );
```

Risultato:

```
1332284400000
```

Date.parse()

È possibile utilizzare il numero di millisecondi per convertirlo in un oggetto data:

```
var msec = Date.parse(new Date());
var d = new Date(msec);
document.write( d );
```

Risultato:

```
Tue Apr 06 2021 09:53:23 GMT+0200 (Ora
legale dell’Europa centrale)
```

Metodi di acquisizione della data

Metodo	Descrizione
getFullYear()	Calcola l'anno come numero a quattro cifre (aaaa)
getMonth()	Calcola il mese come numero (0-11)
getDate()	Calcola il giorno come numero (1-31)
getHours()	Calcola l'ora (0-23)
getMinutes()	Calcola i minuti (0-59)
getSeconds()	Calcola i secondi (0-59)
getMilliseconds()	Calcola i millisecondi (0-999)
getTime()	Calcola l'ora (millisecondi dal 1 gennaio 1970)
getDay()	Calcola il giorno della settimana come numero (0-6)

getTime()

Il metodo **getTime()** restituisce il numero di millisecondi dal 1 gennaio 1970:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript getTime()</h2>
    <p>In JavaScript l'orologio interno conta dalla
    mezzanotte dell'1 gennaio 1970.</p>
    <p>La funzione getTime() restituisce il numero
di
    millisecondi da allora:</p>
    <p id="demo"></p>
    <script>
      var d = new Date();
      document.getElementById("demo").innerHTML =
      d.getTime();
    </script>
  </body>
</html>
```

getFullYear() - getMonth()

Il metodo **getFullYear()** restituisce l'anno di una data come numero di quattro cifre:

```
var d = new Date();
document.write( d.getFullYear() );
```

Risultato:

```
2021
```

Il metodo **getMonth()** restituisce il mese di una data come numero (0-11):

```
var d = new Date();
document.write( d.getMonth() + 1 );
```

Risultato:

```
4
```

getMonth()

Come già detto, il primo mese (gennaio) è il numero 0, quindi dicembre restituisce il numero 11.
È possibile usare una matrice di nomi e getMonth() per restituire il mese come nome:

```
var d = new Date();
var mesi =
["Gennaio","Febbraio","Marzo","Aprile",
"Maggio","Giugno","Luglio","Agosto",
"Settembre","Ottobre","Novembre","Dicembre"];
document.write( mesi[d.getMonth()] );
```

Risultato:

```
Aprile
```

getDate()

Il metodo **getDate()** restituisce il giorno di una data come numero (1-31):

```
var d = new Date();
document.write( d.getDate() );
```

Risultato:

```
21
```

getHours() – getMinutes() – getSeconds()

Il metodi getHours(), getMinutes() e getSeconds() restituiscono le ore(0-23), i minuti (0-59) ed i secondi (0-59 di una data come numero:

```
var d = new Date();
document.write( d.getHours() + ":");
document.write( d.getMinutes() + ":");
document.write( d.getSeconds() );
```

Risultato:

```
10:13:43
```

getMilliseconds()

Il metodo **getMilliseconds()** restituisce i millisecondi di una data come numero (0-999):

```
var d = new Date();
document.write( d.getMilliseconds() );
```

Risultato:

```
2134
```

getDay()

Il metodo **getDay()** restituisce il giorno della settimana di una data come numero (0-6).
È possibile usare una matrice di nomi e getDay() per restituire il giorno come nome:

```
var d = new Date();
var giorni =
["Domenica","Lunedì","Martedì","Mercoledì",
"Giovedì","Venerdì","Sabato"];
document.write( giorni[d.getDay()] );
```

Risultato:

```
Lunedì
```

UTC

I metodi di data UTC vengono utilizzati per lavorare con le date UTC (date del fuso orario universale):

Metodo	Descrizione
getUTCDate()	Restituisce la data in UTC
getUTCDay()	Restituisce il giorno in UTC
getUTCFullYear()	Restituisce l'anno in UTC
getUTCHours()	Restituisce l'ora in UTC
getUTCMilliseconds()	Restituisce i millisecondi in UTC
getUTCMinutes()	Restituisce i minuti in UTC
getUTCMonth()	Restituisce il giorno in UTC
getUTCSeconds()	Restituisce i secondi in UTC

Metodi di impostazione della data

I metodi Set Date consentono di impostare i valori della data per un oggetto Date. Vengono utilizzati per impostare una data:

