Umberto Emanuele

JavaScript 2
Giorno 4

Novembre 2023

Gli oggetti: basi

Gli oggetti in JavaScript

In JavaScript praticamente ogni elemento può essere un oggetto: variabile, array, date, stringhe, espressioni matematiche.

La keyword fondamentale dell'oggetto è new. in particolare in riferimento a:

- stringhe
- numeri
- booleani

Una pratica comune suggerisce di definire un oggetto con la keyword const

```
const pet = {}
```

Per visualizzare un elemento si richiama l'oggetto con un valore:

```
pet.colore => rosso
```

Un oggetto può contenere più valori tipicamente associati ad un nome:

```
nome : valore
```

```
var pet = {
specie : "gatto",
nome : "Tigro",
colore : "rosso",
anni : 4
}
```

Gli attributi principali degli oggetti sono:

- Proprietà, cioè le caratteristiche possedute e che definiscono gli elementi dell'oggetto.
- Metodi, cioè funzioni che possono essere svolte dall'oggetto.

Le proprietà sono definite dai nomi associati ai valori

```
const pet = {
specie : "gatto",
nome : "Tigro"
}
```

specie e nome sono due proprietà dell'oggetto pet

Il metodo definito da una funzione anonima è parte integrante dell'oggetto

```
const pet = {
specie : "gatto",
nome : "Tigro",

tipo: function (){
return this.specie + this.nome;
}
}
```

this = riferimento all'oggetto corrente

Dato un certo oggetto possiamo gestire e manipolare le sue proprietà attraverso il metodo costruttore (dell'oggetto).

Analogamente alle funzioni, il costruttore accetta dei parametri per creare un modello di riferimento.

Per buona pratica il nome del costruttore è scritto in maiuscolo.

```
//Costruttore
function Pet (razza, nome, colore, anni){
this.razza = razza,
this.nome = nome,
this.colore = colore,
this.anni = anni
}

//Oggetto specifico
const gatto = new Pet("soriano", "Tigro",
"rosso", 7);
```

JavaScript possiede un certo numero di oggetti nativi, quindi predefiniti.

Questi posseggono altrettanti costruttori predefiniti.

Costruttori built-in

```
new String()
new Number()
new Boolean()
new Object()
new Array()
new RegExp()
new Function()
new Date()
```

Esempio:

```
const nome = new String();
const somma = new Number
```

Quando abbiamo necessità di aggiungere una proprietà o un metodo ad un oggetto già esistente, possiamo farlo agevolmente:

```
oggetto.nuovaProprietà = valore;

oggetto.proprietà = funzione;
```

```
//Costruttore
function Pet (razza, nome, colore, anni){
this.razza = razza,
this.nome = nome,
this.colore = colore,
this.anni = anni
}
```

```
//Oggetto specifico
const gatto = new Pet("soriano", "Tigro",
"rosso", 7);

//Aggiungo una proprietà
gatto.genere = "maschio";

//Aggiungo un metodo
gatto.tipo = function(){
return this.razza + this.nome;
}
```

Ogni oggetto in JavaScript fa riferimento implicito ad un prototipo attraverso cui è possibile condividere (ereditare) proprietà e metodi.

La keyword prototype richiama questo modello di riferimento che può essere utilizzato per aggiungere proprietà e metodi ad un costruttore esistente.

```
//Costruttore
function Pet (razza, nome, colore, anni){
this.razza = razza,
this.nome = nome,
this.colore = colore,
this.anni = anni
}
```

```
//Aggiungo una proprietà al costruttore
Pet.prototype.genere = “femmina”;

//Aggiungo un metodo al costruttore
Pet.prototype.tipo = function(){
return this.razza + this.nome;
}
```

