

Javascript

Javascript

Javascript è, oggi, un linguaggio di programmazione completo e dotato di tutti i più moderni costrutti, da quelli più semplici (funzioni, array...) a quelli più complessi (oggetti, ereditarietà...).

Nell'ambito delle applicazioni web è utilizzabile sia sul backend che sul frontend.

In ambito frontend Javascript è utilizzato, assieme a HTML e CSS, per dare "dinamicità" alle pagine web.

Tramite Javascript è infatti possibile:

- Modificare la struttura della pagina web, aggiungendo/rimuovendo tag, aggiungendo e rimuovendo contenuti, aggiungendo e rimuovendo classi
- Comunicare con il server anche in situazioni in cui questo non è previsto dallo standard HTML (vedi applicazioni Ajax e SPA)

Caratteristiche di Javascript

Alcune caratteristiche di Javascript:

- E' imperativo. Un programma è cioè costituito di un insieme di istruzioni da eseguire
- E' strutturato. Sono disponibili istruzioni di controllo del flusso e delle iterazioni.
- E' procedurale. E' possibile dividere il codice in blocchi richiamabili in un secondo momento.
- E' funzionale. Le funzioni sono un tipo di dato elementare. E' possibile usarle come parametri e come valori di ritorno verso altre funzioni. E' inoltre possibile definire funzioni all'interno di altre funzioni.
- E' orientato agli oggetti. E' cioè possibile definire oggetti ed utilizzare l'ereditarietà, il polimorfismo e l'incapsulamento.
- E' basato sui prototipi. Non esiste il concetto di classe ma solo di oggetti che svolgono i ruoli di "template" da cui ereditare.

Caratteristiche di Javascript

- E' a tipizzazione dinamica. Il controllo dei tipi di dato e la loro conversione (o non conversione) avviene a runtime e non a compile time. Una variabile può cambiare tipo durante il ciclo di vita dell'applicazione.
- E' a tipizzazione debole. Non è necessario dichiarare il tipo delle variabili. Viene dedotto automaticamente. Il tipo di una variabile può inoltre cambiare nel tempo.
- E' interpretato. Il codice sorgente è interpretato a runtime. E' pertanto necessario un interprete.
- Implementa un garbage collector per liberare la memoria quando le variabili non sono più utilizzate.
- E' standardizzato. Esiste uno standard chiamato ECMA-262.

Il browser come interprete javascript

Tutti i browser web contengono un interprete javascript, necessario per eseguire il codice js contenuto nelle pagine html.

I browser mettono inoltre a disposizione del codice javascript due Object Model con i quali interagire per ottenere le funzionalità desiderate:

- BOM – Browser Object Model – Rappresenta le funzionalità e le caratteristiche del browser. Per esempio: la finestra del browser con le sue proprietà, l'url attuale...
- DOM – Document Object Model – Rappresenta la struttura del documento HTML. Per esempio, tag, classi, contenuti...

Usare Javascript in pagine HTML

Sono possibili più modalità per inserire del codice javascript all'interno di una pagina web. I più frequenti sono:

- Inserimento di codice javascript all'interno dei normali tag html, come valore di attributi che rappresentano eventi
- Inserimento di codice javascript all'interno della pagina html dentro il tag `<script>`
- Import di codice javascript scritto su un file esterno. Anche in questo caso si utilizza il tag `<script>`

In generale, i tag `<script>` vanno inseriti prima della chiusura del tag `<body>`. Ci sono tuttavia eccezioni con alcuni js di terze parti che devono essere inseriti nel tag `<head>`

Usare Javascript in pagine HTML

- Inserimento di codice javascript all'interno dei normali tag html, come valore di attributi che rappresentano eventi

```
<span onclick="alert('Hello World!');"> Click me </span>
```

L'attributo onclick rappresenta un evento. Può contenere del codice javascript. Un insieme di istruzioni, come nell'esempio o la chiamata ad una funzione definita altrove.

Usare Javascript in pagine HTML

- Inserimento di codice javascript all'interno della pagina html dentro il tag <script>

```
<script type="text/javascript">  
    function hw() { alert('Hello World'); }  
</script>
```

Il tag script può essere inserito nell'head oppure in un punto qualsiasi del body.

Usare Javascript in pagine HTML

- Import di codice javascript scritto su un file esterno. Anche in questo caso si utilizza il tag `<script>`

```
<script type="text/javascript" src="nomefile.js"></script>
```

Node.js come interprete javascript

Node.js è un runtime per l'esecuzione di codice Javascript. Utilizza il motore V8 creato da Google ed utilizzato anche in Chrome.

Tramite node è possibile utilizzare javascript anche per la realizzazione di applicazioni backend, in quanto non si è più vincolati alla presenza di un browser.

Node ha una architettura molto performante ed adatta a situazioni in cui sono necessarie alte performance, come nei sistemi real time. Node è inoltre orientato ai messaggi e asincrono.

L'esecuzione di codice js in node è molto semplice. Da riga di comando: `node nomefile.js`

Javascript – Sintassi di base

Javascript – sintassi di base

- Javascript è case sensitive
- Le istruzioni vanno terminate con il ; solo se scritte sulla stessa riga.
- I blocchi di istruzioni vanno identificati con le parentesi graffe { }
- L'indentazione non è necessaria per l'esecuzione del codice (rimane fondamentale per la comprensione).
- I commenti si creano con i caratteri // per i commenti a singola linea e con i caratteri /* */ per i commenti a linea multipla
- Gli operatori matematici sono = + - * / % ++ -- += -= > >= < <= !=
- Gli operatori logici sono == !(not) &&(and) || (or)

Javascript – variabili

Le variabili si istanziano con la parola chiave `var`, senza specificarne il tipo.

Il tipo verrà dedotto dal valore assegnato alla variabile. In caso di mancata assegnazione la variabile varrà `undefined`.

```
var x = 10;
```

```
var y = '10';
```

```
var z = true;
```

```
var w = null;
```

```
var j; //undefined
```

- Qualsiasi numero (intero, decimale) è di tipo `number`
 - I valori booleani si esprimono con `true` e `false`.
 - `null` identifica una variabile inizializzata con valore non noto
 - `undefined` identifica una variabile non inizializzata
- `null` e `undefined` sono sia valori che tipi di dato.

Javascript – visibilità delle variabili

Le variabili si differenziano in locali, se definite all'interno di una funzione, o globali, se definite all'esterno di tutte le funzioni.

Le variabili locali sono "visibili" all'interno della funzione in cui sono state definite e all'interno di tutte le funzioni eventualmente definite dentro di essa.

Le variabili globali sono invece "visibili" in ogni punto del codice.

Le variabili javascript, definite con `var`, godono della proprietà di hoisting, ovvero sono visibili in ogni parte della funzione (o del blocco `<script>`) in cui sono definite, anche prima della definizione stessa (ovviamente con valore `undefined`).

Javascript – visibilità delle variabili

La visibilità di una variabile definita con `var` è molto ampia e può portare ad errori, soprattutto legati a due aspetti:

- La stessa variabile può essere definita più volte senza generare errori.
- Una variabile è visibile anche esternamente al blocco che l'ha definita.

```
var temp="valore temporaneo";  
  
if( ... )  
{  
    var temp = "nuovo valore";  
}  
  
console.log(temp);
```

Javascript – let e const

Per ovviare a questi problemi la versione 6 dello standard del linguaggio (ES6) introduce altre due parole chiave:

- **let**: permette di definire una variabile che può essere assegnata più volte ma inizializzata una volta sola
- **const**: permette di definire una costante che può essere inizializzata ed assegnata una sola volta.

L'utilizzo di let e const inibisce la proprietà di hoisting

```
let temp="valore temporaneo";  
  
if( ... ){  
    let temp = "nuovo valore"; //ERRORE  
    var temp = "nuovo valore"; //ERRORE  
    temp = "nuovo valore"; //OK  
}
```


Javascript – stringhe

Le stringhe in javascript possono essere contenute, indifferentemente, tra una coppia di apici singoli oppure tra una coppia di doppi apici oppure tra una coppia di backtick `

I carattere di apertura e di chiusura della stringa devono coincidere e all'interno di una stringa aperta con un carattere è ammesso l'utilizzo degli altri tipi di carattere.