Metodo	Description
setDate()	Imposta il giorno (1-31)
setFullYear()	Imposta l'anno (opzionalmente mese e giorno)
setHours()	Imposta l'ora (0-23)
setMilliseconds()	Imposta i millisecondi (0-999)
setMinutes()	Imposta i minuti (0-59)
setMonth()	Imposta il mese (0-11)
setSeconds()	Imposta i secondi (0-59)
setTime()	Imposta i millisecondi (dal 1° Gennaio 1970)

setFullYear()

Il metodo **setFullYear()** imposta l'anno di un oggetto data:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript setFullYear()</h2>
    <p>Il metodo setFullYear() imposta
    l'anno di un oggetto data:</p>
    <script>
      var d = new Date();
      d.setFullYear(1975);
      document.write( d );
    </script>
  </body>
</html>
```

setFullYear()

Il metodo **setFullYear()** può **opzionalmente** impostare mese e giorno:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript setFullYear()</h2>
    <p>Il metodo setFullYear() può
    opzionalmente impostare mese e giorno.</p>
    <p>Notare che i mesi partono da 0. Decembre
    corrisponde ad 11:</p>
    <script>
      var d = new Date();
      d.setFullYear(2001, 8, 24);
      document.write( d );
    </script>
  </body>
</html>
```

setMonth()

Il metodo **setMonth()** imposta il mese di un oggetto data (0-11):

```
var d = new Date();
d.setMonth(11);
document.write( d );
```

setDate()

Il metodo **setDate()** imposta il giorno di un oggetto data (1-31):

```
var d = new Date();
d.setDate(15);
document.write( d );
```

Il metodo **setDate()** può essere utilizzato anche per aggiungreregiorni a una data:

```
var d = new Date();
d.setDate(d.getDate() + 60);
document.write( d );
```

setHours() – setMinutes() – setSeconds()

I metodi setHours(), setMinutes() e setSeconds() impostano le ore (0-23), i minuti (0-59) e i secondi (0-59) di un oggetto data:

```
var d = new Date();
d.setHours(22);
d.setMinutes(22);
d.setSeconds(22);
document.write( d );
```

Confrontare le date

Le date possono essere facilmente confrontate.
L'esempio seguente confronta la data odierna con il 14 gennaio 2100:

```
var oggi, unGiorno, text;
oggi = new Date();
unGiorno = new Date();
unGiorno.setFullYear(2100, 0, 14);
if (unGiorno > oggi) {
  text = "Oggi è successivo al 14 Gennaio 2100.";
}
else {
  text = "Oggi è precedente al 14 Gennaio 2100.";
}
document.write( text );
```

Risultato:

```
Oggi è successivo al 14 Gennaio 2100.
```

MATH

Math

L'oggetto **Math** consente di eseguire attività matematiche sui numeri.

```
Math.PI; // restituisce 3.141592653589793
```

A differenza di altri oggetti, Math non ha un costruttore.
L'oggetto Math è statico.
Tutti i metodi e le proprietà possono essere utilizzati senza creare prima un oggetto Math.

Costanti

La sintassi per qualsiasi proprietà matematica è:
Math.*property*
JavaScript fornisce 8 costanti matematiche a cui è possibile accedere come proprietà matematiche:

```
Math.E //restituisce il numero di Eulero
Math.PI //restituisce Pi Greco
Math.SQRT2 //restituisce la radice quadrata di 2
Math.SQRT1_2 //restituisce la radice quadrata di 1/2
Math.LN2 //restituisce il logaritmo naturale di 2
Math.LN10 //restituisce il logaritmo naturale di 10
Math.LOG2E //restituisce il logaritmo in base 2 di E
Math.LOG10E //restituisce il logaritmo in base 10 di E
```

Metodi matematici

La sintassi per qualsiasi metodo matematico è:
Math.*method*.(*number*)

Esistono 4 metodi comuni per arrotondare un numero ad un intero:

Metodo	Descrizione
Math.round(x)	Restituisce x arrotondato al numero intero più vicino
Math.ceil(x)	Restituisce x arrotondato al numero intero più vicino
Math.floor(x)	Restituisce x arrotondato per difetto al numero intero più vicino
Math.trunc(x)	Restituisce la parte intera di x

Math.round()

Math.round(x) restituisce il numero intero più vicino:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.round()</h2>
    <p>Math.round(x) restituisce il valore di x
    arrotondato al numero intero più vicino: </p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      Math.round(4.4); //restiutisce 5
    </script>
  </body>
</html>
```

Math.round()

Math.ceil(x) restituisce il valore di x arrotondato **fino** al suo intero più vicino:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.ceil()</h2>
    <p>Math.ceil(x) restituisce il valore di x
    arrotondato al primo numero intero più grande:
    </p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      Math.ceil(4.4); //restiutisce 5
    </script>
  </body>
</html>
```