Programmazione orientata agli oggetti in JS

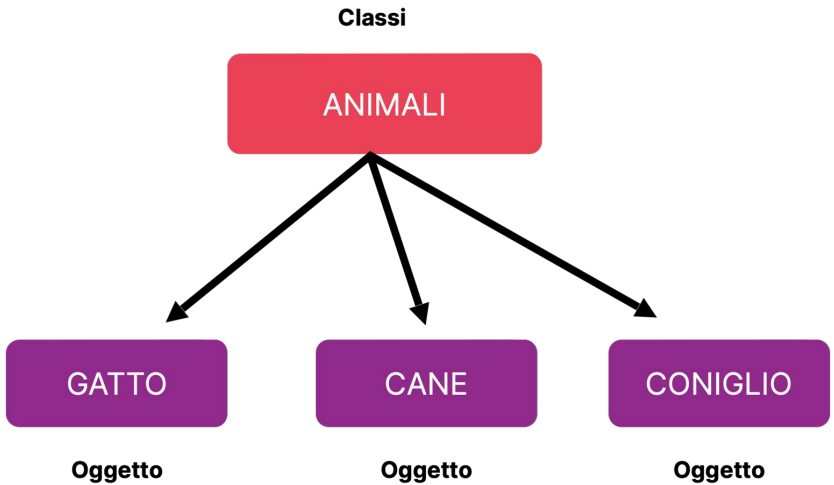
La **Programmazione Orientata agli Oggetti** è un paradigma di programmazione comune alla maggioranza dei linguaggi di programmazione in alternativa al paradigma procedurale/funzionale.

La OOP è una programmazione a struttura modulare basata sulla creazione di modelli generici (**classi**) dai quali è possibile generare (istanziare) degli **oggetti** che sono l'espressione concreta della classe.

Come paradigma è stato introdotto in JS con lo standard ECMA 2015 (ES6) con il supporto alle classi.

La classe in OOP è un vero e proprio template (modello) per l'oggetto.

Da una classe possiamo creare un numero indefinito di oggetti.



Per creare una classe utilizziamo la keyword class.

La classe contiene un metodo costruttore con la keyword constructor.

La classe è utilizzata attraverso gli oggetti che vengono creati:

var gatto = new Animali(“certosino”,”rosso”, 20);

oggetto = istanza della classe

```
class Animali{
constructor(specie, colore, velocità){
this.specie = specie;
this.colore = colore;
this.velocita = velocita
}
}
```

```
class Animali{
constructor(specie, colore,velocita){
this.specie = specie;
this.colore = colore;
this.velocita = velocita
}
corre(){
var distanza = 0;
return distanza * this.velocita;
}
}
```

L'ereditarietà è uno degli aspetti più importanti della OOP.

Possiamo modularizzare ancora di più le nostre applicazioni estendendo le caratteristiche di una classe in un'altra che la eredita.

Usando la keyword extends possiamo creare una classe. Da un'altra che è logicamente collegata.

Superclasse

```
class Animali{
constructor(specie){
this.specie = specie;
}
mostra(){
return this.specie;
}
}
```

Sottoclasse

```
class Mammiferi extends Animali{
constructor(specie, ambiente){
super(specie);
this.ambiente = ambiente;
}
visualizza(){
return this.mostra() + this.ambiente;
}
}
```

Un metodo si definisce statico quando è definito all'interno della classe e ne dipende strettamente.

Questo metodo è definito con la keyword static.

A differenza di altri metodi, non può essere chiamato nell'oggetto ma solo nella classe.
Per usare il metodo nell'oggetto bisogna passarlo come parametro.

```
class Animali{
constructor(specie){
this.specie = specie;
}
static mostra(){
return this.specie;
}
}

var gatto = new Animali("certosino");

gatto.mostra(); => NO
Animali.mostra(); => SI'
```

Programmazione asincrona

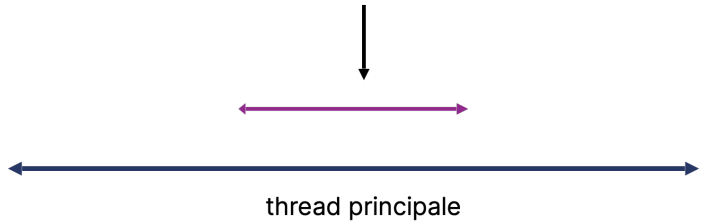
Nella programmazione sincrona le funzioni vengono eseguite una dopo l'altra, tanto che un processo non terminato blocca il successivo.