Questa tripla possibilità permette javascript di essere inserito in più contesti, adattandosi di conseguenza.

```
var a = '10';
```

```
var b = "10";
```

```
var c = "Hello World";
```

```
var w = "Reggio nell'Emilia";
```

Javascript – stringhe

L'utilizzo di backtick ` (alt+96 su windows e option+9 su mac) permette di abilitare due caratteristiche non disponibili con gli altri delimitatori:

- stringhe multiline:

```
var s = `Hello  
World`;
```

- Interpolazione:

```
var n = 10;  
  
var s1 = `Il numero inserito è ${n}`;  
  
var s2 = `Il doppio del numero inserito è ${n*2}`;
```

Javascript – stringhe

Le stringhe sono oggetti e come tali hanno proprietà e metodi. Tra le proprietà troviamo:

- **length** – restituisce il numero di caratteri di cui la stringa è composta

Tra i metodi troviamo invece:

- **slice(), substring(), substr()** – estraggono una porzione di una stringa creandone una nuova. I tre metodi sono identici nell'obiettivo ma differenti nei parametri utilizzati.
- **replace ()** – sostituisce una parte di una stringa con un'altra
- **toUpperCase(), toLowerCase()** – converte la stringa, rispettivamente in tutti caratteri maiuscoli o minuscoli.
- **concat()** – concatena due stringhe (in alternativa utilizzare +)
- **charAt()** – estrae un carattere specificandone l'indice (in alternativa utilizzare [])
- **repeat()** – genera una nuova stringa ripetendo n volte quella iniziale

Javascript – stringhe

Tra i metodi troviamo invece:

- **split()** – converte una stringa in un array di stringhe specificando un separatore
- **indexOf(), search()** – cercano una stringa all'interno di un'altra e restituisce l'indice di partenza della prima occorrenza. Hanno lo stesso scopo ma parametri differenti
- **lastIndexOf()** – cerca una stringa all'interno di un'altra e restituisce l'indice di partenza dell'ultima occorrenza
- **includes()** – cerca una stringa all'interno di un'altra e restituisce true/false in base all'esito della ricerca
- **startsWith(), endsWidth()** – Determina se una stringa inizia/finisce con un'altra stringa passata come parametro

Javascript – valori booleani

In Javascript esistono i valori true e false ma il concetto di vero/falso è più ampio:

- In js sono considerati come **FALSO** i seguenti valori:
 - false
 - undefined
 - null
 - 0
 - ""
 - NaN
- In js è considerato vero tutto ciò che non è falso

`false || 'Hello'` è perfettamente valido e restituisce 'Hello'

Javascript == vs ===

In Javascript esiste anche l'operatore di uguaglianza ===. Ci sono differenze rispetto al classico operatore ==:

- == converte i tipi dei due operandi prima di effettuare il confronto
- === non converte i tipi dei due operandi

```
const a = "test"  
const b = "test"  
console.log(a == b) //true  
console.log(a === b) //true
```

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

```
const a = 1234  
const b = '1234'  
console.log(a == b) //true  
console.log(a === b) //false
```

```
console.log("" == false) //true  
console.log("" === false) //false
```

Javascript == vs ===

Tabella di riepilogo dell'uguaglianza effettuata con ==.

(sorgente <https://dorey.github.io/JavaScript-Equality-Table/>)

[illegible]

Javascript – Controllo del flusso

Javascript – if

L'operatore if ha una forma abbastanza standard:

```
if (condizione) { }  
  
else if(condizione) { }  
  
else { }
```

- La condizione deve essere sempre contenuta in parentesi tonde
- Le parentesi graffe non sono obbligatorie se il blocco di codice è composto da una sola istruzione
- Il blocco else if è facoltativo e ne possono esistere più di uno
- Il blocco else è facoltativo. Se presente va sempre dopo l'ultimo blocco else if

Javascript – operatore ternario

Per semplici espressioni l'operatore if può anche essere scritto in una sola riga:

```
condizione ? valoreSeVero : valoreSeFalso
```

La condizione viene valutata e viene restituito valoreSeVero o valoreSeFalso in base al risultato, rispettivamente, vero o falso.

L'operatore ternario è molto comodo per valorizzare variabili in modo condizionale:

```
var a=10;
```

```
var b=20;
```

```
var c= a<b ? a : b;
```

Javascript – switch case

Tramite l'operatore switch case è possibile valutare una espressione ed eseguire blocchi di codice distinti in base al valore.

```
switch (espressione)
{
    case valore1:
        istruzioni;
        break;

    case valore2:
    case valore3:
        istruzioni;
        break;

    default:
        istruzioni;
}
```

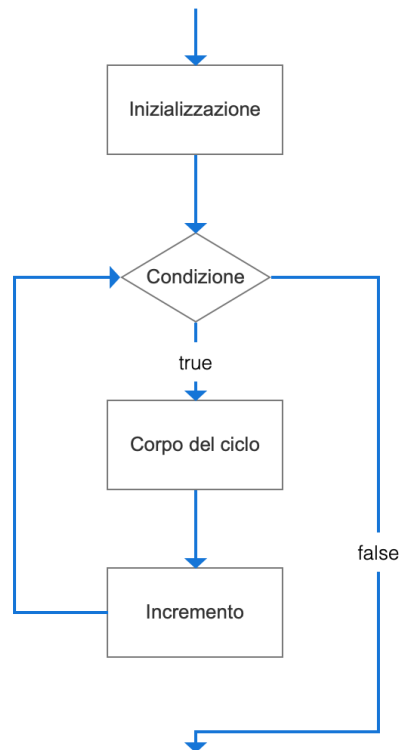
Javascript – switch case

- I blocchi di istruzioni associati ad ogni case non devono essere racchiusi in parentesi graffe.
- Il case di default, eseguito se tutti i case precedenti non sono verificati, non è obbligatorio
- La parola chiave break, posta alla fine di ogni case, interrompe il flusso di esecuzione facendolo uscire dallo switch. Non è obbligatoria ed in caso di omissione l'esecuzione continua verso i case successivi.
- Il case di default, essendo l'ultimo non necessita mai di break

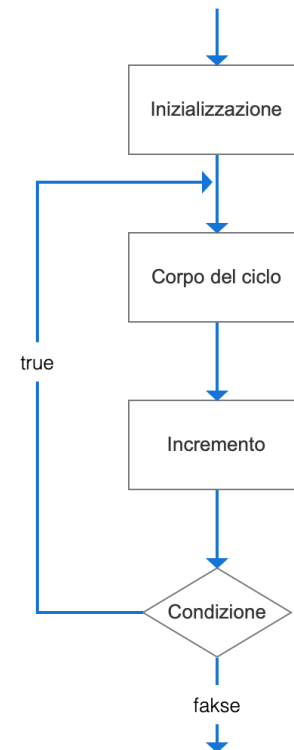
Controllo del flusso – cicli

Tramite i cicli è possibile eseguire un blocco di codice più volte. Esistono due tipologie di cicli, precondizionati e postcondizionati, con i seguenti diagrammi di flusso:

Ciclo precondizionato



Ciclo postcondizionato



Controllo del flusso – cicli

In Javascript i principali costrutti che permettono di realizzare cicli sono:

- While (precondizionato)
- do ... while (postcondizionato)
- For (precondizionato)
- For of (senza condizioni, lo vedremo in seguito)
- For in (senza condizioni, lo vedremo in seguito)

Javascript – while

L'operatore while permette di realizzare cicli preconditionati, eseguendo un blocco di codice fintanto che una determinata condizione è vera:

```
while (condizione)  
{ ... }
```

- La condizione deve essere sempre contenuta in parentesi tonde
- Le parentesi graffe non sono obbligatorie se il blocco di codice è composto da una sola istruzione
- Se la condizione rimane sempre vera si genera un loop infinito

Javascript – do while

L'operatore do while permette di realizzare cicli postcondizionati, eseguendo un blocco di codice fintanto che una determinata condizione è vera. L'esecuzione avviene comunque almeno una volta.

```
do{ ... }
```

```
while (condizione)
```

- La condizione deve essere sempre contenuta in parentesi tonde
- Le parentesi graffe non sono obbligatorie se il blocco di codice è composto da una sola istruzione
- Se la condizione rimane sempre vera si genera un loop infinito

Javascript – for

L'operatore for permette di realizzare cicli precondizionati e articolati caratterizzati da espressioni di inizializzazione, condizione di uscita e espressioni di iterazione.

```
for(espressioneIniziale; condizione; espressioneIterazione)
{ ... }
```

Per prima cosa viene eseguita l'espressioneIniziale. Viene poi valutata la condizione, se è vera si esegue il corpo contenuto tra le graffe, se è falsa l'esecuzione passa oltre.

Una volta terminata l'esecuzione del corpo viene eseguita l'espressione Iterazione e si torna a valutare la condizione.

Il ciclo va avanti fino a quando la condizione non diventa falsa.

Javascript – break e continue

All'interno degli operatori di iterazione (while, do while, for e for in) è possibile utilizzare le istruzioni break e continue con le quali modificare il flusso di esecuzione:

- break: interrompe completamente il ciclo. L'esecuzione riprende dall'istruzione successiva al ciclo.
- continue: interrompe la singola iterazione del ciclo. L'esecuzione passa all'iterazione successiva valutando le condizioni nel modo previsto dal particolare tipo di ciclo utilizzato.

Javascript – Array

Javascript – array

In Javascript gli array sono collezioni che contengono elementi anche di tipo differente.

Ogni elemento ha una posizione all'interno dell'array e la numerazione parte da 0.

In Javascript gli array sono dinamici, ovvero non hanno una dimensione predeterminata. Aumentano o diminuiscono in base al numero di elementi presenti.