Math.floor()

Math.floor(x) restituisce il valore di x arrotondato per difetto al numero intero più vicino:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.floor()</h2>
    <p>Math.floor(x) restituisce il valore di x
    arrotondato per difetto:</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      Math.floor(4.9); //restiutisce 4
    </script>
  </body>
</html>
```

Math.trunc()

Math.trunc(x) restituisce la parte intera di x:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.trunc()</h2>
    <p>Math.trunc(x) restituisce la parte
    intera di x:</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      Math.trinc(-4.9); //restiutisce -4
    </script>
  </body>
</html>
```

Math.sign()

Math.trunc(x) restituisce 1 se x è positivo, 0 se x è nullo e -1 se x è negativo:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.sign()</h2>
    <p>Math.sign(x) restituisce se x è positivo,
    nullo o negativo:</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      Math.sign(-4); //restiutisce -1
    </script>
  </body>
</html>
```

Math.pow()

Math.pow(x, y) restituisce il valore di x elevato a y:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.pow()</h2>
    <p>Math.pow(x,y) restituisce il valore di x
    elevato ad y:</p>
    <p id="demo"></p>
    <script>
     document.getElementById("demo").innerHTML =
     Math.pow(8,2); //restiutisce 64
    </script>
  </body>
</html>
```

Math.sqrt()

Math.sqrt(x) restituisce la radice quadrata di x:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.sqert()</h2>
    <p>Math.sqrt(x,) restituisce la radice quadrata
    di x:</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      Math.sqrt(64); //restiutisce 8
    </script>
  </body>
</html>
```

Math.abs()

Math.abs(x) restituisce il valore assoluto di x:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.abs()</h2>
    <p>Math.abs(x,) restituisce il valore assoluto
    di x:</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      Math.abs(-4,4); //restiutisce 4,4
    </script>
  </body>
</html>
```

Math.sin()

Math.sin(x) restituisce il seno dell'angolo x (espresso in radianti):

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.sin()</h2>
    <p>Math.sin(x,) restituisce il seno di x
    (espresso in radianti). Per utilizzare i gradi
    basta moltiplicare per Pi greco e dividere per
    180:</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      "Il valore del seno di 90 gradi è" +
      Math.sin(90 * Math.PI/180); //restiutisce 1
    </script>
  </body>
</html>
```

Math.cos()

Math.cos(x) restituisce il coseno dell'angolo x (espresso in radianti):

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.cos()</h2>
    <p>Math.cos(x,) restituisce il coseno di x
    (espresso in radianti). Per utilizzare i gradi
    basta moltiplicare per Pi greco e dividere per
    180:</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      "Il valore del coseno di 180 gradi è" +
      Math.sin(180 * Math.PI/180); //restiutisce -1
    </script>
  </body>
</html>
```

Math.min() e Math.max()

Math.min() e **Math.max()** possono essere utilizzati per trovare il valore minimo o massimo in un elenco di argomenti:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.min() e Math.max()</h2>
    <p>Math.min() restituisce il valore minimo in una lista
    di argomenti:</p>
    <p id="demo1"></p>
    <hr />
    <p>Math.max() restituisce il valore massimo in una lista
    di argomenti:</p>
    <p id="demo2"></p>
    <script>
      document.getElementById("demo1").innerHTML =
      Math.min(0, 150, 30, 20, -8, -200); //restituisce -200
      document.getElementById("demo2").innerHTML =
      Math.max(0, 150, 30, 20, -8, -200); //restituisce 150
    </script>
  </body>
</html>
```

Math.random()

Math.random() restituisce un numero casuale compreso tra 0 (incluso) e 1 (escluso):

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.random()</h2>
    <p>Math.random() restituisce un numero casuale
    compreso tra 0 e 1:</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      Math.random();
    </script>
  </body>
</html>
```

Math.log(x)

Math.log(x) restituisce il logaritmo naturale di x:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.log()</h2>
    <p>Math.log() restituisce il logaritmo
    naturale di un numero:</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      Math.log(1); //restituisce 0
    </script>
  </body>
</html>
```

Math.log2(x)

Math.log2(x) restituisce il logaritmo in base 2 di x:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.log2()</h2>
    <p>Math.log2() restituisce il logaritmo in
    base 2 di un numero.</p>
    <p>Quante volte dobbiamo moltiplicare 2 per
    ottenere 8?</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      Math.log2(8); //restituisce 3
    </script>
  </body>
</html>
```

Math.log10(x)