Al contrario nella programmazione asincrona una funzione viene eseguita parallelamente ad altri processi. In effetti, viene anche definita programmazione parallela.

Esempio:
Un bottone premuto dovrà svolgere, parallelamente al thread principale, una certa azione. Per esempio mostrare un testo.

Uno degli elementi fondamentali della programmazione asincrona è poter configurare degli avvisi che una certa azione parallela ha terminato il suo lavoro.

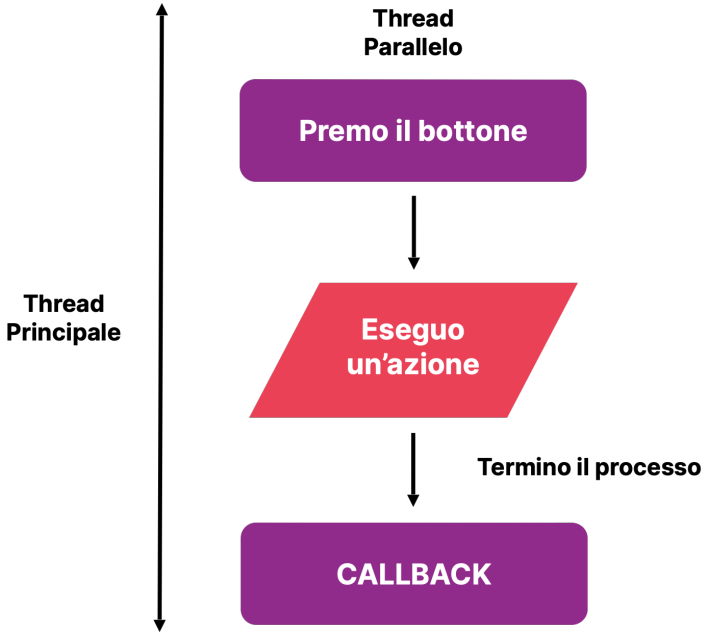
```
bottone.funzione(mostraTesto).chiudiBottone());
```



La funzione invocata dopo che la asincrona ha terminato il suo lavoro è detta di callback.

In pratica, si invoca una funzione come parametro di un'altra. Una funzione chiama un'altra funzione alla fine di un processo.

Le funzioni vengono eseguite nell'ordine in cui vengono invocate.



```
function mostraSomma(parametro){
alert(parametro);
}

function calcolaSomma(numero1, numero2){
var somma = numero1 + numero2;
return somma;
}

var risultato = calcolaSomma(6,7);
mostraSomma(risultato);
```

```
function mostraSomma(parametro){
alert(parametro);
}

function calcolaSomma(numero1, numero2){
        var somma = numero1 + numero2;
        mostraSomma(somma);
}


mostraSomma(risultato);
```

Le promises sono oggetti che permettono di gestire
Agevolmente callback annidate, cioè funzioni che richiamano funzioni che a loro volta richiamano altre funzioni...

Di fatto creano una coda di esecuzione di funzioni attraverso blocchi di codice configurati con la keyword then.

```
var calcolo = new Promise(successo, errore){
successo();
errore();
}
```

```
function miaFunzione(){
        funzioneAsincrona()
        .then(altraFunzione()) => promise
        .then(fineFunzioni()); => promise
}
```

Attributi di una promise:
Stato

- In attesa
- Soddisfatta
- Respinta

Risultato

- Indefinito
- Successo
- Errore

La keyword async configura una funzione che richiama una promise.

La keyword await configura una funzione che attende una promise. await si usa solo all'interno di una funzione async.

```
async mostraTesto() {
        return await funzioneAsincrona();
}
```

Quindi async imposta una funzione perchè lavori In modo asincrono generando un thread parallelo al Principale, mentre await imposta un'attesa fino al termine dell'esecuzione della funzione.

Gestione degli errori, strict mode e debugging

Blocco try / catch

In JavaScript possiamo fare un controllo della correttezza dell'esecuzione di istruzione e degli eventuali errori.