Ci sono molteplici modi per creare un array:

```
var array = [elem1, elem2, ... , elemN];
```

```
var array = [ ];
```

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

```
var array = new Array(10);
```

```
var array = new Array(elem1, elem2, ... , elemN);
```

Javascript – array

L'accesso ad un elemento dell'array, sia in lettura che in scrittura, avviene specificando l'indice dell'elemento tra parentesi quadre:

```
var array = [elem1, elem2, ... , elemN];  
  
array[2] = '12';  
  
var elem1 = array[0];
```

L'aggiunta di un nuovo elemento può avvenire valorizzando un indice precedentemente non esistente, oppure utilizzando uno dei metodi messi a disposizione (vedi prossima slide).

```
var array = [1, 2];  
  
array[2] = 3; //ora l'array ha 3 elementi  
  
array[4] = 5; //ora l'array ha 5 elementi. Il quarto vale undefined
```

Javascript – array

Anche gli array sono oggetti e come tali hanno proprietà e metodi. Tra le proprietà troviamo:

- `length` – restituisce il numero di elementi presenti nell'array

Tra i metodi troviamo invece:

- `concat()` – concatena due o più array restituendone uno nuovo
- `push()` – accoda uno o più elementi ad un array esistente
- `unshift()` – inserisce uno o più elementi all'inizio di un array esistente
- `pop()` – rimuove l'ultimo elemento dell'array e lo restituisce
- `shift()` – rimuove il primo elemento dell'array e lo restituisce
- `splice()` – rimuove elementi dal centro dell'array con la possibilità anche di inserirne

Javascript – array

Tra i metodi troviamo invece:

- `toString()` – Converte l'array in stringa separando gli elementi con la virgola
- `join()` – Converte un array in stringa specificando un separatore
- `slice()` – Estrae una porzione di un array creandone uno nuovo
- `indexOf()` – Restituisce l'indice della prima occorrenza dell'elemento passato come parametro. -1 se non esiste.
- `lastIndexOf()` – Restituisce l'indice dell'ultima occorrenza dell'elemento passato come parametro. -1 se non esiste.
- `includes()` – restituisce true se l'elemento passato come parametro esiste nell'array

NB: Esistono anche altri metodi che prevedono il passaggio di funzioni. Li vedremo in seguito

Javascript – array – Spread Operator

La concatenazione di array può essere scritta più agevolmente usando l'operatore spread, composto da tre punti prima del nome di un array.

L'operatore ha l'effetto di sostituire al nome dell'array tutti i suoi elementi.

```
const array1 = [1,2,3];
```

```
const array2 = [4,5,6];
```

```
const array3 = [...array1, ...array2];
```

E' equivalente a:

```
const array3 = array1.concat(array2);
```


Javascript – for of

L'operatore for of permette di eseguire del codice su tutti gli elementi di oggetti che rappresentano collezioni (per esempio Array).

```
for(let variabile of object) { ... }
```

Per esempio:

```
var array=[1,2,3,4];
```

```
for(let elemento of array) { console.log(elemento); }
```

Javascript – Funzioni

Javascript – funzioni

Le funzioni sono blocchi di codice, ai quali è associato un nome, richiamabili da altri punti del codice.

Le funzioni possono avere parametri in ingresso e possono avere valori di ritorno. In javascript si definiscono con la parola chiave function:

```
function nomeFunzione( param1, param2,..., paramN)
{
    istruzioni;
    return retVal;
}
```

Javascript – funzioni

I parametri in ingresso sono da considerarsi come variabili locali alla funzione stessa. Sono cioè utilizzabili all'interno della funzione e all'interno di eventuali funzioni contenute.

Il valore che la funzione restituisce al suo chiamante viene identificato con la parola chiave `return`.

`Return` ha anche l'effetto di terminare immediatamente l'esecuzione della funzione

La chiamata di una funzione avviene specificandone il nome e gli eventuali parametri.

```
function somma( x, y)
{
    return x+y;
}

var s = somma(10,7);
```

Javascript – funzioni - parametri

In linea di principio il passaggio dei parametri alle funzioni viene fatto per "valore". Il valore viene cioè copiato all'interno della variabile locale della funzione e le modifiche che questa può subire non si ripercuotono sul chiamante. Esempio:

```
function doppio( x)
{
    x = x*2;
    return x;
}

var n = 10;
var r = doppio(n);

//Quanto vale r? e quanto n?
```

Javascript – funzioni - parametri

Javascript non effettua particolari controlli sui parametri passati ad una funzione.

- Se vengono passati meno parametri di quelli previsti, i mancanti avranno il valore undefined
- Se vengono passati più parametri di quelli previsti, quelli in eccesso vengono ignorati.
- Se vengono passati valori di un tipo non previsto non si genera nessun errore. Eventualmente l'errore nascerà in fase di utilizzo del parametro

Javascript – funzioni - parametri

E' possibile definire dei valori di default per alcuni parametri. In questo modo se il parametro non viene passato assumerà il valore di default anziché undefined.

Si definisce un valore di default semplicemente specificandolo nella firma della funzione:

```
function nomeFunzione( param1, param2 = 10)
```

Se non specificato il param2 varrà 10. I parametri con valore di default devono necessariamente essere gli ultimi. Un parametro senza valore di default non può seguire uno che ce l'ha.

La sintassi è equivalente a scrivere:

```
function nomeFunzione( param1, param2) {  
    param2 = param2 || 10;  
}
```

Javascript – funzioni come dati elementari

In Javascript le funzioni sono considerate come dati elementari. In particolare è possibile:

- Assegnare una funzione ad una variabile. La funzione sarà poi invocabile tramite il nome della variabile
- Passare una funzione come parametro di un'altra funzione. La funzione che riceve il parametro può invocare la funzione che gli viene passata.
- Creare funzioni anonime, ovvero senza nome, utilizzabili senza doverle salvare.

Javascript – funzioni come dati elementari

- Assegnare una funzione ad una variabile. La funzione sarà poi invocabile tramite il nome della variabile. La chiamata può avvenire solo dopo l'assegnazione, come previsto per tutte le variabili.

```
function somma( x, y) { return x+y; }
```

```
var s = somma;
```

```
console.log( s(3,5) );
```

```
var quadrato = function(x) { return x*x };
```

```
console.log( quadrato(10) );
```

Javascript – funzioni come dati elementari

- Passare una funzione come parametro di un'altra funzione. La funzione che riceve il parametro può invocare la funzione che gli viene passata

```
function somma( x, y, logger ) {  
    var s = x+y;  
    logger(s);  
    return s;  
}  
  
function logToConsole(n) { console.log(n); }  
  
var s = somma(10,20, logToConsole);
```

Javascript – funzioni come dati elementari

- Creare funzioni anonime, ovvero senza nome, utilizzabili senza doverle salvare.

```
function somma( x, y, logger ) {  
    var s = x+y;  
    logger(s);  
    return s;  
}  
  
var logToConsole = function(n) { console.log(n); }  
  
var s = somma(10,20, logToConsole);
```

Javascript – funzioni come dati elementari

- Creare funzioni anonime, ovvero senza nome, utilizzabili senza doverle salvare.

```
var s = somma(10,20, function(n) { console.log(n); } );
```

E' possibile specificare funzioni anonime con una forma più compatta, detta lambda expression o arrow functions: La funzione anonima `function(n) { console.log(n); }` può essere riscritta come: `(n) => { console.log(n); }`

In generale una funzione anonima `function(p1,p2...,pn) { istruzioni }` può essere riscritta come:

```
(p1,p2...,pn) => { istruzioni }
```

Javascript – funzioni come dati elementari

In generale una funzione anonima `function(p1,p2...,pn) { istruzioni }` può essere riscritta come: `(p1,p2...,pn) => { istruzioni }`

Se c'è solo un parametro è possibile omettere le parentesi tonde: `p1 => { istruzioni }`

Se c'è solo una istruzione è possibile omettere le parentesi graffe: `p1 => istruzione`

Se c'è solo una istruzione il suo risultato sarà considerato come valore di ritorno della funzione anonima. Non serve specificare la parola chiave `return`

Javascript – array - Find

Alcuni metodi degli array prevedono il passaggio di una funzione. Per esempio:

- `find()` – restituisce il primo elemento dell'array che soddisfa determinate condizioni

Il metodo `find` prevede che gli venga passato una funzione che:

- Riceve in input un singolo elemento dell'array
- Restituisce un booleano che vale `true` se l'elemento soddisfa la ricerca, altrimenti `false`

La funzione passata come parametro verrà eseguita per ogni elemento. Verrà restituito il primo elemento dell'array per il quale la funzione ha restituito `true`.