Math.log10(x) restituisce il logaritmo in base 10 di x:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.log10()</h2>
    <p>Math.log10() restituisce il logaritmo in
    base 10 di un numero.</p>
    <p>Quante volte dobbiamo moltiplicare 10 per
    ottenere 10000?</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      Math.log10(10000); //restituisce 4
    </script>
  </body>
</html>
```

Metodi degli oggetti matematici

Metodo	Descrizione
abs(x)	Restituisce il valore assoluto di x
acos(x)	Restituisce l'arcocoseno di x, in radianti
acosh(x)	Restituisce l'arcocoseno iperbolico di x
asin(x)	Restituisce l'arcoseno di x, in radianti
asinh(x)	Restituisce l'arcoseno iperbolico di x
atan(x)	Restituisce l'arcotangente di x come valore numerico compreso tra -PI / 2 e PI / 2 radianti
atan2(y, x)	Restituisce l'arcotangente del quoziente dei suoi argomenti
atanh(x)	Restituisce l'arcotangente iperbolico di x
cbrt(x)	Restituisce la radice cubica di x
ceil(x)	Restituisce x, arrotondato per eccesso all'intero più vicino

Metodi degli oggetti matematici

Metodo	Descrizione
cos(x)	Restituisce il coseno di x (x è in radianti)
cosh(x)	Restituisce il coseno iperbolico di x
exp(x)	Restituisce il valore di e^x
floor(x)	Restituisce x, arrotondato per difetto al numero intero più vicino
log(x)	Restituisce il logaritmo naturale (base E) di x
max(x, y, z, ..., n)	Restituisce il numero con il valore più alto
min(x, y, z, ..., n)	Restituisce il numero con il valore più basso
pow(x, y)	Restituisce il valore di x alla potenza di y
random()	Restituisce un numero casuale compreso tra 0 e 1
round(x)	Arrotonda x al numero intero più vicino

Metodi degli oggetti matematici

Metodo	Descrizione
sign(x)	Restituisce se x è negativo, nullo o positivo (-1, 0, 1)
sin(x)	Restituisce il seno di x (x è in radianti)
sinh(x)	Restituisce il seno iperbolico di x
sqrt(x)	Restituisce la radice quadrata di x
tan(x)	Restituisce la tangente di un angolo
tanh(x)	Restituisce la tangente iperbolica di un numero
trunc(x)	Restituisce la parte intera di un numero

Math.random()

Come già visto **Math.random()** restituisce un numero casuale compreso tra 0 (incluso) e 1 (escluso), quindi restituisce sempre un numero inferiore a 1.

Combinato con **Math.floor()** può essere utilizzato per restituire numeri interi casuali.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.random()</h2>
    <p>Math.floor(Math.random() * 10) restituisce un
    numero intero casuale compreso tra 0 e 9
    (inclusi):</p>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
      Math.floor(Math.random() * 10);
    </script>
  </body>
</html>
```

Math.random()

```
Math.floor(Math.random() * 11);
/* restituisce un numero intero casuale compreso tra
0 e 10 */

Math.floor(Math.random() * 100);
/* restituisce un numero intero casuale compreso tra
0 e 99 */

Math.floor(Math.random() * 101);
/* restituisce un numero intero casuale compreso tra
0 e 100 */

Math.floor(Math.random() * 10) + 1;
/* restituisce un numero intero casuale compreso tra
1 e 10 */

Math.floor(Math.random() * 100) + 1;
/* restituisce un numero intero casuale compreso tra
1 e 100 */
```

Funzione per generare numeri casuali

Come abbiamo visto dagli esempi precedenti, potrebbe essere una buona idea creare una funzione appropriata da utilizzare per ottenere dei numeri interi casuali compresi tra min (incluso) e max (escluso):

```
<!DOCTYPE html>
<html>
  <body>
    <h2>JavaScript Math.random()</h2>
    <p>Ogni volta che clicchi sul pulsante, la funzione
    getRndInteger(min, max) restituisce un numero casuale
    compreso tra 0 e 9 (entrambi inclusi):</p>
    <button
      onclick="document.getElementById('demo').innerHTML =
      getRndInteger(0,10)">Cliccami</button>
    <p id="demo"></p>
    <script>
      function getRndInteger(min, max) {
        return Math.floor(Math.random() * (max - min)) + min;
      }
    </script>
  </body>
</html>
```

Per includere tra i numeri generati anche max basta far restituire alla funzione il valore di `Math.floor(Math.random() * (max – min +1 )) + min;`

BOOLEANI

Boolean()