Struttura del blocco:

```
try(){
    //blocco di codice da eseguire e provare
}catch(messaggio - errore){
   //blocco di codice per gestire l'errore
}
```

```
try(){
   windowalert("Controllo errori");
}catch(errore){
   errore.message;
}
```

output => segnalazione di un errore su windowalert

Blocco try / catch

Messaggio errore custom

Con la keyword throw possiamo customizzare

Il messaggio di errore intercettato da catch().

Gli errori possono essere anche intesi quali eccezioni da controllare: throw exception.

```
…
try {
            if(x < 3)  throw “troppo basso";
            if(x > 15)   throw “troppo alto";
 }
 catch(err) {
            message.innerHTML = err;
 }
…
```

Uso di strict mode

Con la sintassi “strict mode”, modalità rigorosa, intendiamo una scrittura di JavaScript che impedisce alcuni errori soprattutto a carico della definizione delle variabili.

Esempio:

In JS non strict mode è ammessa questa sintassi:

```
numero = 7;
alert(numero);
```

Se uso la direttiva “use strict”;
Il codice precedente non sarà eseguito poiché non è possibile ammettere la variabile non definita.

La modalità di debug più usata e versatile in JS

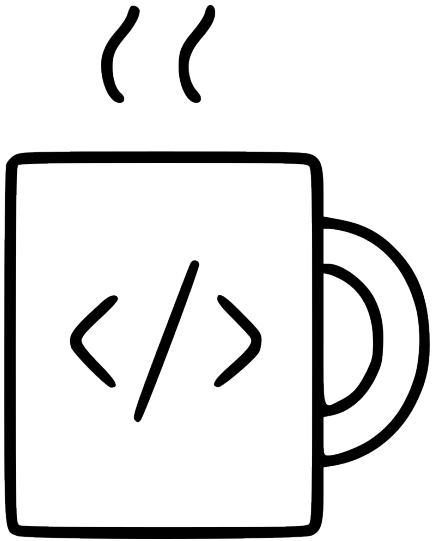
è la direttiva console.log() i cui messaggi a console

Permettono il controllo del codice senza influire

Nello script e nella sua visualizzazione.

Alcune direttive dell'oggetto console ci permettono di personalizzare i messaggi:

```
console.info("messaggio in stringa bianca");
console.error("messaggio in stringa rossa ");
console.warn("messaggio in stringa gialla");
```



PAUSA

Ci vediamo alle ore 11.32

Dom Traversing:
metodi ed elementi

Il DOM traversing è la possibilità di JavaScript di selezionare e manipolare strutture del Document object Model.

La selezione si serve di specifici metodi che intercettano l'elemento tramite:

- Id
- Classe
- Nome del tag
- Valore dell'attributo value
- Selezione di una o più proprietà CSS

Questi sono i principali metodi applicati all'oggetto Document la cui lista è particolarmente nutrita.

La manipolazione degli elementi avviene attraverso l'assegnazione di metodi all'elemento selezionato.

Struttura html

Selezione in base all'id con il metodo getElementById
document.getElementById("test");

Applicazione di una proprietà (innerHTML)
document.getElementById("test").innerHTML = "testo";

Metodi e proprietà per gli elementi HTML

Qualsiasi tipo di elemento HTML può essere selezionato e manipolato in JS.

Proprietà e metodi di JS possono essere applicati ad ogni elemento HTML.

Esempi

aggiungiamo un elemento ad una lista
document.createElement("*LI*");

Selezioniamo elementi secondo una classe html
document.getElementsByClassName(" *nomeClasse*");

Selezioniamo un elemento dal suo id e aggiungiamo una classe
document.getElementById(" *nomeID*").className = "nomeClasse";
Selezioniamo tutti gli elementi di una classe
querySelectorAll(". *nomeClasse* ");

JavaScript attraverso l'oggetto style può manipolare integralmente lo stile di un elemento HTML selezionato

```
document.getElementById("nomeId").style;
```

All'oggetto style può essere assegnata una proprietà CSS per una formattazione specifica

```
document.getElementById("nomeId").style.color = "green";
```

Esempi

```
querySelectorAll(".nomeClasse").style.color = "green";

document.getElementById("nomeId").style.fontSize = "large";

createElement("DIV").style.backgroundColor = "red";
```

challenge
NETWORK

shaping the skills of tomorrow

challengenetwork.it