Javascript – array - Find

La funzione può essere passata in più modi:

1. Passando il nome di una funzione esistente

```
function criterioDiRicerca(element)
{
    return element > 0;
}
```

```
const array = [-1, 0, 1, 2];
const risultato = array.find(criterioDiRicerca);
```

Javascript – array - Find

La funzione può essere passata in più modi:

2. Passando una funzione anonima creata al volo

```
const array = [-1,0,1,2];  
const risultato = array.find(function(element)  
{  
    return element > 0;  
});
```

3. Passando una arrow function

```
const array = [-1,0,1,2];  
const risultato = array.find( element => element > 0 );  
const risultato = array.find( e => e > 0 );
```


Javascript – array - Sort

Alcuni metodi degli array prevedono il passaggio di una funzione. Per esempio:

- `sort()` – ordina l'array usando, come criterio di ordinamento, la funzione passata come parametro. Se non viene passato nessun parametro l'ordinamento è alfabetico. L'ordinamento avviene direttamente sull'array e viene anche restituito un riferimento all'array stesso

Il metodo `sort` prevede che gli venga passato una funzione che:

- Riceve in input due elementi dell'array
- Restituisce un valore numerico < 0 se il primo elemento è minore del secondo, zero se sono uguali, > 0 se il primo elemento è $>$ del secondo

La funzione passata come parametro verrà eseguita per ogni coppia di elementi, fino a determinare il nuovo ordinamento dell'array

Javascript – array - Sort

La funzione può essere passata in più modi:

1. Passando il nome di una funzione esistente

```
function criterioDiOrdinamento(element1, element2)
{
    return element2 - element1;
}
```

```
const array = [-1,0,1,2];
array.sort(criterioDiOrdinamento);
```

NB: Quale ordinamento stiamo generando con questo codice?

Javascript – array - Sort

La funzione può essere passata in più modi:

2. Passando una funzione anonima creata al volo

```
const array = [-1,0,1,2];  
array.sort(function(element1,element2)  
{  
    return element2 - element1;  
});
```

3. Passando una arrow function

```
const array = [-1,0,1,2];  
array.sort( (element1,element2) => element2 - element1);
```

Javascript – array - forEach

Alcuni metodi degli array prevedono il passaggio di una funzione. Per esempio:

- `forEach()` – esegue la funzione passata come parametro per ogni elemento dell'array, procedendo per ordine, dal primo all'ultimo elemento..

Il metodo `forEach` prevede che gli venga passato una funzione che:

- Riceve in input un elemento dell'array
- Definisce il codice da eseguire su quell'elemento

Javascript – array - forEach

La funzione può essere passata in più modi:

1. Passando il nome di una funzione esistente

```
function criterioEsecuzione(element)  
{  
    console.dir(element);  
}
```

```
const array = [-1,0,1,2];  
array.forEach(criterioEsecuzione);
```

Javascript – array - forEach

La funzione può essere passata in più modi:

2. Passando una funzione anonima creata al volo

```
const array = [-1,0,1,2];  
array.forEach(function(element)  
{  
    console.dir(element);  
});
```

3. Passando una arrow function

```
const array = [-1,0,1,2];  
array.forEach( element => console.dir(element));
```

Javascript – array - every

Alcuni metodi degli array prevedono il passaggio di una funzione. Per esempio:

- `every()` – Determina se tutti gli elementi dell'array soddisfano la funzione passata come parametro.

Il metodo `every` prevede che gli venga passato una funzione che:

- Riceve in input un elemento dell'array
- Restituisce `true` se l'elemento soddisfa il criterio

Il metodo `every` restituisce `true` solo se la funzione ha restituito `true` per ogni elemento. Nel momento in cui un elemento restituisce `false`, quelli successivi non vengono più valutati

Javascript – array - every

La funzione può essere passata in più modi:

1. Passando il nome di una funzione esistente

```
function criterioTuttiPari(element)
{
    return element % 2 == 0
}
```

```
const array = [-1,0,1,2];
const risultato = array.every(criterioTuttiPari)
```


Javascript – array - every

La funzione può essere passata in più modi:

2. Passando una funzione anonima creata al volo

```
const array = [-1,0,1,2];  
const risultato = array.every(function(element)  
{  
    return element % 2 == 0;  
});
```

3. Passando una arrow function

```
const array = [-1,0,1,2];  
const risultato = array.every( element => element % 2 == 0);
```

Javascript – array

Alcuni metodi degli array prevedono il passaggio di una funzione. Per esempio:

- `some()` – Determina se almeno un elemento dell'array soddisfa la funzione passata come parametro. La funzione deve avere la stessa firma prevista per `every`
- `filter()` – Restituisce un array con tutti gli elementi che soddisfano la funzione passata come parametro. La funzione deve avere la stessa firma prevista per `find`
- `map ()` – restituisce un array in cui ogni elemento è il risultato dell'elaborazione effettuato dalla funzione passata come parametro. La funzione riceve come input un elemento dell'array (più alcuni parametri opzionali) e restituisce l'elemento opportunamente elaborato
- `reduce()` – restituisce una "somma" calcolata su tutti gli elementi di un array. La somma viene calcolata con una funzione passata come parametro, con firma composta da due parametri, un totalizzatore e il singolo elemento dell'array. La funzione deve restituire il nuovo valore per il totale. Reduce accetta come secondo parametro opzionale il valore iniziale per il totalizzatore

Javascript – Asynchronous programming (cenni)

Javascript fetch api

- La funzione `fetch` permette di inviare una richiesta HTTP ad un server, intercettando la risposta.
- E' utilizzata per recuperare dati da API RESTFull o per inviare dati ad un backend remoto
- La chiamata è ASINCRONA, ovvero l'applicazione continua la sua esecuzione senza attendere la risposta del server
- Il risultato della chiamata sarà gestito tramite callback, promises e async/await

Javascript – Cenni di programmazione asincrona

- In precedenza abbiamo visto che Javascript è event based.
- Nessun codice viene eseguito se non in risposta ad un evento:
 - Deve essere "lanciato l'evento"
 - Deve esistere un "event handler"
- L'interprete javascript è inoltre single threaded:
 - Viene eseguito un event handler alla volta
 - Non è possibile eseguirne molteplici in parallelo
 - Fintanto che un event handler non è terminato non si può passare al prossimo
 - Esiste una coda degli eventi in attesa di esecuzione (event loop)

Javascript – Cenni di programmazione asincrona

- Un event handler che richiede un lungo tempo di esecuzione ritarderà l'esecuzione di tutti gli altri, di fatto bloccando l'applicazione
- In alcune circostanze si può generare il fenomeno dell'attesa attiva, ovvero un event handler che risulta in esecuzione (bloccando tutti gli altri) senza però eseguire nessuna istruzione.
- Si può verificare attesa attiva in tutte quelle situazioni in cui il codice si interfaccia con periferiche esterne. Per esempio:
 - Lettura / scrittura di un file
 - Comunicazione in rete (p.e. verso un backend remoto)

Javascript – Cenni di programmazione asincrona

- La programmazione asincrona permette di risolvere il problema dell'attesa attiva, ottenendo applicazioni più performanti e responsive.
- Nella programmazione asincrona il codice viene "diviso in due parti", quella eseguita prima e quella eseguita dopo l'attesa attiva.
- Il codice eseguito dopo l'attesa attiva viene inserito nell'event loop ed eseguito in seguito, liberando la possibilità di eseguire altri event handler

Javascript – Cenni di programmazione asincrona

- Una funzione asincrona (p.e. `fetch`) non restituisce un risultato ma una Promise, ovvero un oggetto che rappresenta un risultato che arriverà in modo asincrono in un secondo momento
- L'oggetto promise prevede tre metodi principali:
 - `then` → per specificare del codice da eseguire all'arrivo del risultato
 - `catch` → per specificare del codice da eseguire in caso di errore
 - `finally` → per specificare del codice da eseguire in ogni caso, sia di successo che di errore
- Ognuno dei metodi precedenti restituisce a sua volta una Promise, permettendo di concatenarli tra di loro

Javascript fetch api – Esempi di richiesta in GET

```
fetch('http://example.com')  
  .then(response => console.log(response))  
  .catch(error => console.log('Errore'));
```

L'oggetto Response contiene alcuni metodi utili per recuperare i dati ricevuti con la risposta. `text()` e `json()` restituiscono i dati rispettivamente in forma testuale e in forma di oggetto json. Entrambi sono asincroni e restituiscono una promise

```
fetch('http://example.com')  
  .then(response => response.json())  
  .then(response => console.log(response))  
  .catch(error => console.log('Errore'));
```

Javascript fetch api – Esempio di richiesta in POST

```
fetch('http://example.com', {  
    method: 'post',  
    body: JSON.stringify({ username: 'admin', pwd: '123456' })  
})  
  
.then(response => response.json())  
  
.then(data => console.log('Successo:', data))  
  
.catch(error => console.log('Errore'));
```

Javascript – Cenni di programmazione asincrona – async / await

- La programmazione asincrona può essere resa più leggibile utilizzando le parole chiave `async` e `await`, tramite le quali il codice asincrono appare simile al codice sincrono
- Il codice asincrono può essere scritto con la stessa sintassi del codice sincrono, a patto di:
 - Utilizzare la parola chiave `await` prima della chiamata della funzione asincrona
 - Inserire la parola chiave `async` prima della funzione dentro la quale si utilizza la parola chiave `await`
 - Gestire gli errori con i classici `try / catch`

Javascript – Cenni di programmazione asincrona – async / await

```
async function AsyncFunction()  
{  
    let response = await fetch('http://example.com')  
    let json = await response.json()  
    console.log(json)  
}
```

Javascript – Cenni di Express.js

Express.js

- Express.js è un framework minimalista per la realizzazione di backend in Javascript
- Utilizza node.js
- Permette la realizzazione sia di Applicazioni Web che di Web API
- E' leggero, modulare e semplice da utilizzare
- Può contare su ampia community di sviluppatori