Possiamo usare la funzione **Boolean()** per scoprire se un'espressione (o una variabile) è vera:

```
<!DOCTYPE html>
<html>
  <body>
    <p>Visualizza il valore di Boolean(10 > 9):</p>
    <button onclick="myFunction()">Cliccami</button>
    <p id="demo"></p>
    <script>
      function myFunction() {
        document.getElementById("demo").innerHTML =
        Boolean(10 > 9); //restituisce true
      }
    </script>
  </body>
</html>
```

Avremo lo stesso risultato anche scrivendo:

```
document.getElementById("demo").innerHTML = 10 > 9;
```

Boolean()

Tutto ciò che ha un "valore" è vero:

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      var b1 = Boolean(100);
      var b2 = Boolean(3.14);
      var b3 = Boolean(-15);
      var b4 = Boolean("Hello");
      var b5 = Boolean('false');
      var b6 = Boolean(1 + 7 + 3.14);
      document.write(
        "100 è " + b1 + "<br />" +
        "3.14 è " + b2 + "<br />" +
        "-15 è " + b3 + "<br />" +
        "Qualsiasi stringa (non vuota) è " + b4 + "<br />" +
        "Anche la stringa 'false' è " + b5 + "<br />" +
        "Qualsiasi espressione (tranne zero) è " + b6 );
    </script>
  </body>
</html>
```

Boolean()

Tutto ciò che privo di un "valore" è falso:

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      var b1 = Boolean(0);
      var b2 = Boolean(-0);
      var b3 = Boolean("");
      var b4 = Boolean(b4);
      var b5 = Boolean(null);
      var b6 = Boolean(false);
      var b7 = Boolean(NaN);
      document.write(
        "0 è" + b1 + "<br />" +
        "-0 è " + b2 + "<br />" +
        "Una stringa vuota è " + b3 + "<br />" +
        "Un valore non definito è " + b4 + "<br />" +
        "null è " + b5 + "<br />" +
        "false è " + b6 + "<br />" +
        "NaN è " +b7);
    </script>
  </body>
</html>
```

Booleani come oggetti

Normalmente i booleani sono valori primitivi creati da letterali:

```
var x = false;
```

Ma i booleani possono anche essere definiti come oggetti con la parola chiave **new**:

```
var y = new Boolean(false);
```

Booleani come oggetti

```
<!DOCTYPE html>
<html>
  <body>
    <p>Non creare mai booleani come oggetti.</p>
    <p>Booleani e oggetti non possono essere
    confrontati in modo sicuro.</p>
    <script>
      var x = false;                  // x è un booleano
      var y = new Boolean(false); // y è un oggetto
      document.write("x è di tipo " +  typeof x +
      "<br />" + "y è di tipo " + typeof y );
    </script>
  </body>
</html>
```

Creare oggetti booleani rallenta la velocità di esecuzione.
La parola chiave new complica il codice.

Booleani come oggetti

Creare oggetti booleani può produrre alcuni risultati inaspettati. Quando si utilizza l'operatore ==, i valori booleani uguali sono uguali:

```
var x = false;
var y = new Boolean(false);
/* (x == y) è true perché x e y hanno lo stesso valore */
```

Quando si utilizza l' operatore ===, i valori booleani uguali non sono uguali, perché l' operatore === si aspetta l'uguaglianza sia nel tipo che nel valore.

```
var x = false;
var y = new Boolean(false);
/* (x == y) è false perché x e y sono di tipo differente */
```

OPERATORI DI CONFRONTO

Operatori di confronto

Gli operatori di confronto vengono utilizzati nelle istruzioni logiche per determinare l'uguaglianza o la differenza tra variabili o valori.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Confronti JavaScript</h2>
    <p>Assegna 5 a x e visualizza il valore del
    confronto:</p>
    <script>
      var x = 5;
      document.write( "(x == 8) " + (x == 8) + "<br />" );
      document.write( "(x == 5) " + (x == 5) + "<br />" );
      document.write( "(x == '5') " + (x == "5") );
    </script>
  </body>
</html>
```

Operatori di confronto

Gli operatori di confronto vengono utilizzati nelle istruzioni logiche per determinare l'uguaglianza o la differenza tra variabili o valori.
Assegnamo 5 alla variabile x e visualizziamo il valore dei vari confronti:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Confronti JavaScript</h2>
    <p>Assegnato 5 ad x visualizziamo il valore del confronto:</p>
    <script>
      var x = 5;
      document.write( "(x == 8) " + (x == 8) + "<br />" );
      document.write( "(x == 5) " + (x == 5) + "<br />" );
      document.write( "(x == '5') " + (x == "5") + "<br />" );
      document.write( "(x === 5) " + (x === 5) + "<br />" );
      document.write( "(x === '5') " + (x === "5") + "<br />" );
      document.write( "(x != 8) " + (x != 8) + "<br />" );
      document.write( "(x !== 5) " + (x !== 5) + "<br />" );
      document.write( "(x !== '5') " + (x !== "5") + "<br />" );
      document.write( "(x !== 8) " + (x !== 8) + "<br />" );
      document.write( "(x x > 8) " + (x > 8) + "<br />" );
      document.write( "(x x < 8) " + (x < 8) + "<br />" );
      document.write( "(x >= 8) " + (x >= 8) + "<br />" );
      document.write( "(x <= 8) " + (x <= 8) + "<br />" );
    </script>
  </body>
</html>
```

Operatori di confronto

Gli operatori di confronto possono essere utilizzati nelle istruzioni condizionali per confrontare i valori ed agire in base al risultato:

```
if (anni < 18) text = "Troppo giovane per comprare alcolici";
```

Operatori logici

Gli operatori logici vengono utilizzati per determinare la logica tra variabili o valori.
Assegniamo a x = 6 e y = 3, il seguente codice spiega gli operatori logici:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Confronti JavaScript</h2>
    <p><hr />L'operatore AND (&&) restituisce true se entrambe le espressioni sono vere,
    altrimenti restituisce false.</p>
    <p id="demoAND"><p>
    <p>L'operatore OR (||) restituisce true se una o entrambe le espressioni sono vere,
    altrimenti restituisce false.</p>
    <p id="demoOR"><p>
    <p>L'operatore NOT (!) restituisce true per affermazioni false e false per affermazioni
    vere.</p>
    <p id="demoNOT"><p>
    <script>
      var x = 6;
      var y = 3;
      document.getElementById("demoAND").innerHTML =
        "x<10 e y>1 " + (x < 10 && y > 1) + "<br />" +
        "x<10 e y<1 " + (x < 10 && y < 1) + "<hr />";
      document.getElementById("demoOR").innerHTML =
        "x=5 o y=5 " + (x == 5 || y == 5) + "<br />" +
        "x=6 o y=0 " + (x == 6 || y == 0) + "<br />" +
        "x=0 o y=3 " + (x == 0 || y == 3) + "<br />" +
        "x=6 o y=3 " + (x == 6 || y == 3) + "<hr />";
      document.getElementById("demoNOT").innerHTML =
        "x diverso valore e diverso tipo di y " +  !(x === y) + "<br />" +
        "x non maggiore di y " + !(x > y);
    </script>
  </body>
</html>
```

Operatore condizionale

JavaScript contiene anche un operatore condizionale che assegna un valore a una variabile in base a una condizione.
nomeVariabile = (*condizione*) ? *valore1*:*valore2*

```
<!DOCTYPE html>
<html>
  <body>
    <p>Inserisci la tua età e clicca sul pulsante:</p>
    <input id="anni" value="18" />
    <button onclick="myFunction()">Prova</button>
    <p id="demo"></p>
    <script>
      function myFunction() {
        var anni, possibile;
        anni = document.getElementById("anni").value;
        possibile = (anni < 18) ?
        "Troppo giovane":"Abbastanza grande";
        document.getElementById("demo").innerHTML =
        possibile + " per votare.";
      }
    </script>
  </body>
</html>
```

Se la variabile età è un valore inferiore a 18, il valore della variabile possibile sarà "Troppo giovane", altrimenti il valore di possibile sarà "Abbastanza grande".

Confronto di diversi tipi

Il confronto di dati di diversi tipi può dare risultati imprevisti.
Quando si confronta una stringa con un numero, JavaScript convertirà la stringa in un numero durante il confronto. Una stringa vuota viene convertita in 0.

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Confronti JavaScript</h2>
    <p>2 &#60; 12 <span id="a1"></span></p>
    <p>2 &#60; "12" <span id="a2"></span></p>
    <p>2 &#60; "John" <span id="a3"></span></p>
    <p>2 &#62; "John" <span id="a4"></span></p>
    <p>2 == "John" <span id="a5"></span></p>
    <p>"2" &#60; "12" <span id="a6"></span></p>
    <p>"2" == "12" <span id="a7"></span></p>
    <script>
      document.getElementById("a1").innerHTML =  2 < 12;
      document.getElementById("a2").innerHTML =  2 < "12";
      document.getElementById("a3").innerHTML = 2 < "John";
      document.getElementById("a4").innerHTML = 2 > "John";
      document.getElementById("a5").innerHTML = 2 == "John";
      document.getElementById("a6").innerHTML = "2" < "12";
      document.getElementById("a7").innerHTML = "2" == "12";
    </script>
  </body>
</html>
```