Express.js – Installazione

- Dopo aver inizializzato un nuovo progetto node.js, è sufficiente installare il package express

```
npm init -y
```

```
npm install express
```

Express.js – Creazione di un server

- Esempio minimale di server.js che utilizza express

```
const express = require('express');  
  
const app = express();  
  
const PORT = 3000;  
  
app.listen(PORT, () => {  
    console.log(`Server in ascolto su http://localhost:${PORT}`);  
});
```


Express.js – Realizzare App Web

- Esempio di App Web che restituisce una stringa HTML

```
app.get('/', (req, res) => {  
    res.send('<h1>Benvenuto nel mio sito!</h1><p>Questa è una  
pagina HTML generata da Express.</p>');  
});
```

Express.js – Realizzare App Web

- Esempio di App Web che restituisce un file html statico

```
const path = require('path');  
app.get('/', (req, res) => {  
    res.sendFile(path.join(__dirname, 'public', 'index.html'));  
})
```

Express.js – API in GET

- Esempio di un api endpoint in GET, senza parametri

```
app.get('/api/saluto', (req, res) => {  
    res.json({ message: "Ciao, benvenuto nella mia API!" });  
});
```

Express.js – API in GET con parametri

- Esempio di un api endpoint in GET, con un parametro nell'url

```
app.get('/api/saluto/:nome', (req, res) => {  
    const nome = req.params.nome;  
    res.json({ message: `Ciao, ${nome}!` });  
});
```

Express.js – API in GET con parametri

- Esempio di un api endpoint in GET, con un parametri in query string

```
app.get('/api/calcola', (req, res) => {  
    const a = parseInt(req.query.a) || 0;  
    const b = parseInt(req.query.b) || 0;  
    res.json({ risultato: a + b });  
});
```

Express.js – CORS (Cross Origin Resource Sharing)

- Per default i browser bloccano le richieste inviate verso backend differenti rispetto a quello di origine della pagina
- L'applicazione di backend può però essere configurata per accettare tali richieste
- In node.js è necessaria l'installazione del package cors
 - `npm install cors`

Express.js – CORS (Cross Origin Resource Sharing)

- **Abilitazione delle richieste cors da tutti i domini di origine**

```
const cors = require('cors');  
  
app.use(cors());
```

- **Abilitazione delle richieste cors da un solo dominio di origine**

```
const cors = require('cors');  
  
app.use(cors({  
  origin: 'http://miodominio.com'  
}));
```

Javascript - Objects

Oggetti

Gli oggetti sono strutture dati che permettono una maggiore integrazione rispetto agli array e alle funzioni.

Tramite gli oggetti possiamo creare strutture dati che integrano sia dati (**campi o proprietà**) che funzionalità (**metodi**).

Un oggetto è costituito, principalmente, da alcune componenti:

- Un insieme di campi/proprietà
- Un insieme di metodi
- Un insieme di costruttori

Oggetti - Creazione

La creazione di un oggetto avviene specificando una serie di coppie chiave/valore, dove la chiave è il nome del campo/metodo e il valore è un valore singolo o una funzione.

```
const cerchio = {  
    raggio : 1,  
    posizioneX : 10,  
    posizioneY : 10,  
  
    area: function()  
    {  
        return raggio*raggio*3.14;  
    }  
}
```

L'utilizzo di campi e metodi avviene con la notazione puntata o con le []:

```
cerchio.raggio=10; console.dir(cerchio.area); console.dir(cerchio["area"]);
```

Oggetti - Creazione

I campi possono essere di qualsiasi tipo: valori semplici (per esempio una stringa o un numero), array, altri oggetti:

```
const cerchio = {  
  raggio : 1,  
  posizione : {  
    x: 10,  
    y: 10  
  },  
  area: function()  
  {  
    return raggio*raggio*3.14;  
  }  
}
```

Oggetti – Creazione con factory Function

La creazione di un oggetto può essere migliorata, evitando la duplicazione del codice, utilizzando una Factory Function

```
function creaCerchio(raggio, location)
{
    return {
        raggio : raggio,
        location: location,

        area: function()
        {
            return raggio*raggio*3.14;
        }
    }
}
```

Oggetti – Creazione con factory Function

La sintassi può essere semplificata nel caso in cui chiave e valore siano gli stessi:

```
function creaCerchio(raggio, location)
{
    return {
        raggio,
        location,

        area: function()
        {
            return raggio*raggio*3.14;
        }
    }
}
```

Oggetti – Creazione con factory Function

Anche la scrittura del metodo può essere semplificata:

```
function creaCerchio(raggio, location)
{
    return {
        raggio,
        location,

        area()
        {
            return raggio*raggio*3.14;
        }
    }
}
```

Oggetti – Creazione con Costruttore

Possiamo creare un oggetto tramite un costruttore o Constructor Function. Il costruttore avrà lo stesso nome dell'oggetto. Esempio:

```
function Cerchio(raggio, location)  
  
{  
  
    this.raggio = raggio;  
    this.location = location;  
    this.area = function()  
    {  
        return raggio*raggio*3.14;  
    }  
  
}
```

La generazione dell'oggetto avviene con `let cerchio = new Cerchio(10, {x:10,x:10 })`

Oggetti – Creazione con Costruttore

```
function Cerchio(raggio, location)
{
    this.raggio = raggio;
    this.location = location;
    this.area = function()
    {
        return raggio*raggio*3.14;
    }
}

let cerchio = new Cerchio(10, {x:10,x:10 } )
```

1. La parola chiave new genera un nuovo oggetto
2. La chiamata al costruttore Cerchio inizializza l'oggetto
3. All'interno del costruttore this fa riferimento all'oggetto appena creato al punto 1
4. Tramite this aggiungo i campi e i metodi che mi servono
5. Il costruttore restituisce l'oggetto creato
6. Non è necessario eseguire return dell'oggetto. E' implicito vista la chiamata a new

Oggetti – Creazione con Costruttore

Un modo alternativo per definire un costruttore:

```
Let Cerchio = function(raggio, location)
{
    this.raggio = raggio;
    this.location = location;
    this.area = function()
    {
        return raggio*raggio*3.14;
    }
}
```

```
let cerchio = new Cerchio(10, {x:10,x:10 } )
```

Oggetti – Creazione con Costruttore

All'interno del costruttore posso definire campi che non saranno disponibili all'esterno dell'oggetto, con la dot notation. E' sufficiente definirle normalmente, senza la parola chiave `this`.

```
function Cerchio(raggio, location)
{
    const pi = 3.14;

    this.raggio = raggio;
    this.location = location;
    this.area = function()
    {
        return raggio*raggio*pi;
    }
}

let cerchio = new Cerchio(10, {x:10,x:10 } )
console.log(cerchio.pi); //ERRORE
```

Oggetti

Gli oggetti in javascript sono dinamici. E' possibile, in qualsiasi momento, aggiungere o rimuovere sia campi che metodi.

Per aggiungere un campo/metodo è sufficiente assegnarlo, mentre la cancellazione avviene con l'istruzione delete:

```
const cerchio = {  
    raggio : 1  
};
```

```
cerchio.colore = 'verde';  
delete cerchio.raggio;
```

Value type e reference type

In javascript i tipi di dati si dividono in due famiglie

- **Value type:** Tutti i tipi di dati elementari (number, string, boolean, undefined, null...)
- **Reference type:** Oggetti, funzioni e array

Le differenze tra le due tipologie è legata alla modalità di gestione della memoria.

Nei value type l'identificatore "punta" direttamente al valore memorizzato. Copiando un value type abbiamo quindi un duplicazione del valore.

Nei reference type l'identificatore "punta" ad un riferimento all'area di memoria (heap) in cui è memorizzato il valore. Copiando un reference type abbiamo la duplicazione del puntatore, generando due variabili che puntano allo stesso valore.

Javascript – for in

L'operatore for in permette di eseguire del codice su tutte i campi/proprietà di un oggetto.

```
for(let variabile in object) { ... }
```

Per esempio:

```
var myObject = { a:1, b:2 }  
  
for(let prop in myObject)  
{  
  
    console.log(prop);  
  
    console.log(myObject[prop]);  
  
}
```

Javascript – for of su oggetti

Il ciclo for of è utilizzabile solo su collezioni (array) e non, direttamente, su oggetti.

E' però possibile ottenere l'elenco dei campi, strutturato come array, utilizzando la proprietà keys dell'oggetto generico Object.

Per esempio:

```
var myObject = { a:1, b:2 }  
  
for (let prop of Object.keys(myObject))  
{  
  
    console.log(prop);  
  
    console.log(myObject[prop]);  
  
}
```

Clonare un oggetto

La dinamicità degli oggetti e la possibilità di iterare sui campi ci fornisce la possibilità di clonare un oggetto:

```
let myObject = { a:1, b:2 };  
  
let newObject = { };  
  
for(let prop in myObject)  
    newObject[prop] = myObject[prop];
```

Clonare un oggetto

Un altro modo per clonare un oggetto è quello di usare il metodo assign dell'oggetto Object.

```
let myObject = { a:1, b:2 };  
let newObject = Object.assign({}, myObject);
```

I campi e i metodi di myObject saranno aggiunte all'oggetto passato come primo parametro. In caso non fosse un oggetto vuote, il risultato sarà un merge tra i due oggetti.

Clonare un oggetto

Un terzo modo per clonare un oggetto è quello di usare lo spread operator già visto per gli array:

```
let myObject = { a:1, b:2 };  
let newObject = { ...myObject };
```

Oggetti di sistema

Javascript implementa un grande numero di oggetti con funzionalità già pronte all'uso.

Ogni oggetto prevede metodi, la cui documentazione completa può essere trovata su: developer.mozilla.org

Alcuni oggetti:

- Math: Con costanti e operazioni matematiche
- String: Con operazioni per la manipolazione di stringhe
 - NB: Metodi utilizzabili direttamente su variabili di tipo string
- Date: Con operazioni su date e orari

Javascript - OOP

Javascript OOP

Javascript permette di realizzare oggetti conformi con i principi della programmazione orientata agli oggetti (OOP – Object Oriented Programming), ovvero:

- Incapsulamento: Capacità di nascondere i dettagli implementativi, permettendo agli utilizzatori di un oggetto di usare solamente metodi e proprietà rilevanti
- Ereditarietà: Capacità di definire un oggetto che eredita, ed estende, metodi, campi e proprietà di un altro oggetto.
- Polimorfismo: Capacità di un oggetto di modificare il suo tipo

Javascript OOP - Incapsulamento

L'incapsulamento si ottiene, in fase di costruzione dell'oggetto, creando variabili (campi) locali SENZA usare la parola chiave `this`.

L'utilizzo della parola chiave `this` stabilisce la parte PUBBLICA dell'oggetto, mentre tutto il resto stabilisce la parte PRIVATA che sarà utilizzabile solamente dai metodi dell'oggetto.

```
function Cerchio(raggio)
{
    const pi = 3.14; //PRIVATA
    this.raggio = raggio; //PUBBLICA
}
```

Javascript OOP - Incapsulamento

Per un miglior incapsulamento è necessario poter controllare l'accesso ai campi pubblici. Per alcuni campi è accettabile un accesso in lettura e in scrittura mentre per altri è accettabile l'accesso in sola lettura o in sola scrittura.

Si può ottenere questo comportamento combinando un campo privato con un metodo pubblico:

```
function Cerchio(raggio)
{
    const pi = 3.14; //PRIVATA
    this.getPi = function() { return pi; }
    this.raggio = raggio; //PUBBLICA
}
```

L'approccio, sebbene funzionale, non è ottimale in quanto richiede l'utilizzo di un metodo.

Javascript OOP – Incapsulamento – getter e setter

Un modo alternativo è quello di definire proprietà, ovvero variabili virtuali, pubbliche, che si appoggiano su campi privati.

```
function Cerchio(raggio)
{
    const pi = 3.14; //PRIVATA
    //PROPRIETA' PUBBLICA in sola lettura
    Object.defineProperty(this, 'pi',
    {
        get : function() { return pi }
    });
    this.raggio = raggio; //PUBBLICA
}
```

Javascript OOP – Incapsulamento – getter e setter

```
function Voto()  
{  
    let _punteggio; //PRIVATA  
    //PROPRIETA' PUBBLICA in lettura e scrittura  
    Object.defineProperty(this, 'Punteggio',  
    {  
        get : function() { return _punteggio },  
        set: function(value) {  
            if( value > 30 )  
                _punteggio = 30;  
            else  
                _punteggio = value  
        }  
    }  
}
```


Javascript – Ereditarietà prototipale

In generale, l'ereditarietà (inheritance) è una relazione tra due classi che permette ad una classe di ereditare il comportamento da un'altra classe, per poi estenderlo.

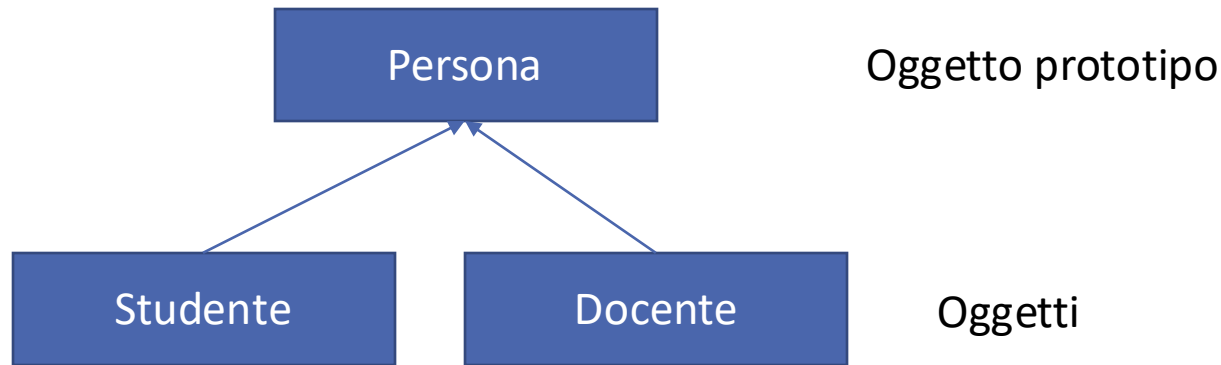
La relazione tra due classi è di tipo is a: se la classe B eredita dalla classe A possiamo dire che B is a A. Esempio:

- Classe Persona
- Classe Studente che eredita da Persona → Studente is a Persona

In Javascript non esiste il concetto di classe ma solo quello di oggetto. Pertanto, l'ereditarietà avviene collegando un oggetto ad un altro oggetto, chiamato prototipo, dal quale erediterà metodi e campi.

Classi – Ereditarietà prototipale

L'ereditarietà tra oggetti può essere rappresentata graficamente:



NB: In js tutti gli oggetti, ad eccezione di `Object`, hanno un prototipo. Se non specificato esplicitamente questo sarà `Object`, che pertanto può essere considerato la radice della gerarchia di oggetti.

Javascript – Ereditarietà prototipale

La possibilità di ereditare, ed estendere, i comportamenti è possibile grazie al modo in cui javascript cerca le proprietà, i campi e i metodi quando questi vengono utilizzati.

1. Per prima cosa vengono cercati sull'oggetto attuale
2. Se non vengono trovati, vengono cercati sul prototipo
3. Eventualmente si risale nella ricerca fino ad arrivare ad Object

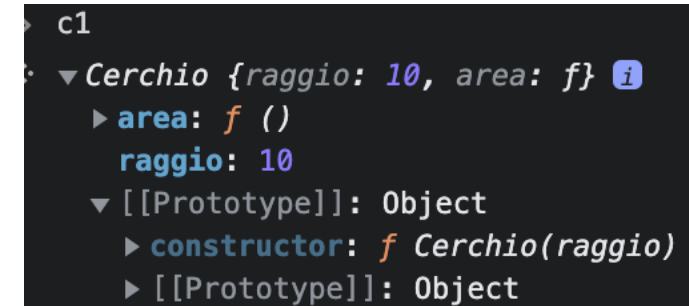
E' pertanto possibile una ereditarietà multilivello.

Javascript – Ereditarietà prototipale

Ogni volta che si crea un oggetto con la parola chiave new, questo ha come prototipo un oggetto composto dal solo costruttore (e dal riferimento al prototipo di livello superiore).

```
function Cerchio(raggio)
{
    this.raggio = raggio;
    this.area = function()
    {
        return this.raggio * this.raggio * 3.14;
    }
}

const c1 = new Cerchio();
```



```
> c1
▼ Cerchio {raggio: 10, area: f} ⓘ
  ► area: f ()
  ► raggio: 10
  ▼ [[Prototype]]: Object
    ► constructor: f Cerchio(raggio)
    ► [[Prototype]]: Object
```

Javascript – Ereditarietà prototipale

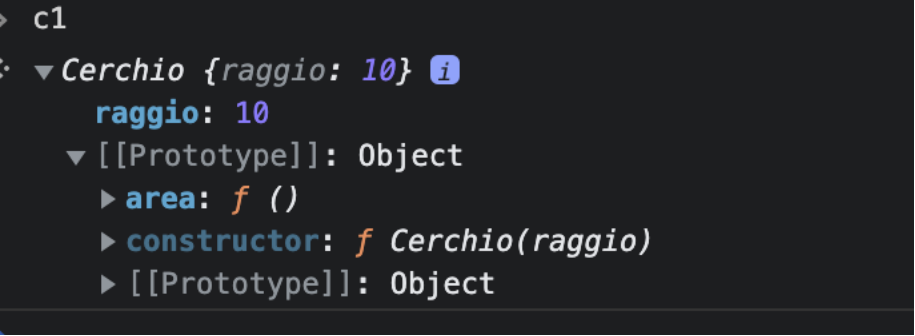
La creazione di più oggetti dello stesso tipo comporta la non necessaria duplicazione di tutti i metodi (mentre rimane necessaria la duplicazione dei campi).