Confronto di stringhe

Quando si confrontano due stringhe, "2" sarà maggiore di "12", perché (in ordine alfabetico) 1 (one) è minore di 2 (two).
Per garantire un risultato corretto, le variabili devono essere convertite nel tipo corretto prima del confronto:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>Confronto su JavaScript</h2>
    <p>Inserisci la tua età e clicca su verifica:</p>
    <input id="anni" value="18" />
    <button onclick="myFunction()">Verifica</button>
    <p id="demo"></p>
    <script>
      function myFunction() {
        var anni, possibile;
        anni = Number(document.getElementById("anni").value);
        if (isNaN(anni)) {
          possibile = "Il valore inserito non è un numero";
        }
        else {
          possibile = (anni < 18) ? "Troppo giovane":"Abbastanza grande";
        }
        document.getElementById("demo").innerHTML = possibile;
      }
    </script>
  </body>
</html>
```

ISTRUZIONI CONDIZIONALI

Molto spesso, quando scriviamo un codice, desideriamo eseguire azioni diverse per condizioni diverse.
Per eseguire questa operazione nel codice, è possibile utilizzare le istruzioni condizionali.
In JavaScript abbiamo le seguenti istruzioni condizionali:

- **if** per specificare un blocco di codice da eseguire se una condizione specificata è vera
- **else** per specificare un blocco di codice da eseguire se la stessa condizione è falsa
- **else if** per specificare una nuova condizione da verificare se la prima condizione è falsa
- **switch** per specificare più blocchi alternativi di codice da eseguire

if

Utilizziamo l'istruzione **if** per specificare un blocco di codice da eseguire se una condizione è vera.

```
if (condizione) {
  /* blocco di codice da eseguire se la
  condizione è vera */
}
```

Notare che if è in lettere minuscole. Le lettere maiuscole (If o IF) genereranno un errore.

```
<!DOCTYPE html>
<html>
  <body>
    <p>Visualizza "Buon giorno!" se l'ora è inferiore alle
    18:00:</p>
    <p id="demo">Buon pomeriggio!</p>
    <script>
      if (new Date().getHours() < 18) {
        document.getElementById("demo").innerHTML =
        "Buon giorno!";
      }
    </script>
  </body>
</html>
```

else

Utilizziamo l'istruzione **else** per specificare un blocco di codice da eseguire se la condizione è falsa.

```
if (condizione) {
   /* blocco di codice da eseguire se la
   condizione è vera */
}
else {
   /* blocco di codice da eseguire se la
   condizione è falsa */
}
```

else

Esempio di utilizzo:

```
<!DOCTYPE html>
<html>
  <body>
    <p>Cliccare sul pulsante per visualizzare un saluto in
    base all'orario:</p>
    <button onclick="myFunction()">Prova</button>
    <p id="demo"></p>
    <script>
      function myFunction() {
        var ora = new Date().getHours();
        var saluto;
        if (ora < 18) {
          saluto = "Buon giorno!";
        }
        else {
          saluto = "Buona sera!";
        }
          document.getElementById("demo").innerHTML = saluto;
      }
    </script>
  </body>
</html>
```

else if

Utilizziamo l'istruzione **else if** per specificare una nuova condizione se la prima condizione è falsa.

```
if (condizione1) {
  /* blocco di codice da eseguire se condizione1 è vera */
} else if (condizione2) {
   /* blocco di codice da eseguire se condizione1 è falsa e condizione2 è vera */
} else {
   /* blocco di codice da eseguire se condizione1 e condizione2 sono false */
}
```

else if

Esempio. Generare un saluto in base all'orario:

```
<!DOCTYPE html>
<html>
  <body>
    <p>Cliccare sul pulsante per ottenere un saluto in base
    all'orario:</p>
    <button onclick="myFunction()">Saluta</button>
    <p id="demo"></p>
    <script>
      function myFunction() {
        var saluto;
        var time = new Date().getHours();
        if (time < 14) {
          saluto = "Buongiorno!";
        }
        else if (time < 19) {
          saluto = "Buon pomeriggio!";
        }
        else {
          saluto = "Buona sera!";
        }
        document.getElementById("demo").innerHTML = saluto;
      }
    </script>
  </body>
</html>
```

switch

L'istruzione **switch** viene utilizzata per eseguire azioni diverse in base a diverse condizioni.
Utilizziamo l'istruzione switch per selezionare uno dei tanti blocchi di codice da eseguire.