La soluzione è quella di aggiungere i metodi sul prototipo (prototype methods) e non nell'oggetto (instance/own methods).

```
function Cerchio(raggio)
{
    this.raggio = raggio;
}

Cerchio.prototype.area = function()
{
    return this.raggio * this.raggio * 3.14;
}

const c1 = new Cerchio(10);
```



```
> c1
▼ Cerchio {raggio: 10} ⓘ
  raggio: 10
  [[Prototype]]: Object
    ► area: f ()
    ► constructor: f Cerchio(raggio)
    ► [[Prototype]]: Object
```

Javascript – Override dei metodi

E' possibile modificare il comportamento di un metodo già definito in un prototipo semplicemente ridefinendolo, come instance method o come prototype method.

```
function Cerchio(raggio)
{
    this.raggio = raggio;
}

Cerchio.prototype.toString = function()
{
    return 'Cerchio con raggio ' + this.raggio;
}
```

```
const c1 = new Cerchio(10);
console.dir(c1.toString());
```

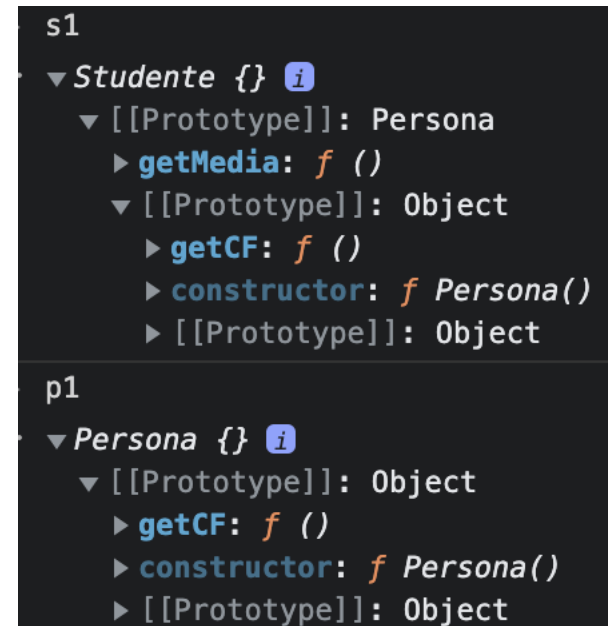


```
Cerchio con raggio 10
```

Javascript – Ereditarietà prototipale

La creazione di un oggetto che eredita da un altro oggetto avviene tramite il metodo `create` dell'oggetto `Object`. Il metodo viene usato per sovrascrivere il prototipo dell'oggetto derivato con quello dell'oggetto da cui ereditare

```
function Persona() { }  
Persona.prototype.getCF = function(){}  
  
function Studente() { }  
Studente.prototype = Object.create(Persona.prototype);  
Studente.prototype.getMedia = function(){}  
  
p1 = new Persona();  
s1 = new Studente();
```



The screenshot displays the prototype chain for two objects, `s1` and `p1`, in a dark-themed console. For `s1`, the `Studente` object is shown with its `[[Prototype]]` set to `Persona`, which in turn has its `[[Prototype]]` set to `Object`. For `p1`, the `Persona` object is shown with its `[[Prototype]]` set to `Object`. Both objects show their respective methods like `getMedia` and `getCF`.

```
s1  
▼ Studente {} ⓘ  
  ▼ [[Prototype]]: Persona  
    ► getMedia: f ()  
    ▼ [[Prototype]]: Object  
      ► getCF: f ()  
      ► constructor: f Persona()  
      ► [[Prototype]]: Object  
  
p1  
▼ Persona {} ⓘ  
  ▼ [[Prototype]]: Object  
    ► getCF: f ()  
    ► constructor: f Persona()  
    ► [[Prototype]]: Object
```

Javascript – Ereditarietà prototipale - costruttori

Nel caso, frequente, in cui gli oggetti abbiano costruttori a cui passare parametri, è necessario ripristinarne il corretto funzionamento:

```
function Persona(nome,cf)
{
    this.nome = nome;
    this.cf = cf;
}
```

```
let Studente = function(matricola)
{
    this.matricola = matricola;
}
```

```
Studente.prototype = new Persona("Mario Rossi","MRRSS");
Studente.prototype.constructor = Studente;
let s = new Studente("A2345");
```


Javascript – Polimorfismo

In Javascript il polimorfismo è "automaticamente" ottenuto grazie alla tipizzazione debole del linguaggio.

```
function Persona(nome,cf)
{
    this.whoami = function() { console.log("Persona"); }
}
```

```
let Studente = function(matricola)
{
    this.whoami = function() { console.log("Studente"); }
}
```

```
Studente.prototype = new Persona("Mario Rossi","MRRSS");
Studente.prototype.constructor = Studente;
let p = new Persona("Luigi Bianchi","BCHLGG");
p = new Studente("A2345");
p.whoami(); //STAMPA Studente
```

Document Object Model

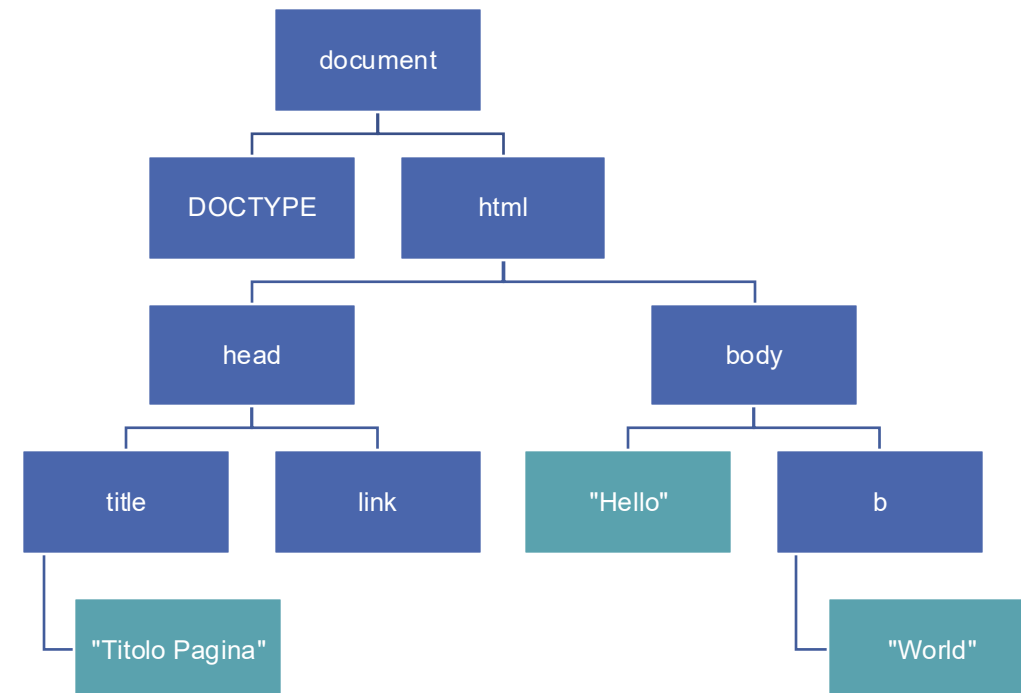
DOM

Il Document Object Model è una rappresentazione ad albero di oggetti della struttura di una pagina HTML.

Gli oggetti sono utilizzabili dal codice javascript tramite metodi/proprietà che questi espongono.

La pagina HTML è rappresentata nel DOM come una struttura ad albero, dove ogni nodo rappresenta un tag HTML o un contenuto testuale di un nodo HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Titolo Pagina</title>
    <link rel="stylesheet" href="./style.css">
  </head>
  <body>
    Hello <b>World</b>
  </body>
</html>
```



DOM

Ogni nodo del DOM è un oggetto che implementa interfacce che dipendono dalla sua tipologia.

Tutti i nodi implementano le interfacce generiche **Node** e **Element**, che forniscono metodi e proprietà di base, e interfacce specifiche quali **HTMLHtmlElement**, **HTMLParagraphElement**, **HTMLDivElement** ... che forniscono metodi e proprietà associate alla particolare tipologia di nodo.

L'elenco completo delle interfacce e delle relative proprietà/metodi è disponibile nel sito:

<https://developer.mozilla.org/en-US/docs/Web/API>

DOM – oggetto document

L'oggetto document è implementa la classe Element ed è sempre presente in quanto rappresenta la radice del DOM.

Partendo da questo oggetto possiamo "navigare" la struttura dell'albero utilizzando i metodi disponibili.

```
console.log(document); //restituisce l'HTML della pagina
```

```
console.dir(document); //restituisce la le proprietà e gli oggetti che compongono document
```

DOM – Selezionare elementi

L'oggetto document mette a disposizione una serie di metodi tramite i quali navigare nel DOM e selezionarne porzioni:

```
Element getElementById(elementId);
```

Seleziona l'elemento che ha l'attributo id uguale a quello passato come parametro. Restituisce null in caso non esista. L'elemento restituito implementa a sua volta l'interfaccia Element.

Nel caso di più elementi con lo stesso id i browser tipicamente restituiscono il primo. Ma non è un comportamento standard in quanto L'ATTRIBUTO ID DEVE ESSERE UNIVOCO.

DOM – Selezionare elementi

```
NodeList getElementsByName(elementName) ;
```

Seleziona tutti gli elementi che hanno l'attributo name uguale a quello passato come parametro. Restituisce null in caso non ne esista nessuno.

Il risultato è una collezione di elementi che implementano l'interfaccia Element.

```
let elements = document.getElementsByName('username');  
elements.forEach(item => console.log(item) );
```

DOM – Selezionare elementi

Anche l'interfaccia `Element`, implementata anche da `document`, mette a disposizione una serie di metodi tramite i quali navigare nel DOM e selezionarne porzioni.

In questo caso la ricerca avviene solamente nel sottoalbero che ha come radice l'elemento su cui è stato chiamato il metodo.

```
NodeList getElementsByTagName (tagName) ;
```

Seleziona tutti gli elementi realizzati con il tag uguale a quello passato come parametro. Restituisce null in caso non ne esista nessuno.

Il risultato è una collezione di elementi che implementano l'interfaccia `Element`.

```
var forms = document.getElementsByTagName('form'); //cerca in tutto il documento html  
var inputs = forms[0].getElementsByTagName('input'); //cerca nella prima form
```


DOM – Selezionare elementi

```
NodeList getElementsByClassName(className);
```

Seleziona tutti gli elementi che hanno la classe passata come parametro. Restituisce null in caso non ne esista nessuno.

Il risultato è una collezione di elementi che implementano l'interfaccia Element.

La ricerca avviene solamente nel sottoalbero che ha come radice l'elemento su cui è stato chiamato il metodo.

Come parametro possono essere passati i nomi di più classi, separati da spazio. Verranno selezionati tutti gli elementi che contengono tutte le classi specificate, indipendentemente dall'ordine.

DOM – Selezionare elementi

```
NodeList querySelectorAll(selector);
```

Seleziona tutti gli elementi che soddisfano il selettore css passato come parametro. Restituisce null in caso non ne esista nessuno.

Il risultato è una collezione di elementi che implementano l'interfaccia Element.

La ricerca avviene solamente nel sottoalbero che ha come radice l'elemento su cui è stato chiamato il metodo.

Esiste anche:

```
Element querySelector(selector);
```

che restituisce il primo elemento che soddisfa il selettore css

DOM – Selezionare elementi

Dato un nodo che implementa la classe `Element` è possibile navigare nell'albero anche con alcune proprietà:

- `Element parentNode` - restituisce il nodo padre
- `NodeList childNodes` - restituisce una collezione con tutti i nodi figli (di tutti i tipi)
- `Node firstChild` - restituisce il primo figlio (di qualsiasi tipo)
- `Node lastChild` - restituisce l'ultimo figlio (di qualsiasi tipo)
- `Node nextSibling` - restituisce il fratello successivo (di qualsiasi tipo)
- `Node previousSibling` - restituisce il fratello precedente (di qualsiasi tipo)

DOM – Selezionare elementi

Molti metodi della slide precedente restituiscono i nodi figli di qualsiasi tipo. Molto spesso però cerchiamo solo i nodi di tipo `Element`. Esistono metodi per accedere solamente a questi nodi:

- `NodeList children` - restituisce una collezione con tutti i nodi figli di tipo `Element`
- `Element firstElementChild` - restituisce il primo figlio di tipo `Element`
- `Element lastElementChild` - restituisce l'ultimo figlio di tipo `Element`
- `Element nextElementSibling` - restituisce il fratello successivo di tipo `Element`
- `Element previousElementSibling` - restituisce il fratello precedente di tipo `Element`
- `int childElementCount` - restituisce il numero di figli di tipo `Element`

DOM – Modificare la struttura ad albero

L'oggetto document contiene metodi per creare nuovi elementi:

- `Element createElement(tagName)` - restituisce un oggetto Element relativo al tag passato come parametro
- `Text createTextNode(text)` - restituisce un oggetto text con il contenuto passato come parametro

I singoli nodi contengono metodi per clonare/aggiungere/rimuovere elementi:

- `Node appendChild(newChild)` – Accoda l'elemento passato come parametro ultimo figlio del chiamante.
- `Node insertBefore(newChild, refChild)` – Inserisce il nuovo elemento prima dell'elemento refChild. RefChild deve essere un figlio del chiamante.
- `Node removeChild(child)` – Rimuove dal DOM il figlio passato come parametro.
- `Node replaceChild(newChild, oldChild)` – sostituisce oldChild con newChild

DOM – Modificare il contenuto di un elemento

Gli Element hanno alcune proprietà che permettono di leggerne e modificarne il contenuto:

- `innerText` - restituisce tutto il testo, VISIBILE a video, presente all'interno di un elemento e dei suoi sottoelementi. Non vengono restituiti i tag html
- `innerHTML` - restituisce tutto l'html presente all'interno di un elemento e dei suoi sottoelementi. Vengono pertanto restituiti anche i tag html.
- `outerText` – identico a `innerText` se utilizzato in lettura. In scrittura però sostituisce anche l'elemento corrente.
- `outerHTML` - identico a `innerHTML` se utilizzato in lettura. In scrittura però sostituisce anche l'elemento corrente.
- `textContent` – analogo a `innerText` con la differenza che restituisce in contenuto anche di eventuali tag presenti nel documento ma non visualizzati a video

DOM – innerText vs textContent

```
<ul id="elenco">
  <li>Html</li>
  <li>Css</li>
  <li>Javascript</li>
  <li style="display:none">Typescript</li>
</ul>
```

- `document.getElementById("elenco").innerText` restituisce Html, Css, Javascript
- `document.getElementById("elenco").textContent` restituisce Html, Css, Javascript e Typescript

DOM – Modificare gli attributi di un elemento

Gli Element hanno alcuni metodi che permettono di accedere agli attributi:

- `getAttribute(name)` - restituisce il valore dell'attributo passato come parametro.
- `setAttribute(name, value)` – imposta l'attributo passato come primo parametro al valore passato come secondo parametro
- `removeAttribute(name)` – rimuove l'attributo passato come parametro
- `hasAttribute(name)` – verifica l'esistenza dell'attributo passato come parametro. Restituisce un boolean.

DOM – Modificare gli attributi di un elemento

Gli attributi sono anche mappati come proprietà dell'oggetto Element, utilizzando la seguente convenzione per i nomi:

- Se l'attributo è formato da una sola parola allora la proprietà ha lo stesso nome dell'attributo, tutto in minuscolo.
- Se l'attributo è formato da più parole (per esempio readonly), la proprietà utilizza la notazione camel case con la prima lettera in minuscolo (per esempio readOnly)
- Se l'attributo ha un nome uguale ad una parola chiave di javascript (per esempio for) allora la proprietà avrà il prefisso html e il nome dell'attributo sarà scritto con la prima lettera maiuscola (per esempio htmlFor)
- Il tipo di dato della proprietà riflette il tipo di dato del relativo attributo (per esempio: readOnly --> boolean)

DOM – Modificare gli attributi di un elemento

L'attributo class ha una gestione dedicata tramite due proprietà:

- `className` – Restituisce l'intero contenuto dell'attributo
- `classList` – Restituisce una collezione contenente tutte le classi. E' una proprietà in sola lettura e può essere modificata con i metodi `add(nomeClasse)`, `remove(nomeClasse)` e `toggle(nomeClasse)`

DOM – Gestione degli eventi

La gestione degli eventi serve per eseguire codice javascript per nel momento in cui si verifica una determinata condizione. Per esempio:

- La pagina ha terminato il suo caricamento (onload)
- L'utente ha cliccato su un elemento (onclick)
- Il mouse si è soffermato su un elemento (onmouseover)
- Il mouse ha lasciato un elemento (onmouseout)
- E' stato premuto un pulsante (onkeydown)
- ...

Gli eventi sono pertanto associati ad elementi. E' cioè l'elemento che definisce il codice javascript da eseguire a fronte di un determinato evento.

DOM – Gestione degli eventi

Esistono 3 modi per associare codice javascript ad un evento:

Direttamente nell'HTML:

```
<element eventName="javascript">
```

In javascript sfruttando il mapping attributi/proprietà:

```
element.eventName = function( ) {...}
```

In javascript sfruttando l'apposito metodo:

```
element.addEventListener("eventName", function( ) {...} )
```

Browser Object Model

BOM – Browser object model

Le proprietà del browser sono disponibili per javascript tramite un oggetto, dal formato non completamente standardizzato, chiamato window. Contiene diverse proprietà, tra cui:

- console – restituisce l'oggetto tramite il quale accedere alla console
- document – restituisce l'oggetto radice del DOM
- history – restituisce un oggetto che contiene la storia di navigazione della pagina. Su questo oggetto è possibile invocare i metodi back() e forward()
- innerHeight e innerWidth - restituiscono rispettivamente l'altezza e la larghezza dell'area disponibile per la pagina.
- outerHeight e outerWidth - restituiscono rispettivamente l'altezza e la larghezza della finestra del browser
- location – restituisce un oggetto che rappresenta l'url della pagina corrente
- localStorage – restituisce un oggetto che permette di memorizzare dati sul client.