```
switch(espressione) {
  case x:
    // blocco di codice
    break;
  case y:
     // blocco di codice
    break;
  default:
     // blocco di codice
}
```

switch

Come funziona:

- L'espressione switch viene valutata una volta.
- Il valore dell'espressione viene confrontato con i valori di ogni caso.
- Se c'è una corrispondenza, viene eseguito il blocco di codice associato.
- Se non c'è corrispondenza, viene eseguito il blocco di codice predefinito.

switch

Esempio. Determina il giorno della settimana:

```
<!DOCTYPE html>
<html>
  <body>
    <p id="demo"></p>
    <script>
      var giorno;
      switch (new Date().getDay()) {
        case 0:
          giorno = "Domenica";
          break;
        case 1:
          giorno = "Lunedì";
          break;
        case 2:
          giorno = "Martedì";
          break;
        case 3:
          giorno = "Mercoledì";
                 break;
        case 4:
          giorno = "Giovedì";
          break;
        case 5:
          giorno = "Venerdì";
          break;
        case 6:
          giorno = "Sabato";
      }
      document.getElementById("demo").innerHTML = "Oggi è " + giorno;
    </script>
  </body>
</html>
```

Parola chiave break

Quando viene raggiunto la parola chiave **break**, viene terminato il blocco switch.

Ciò interromperà l'esecuzione del blocco switch.

Non è necessario inserire break nell'ultima condizione in un blocco switch. Il blocco finisce comunque.

Nota: se si omette l'istruzione break, la condizione successiva verrà eseguita anche se la valutazione non corrisponde alla condizione.

Parola chiave default

La parola chiave **default** specifica il codice da eseguire se non c'è corrispondenza tra maiuscole e minuscole.
Il metodo getDay() restituisce il giorno della settimana come numero compreso tra 0 e 6.
Se oggi non è né sabato (6) né domenica (0), scrivi un messaggio predefinito:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>switch</h2>
    <p id="demo"></p>
    <script>
      var testo;
      switch (new Date().getDay()) {
        case 6:
          testo = "Oggi è sabato!";
          break;
        case 0:
          testo = "Oggi è domenica!";
          break;
        default:
          testo = "In attesa del fine settimana!";
      }
      document.getElementById("demo").innerHTML = testo;
    </script>
  </body>
</html>
```

Parola chiave default

La condizione default non deve essere necessariamente l'ultima in un blocco switch.
Se default non è l'ultima condizione, ricordarsi di terminare il caso predefinito con un'interruzione.

```
switch (new Date().getDay()) {
  default:
    testo = "In attesa del fine settimana!";
    break;
  case 6:
    testo = "Oggi è sabato!";
    break;
  case 0:
    testo = "Oggi è domenica!";
}
```

Casi comuni

A volte si desidera che alcuni casi di switch utilizzino lo stesso codice.
In questo esempio, i casi 4 e 5 condividono lo stesso blocco di codice e 0 e 6 condividono un altro blocco di codice:

```
<!DOCTYPE html>
<html>
  <body>
    <h2>switch</h2>
    <p id="demo"></p>
    <script>
      var testo;
      switch (new Date().getDay()) {
        case 4:
        case 5:
          testo = "Presto sarà il fine settimana!";
          break;
        case 0:
        case 6:
          testo = "È il fine settimana!";
          break;
        default:
          testo = "In attesa del fine settimana!";
      }
      document.getElementById("demo").innerHTML = testo;
    </script>
  </body>
</html>
```

Dettagli di commutazione

Se più casi corrispondono a uno stesso valore, viene selezionato il **primo** caso.

Se non vengono trovati casi corrispondenti, il programma continua con il caso **predefinito**.

Se non viene trovata alcuna etichetta predefinita, il programma continua con le istruzioni **dopo lo switch**.

I casi di switch utilizzano un confronto **rigoroso** (===).

I valori devono essere dello stesso tipo per corrispondere.

Un confronto rigoroso può essere vero solo se gli operandi sono dello stesso tipo.

Nel prossimo esempio non ci sarà corrispondenza per x:

Confronto rigoroso

```
<!DOCTYPE html>
<html>
  <body>
    <h2>switch</h2>
    <p id="demo"></p>
    <script>
      var x = "0";
      switch (x) {
        case 0:
          testo = "Off";
          break;
        case 1:
          testo = "On";
          break;
        default:
          testo = "Nessun valore corrispondente";
      }
      document.getElementById("demo").innerHTML = testo;
    </script>
  </body>
</html>
```