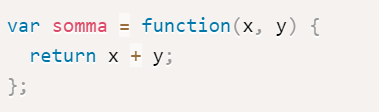
**29 Funzioni anonime**

In JavaScript una funzione è qualcosa in più: una **funzione JavaScript è in realtà un oggetto.**

A differenza degli altri oggetti JavaScript basati, come abbiamo visto, su Object, una funzione ha come oggetto base Function.

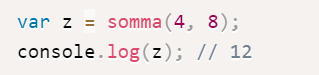


A differenza della definizione classica, però, in questo caso non abbiamo specificato un nome per la funzione.

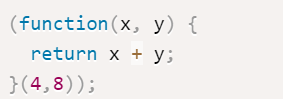
Una definizione di funzione che non specifica il nome è detta in genere funzione anonima o funzione letterale o funzione lambda.

Tuttavia, nessuno ci vieta di assegnare un nome ad una funzione anche quando questa viene assegnata ad una variabile.

Possiamo invocare una funzione associata ad una variabile accostando semplicemente al nome della variabile le parentesi tonde con gli eventuali parametri, come mostrato nel seguente esempio:



Più in generale, possiamo invocare una funzione utilizzando direttamente la sua versione anonima o letterale:

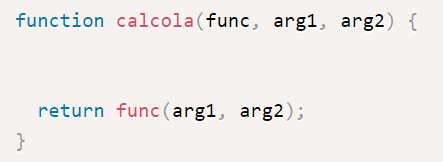


Facciamo notare che l’intera espressione è racchiusa tra parentesi tonde. Questo è necessario perchè altrimenti il parser JavaScript, trovando la parola chiave function, tenterebbe di interpretare l’espressione come una definizione di funzione nel contesto globale.

Questo tipo di espressione è in genere chiamata immediately-invoked function expression oppure self-executing anonymous function

***30 Callback***

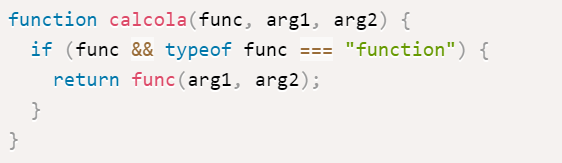
La funzione passata come parametro è detta generalmente funzione di callback o semplicemente **callback**.



La funzione calcola() prevede tre argomenti. Il primo argomento è una funzione di callback che viene invocata sul secondo e terzo argomento. In questo caso il valore restituito dalla funzione è il valore restituito dall’applicazione della funzione di callback sugli altri due argomenti.

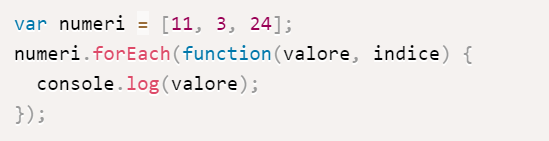


Un aspetto a cui prestare attenzione quando scriviamo funzioni che accettano callback è quello di accertarsi che venga passata effettivamente una funzione prima di invocarla. Il seguente esempio mostra come effettuare tale verifica:



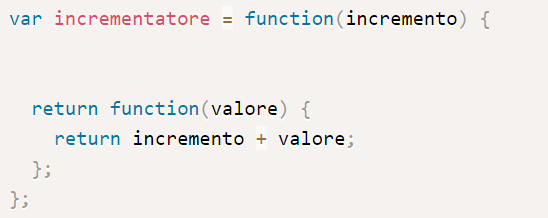
La condizione dell’if verifica che l’argomento func sia stato effettivamente passato, cioè che non sia null, e che il suo tipo sia function.

L’uso di funzioni di callback è molto frequente nella programmazione JavaScript. Ad esse si ricorre ad esempio, nell’esecuzione di alcuni metodi e funzioni predefinite:

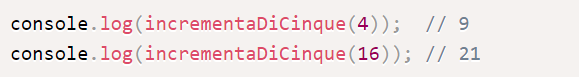


**31 funzioni che restituiscono funzioni**

Dal momento che una funzione è un oggetto è possibile restituirla come valore di ritorno dell’esecuzione di un’altra funzione.



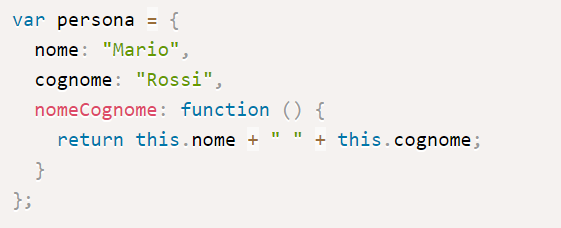




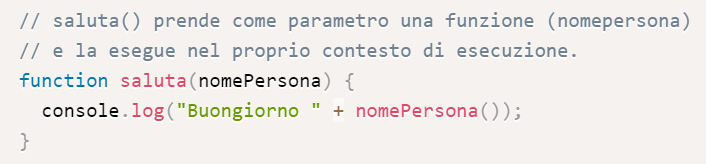
**32 L’oggetto this e il contesto di esecuzione**

Un contesto di esecuzione è composto dall’insieme delle variabili accessibili in un dato momento dell’esecuzione di un’istruzione, compresi gli eventuali argomenti di una funzione e l’oggetto this

Un classico esempio è la **corretta risoluzione dell’oggetto this**, che rappresenta l’oggetto a cui è associata una proprietà o un metodo.



Creiamo ora una funzione che prende per parametro il riferimento ad un altra funzione e tenta di lanciarla



… chiamando saluta(persona.nomeCognome) non invochiamo il metodo nomeCognome() ma lo passiamo alla funzione saluta(), all’interno della quale verrà poi invocato (console.log("Buongiorno " + nomePersona())).

Dal momento che nomeCognome() è invocato al di fuori del suo contesto di esecuzione, this non rappresenta più l’oggetto persona ma l’oggetto globale, nel quale non è presente alcuna definizione di variabili nome e cognome.

Il risultato che avremo è quindi:



Per risolvere il nostro problema possiamo fare ricorso a due metodi dell’oggetto funzione che ci consentono di specificare il significato che intendiamo associare alla parola chiave this

**call**()

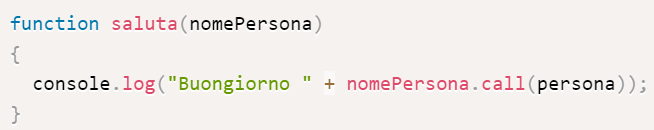


**apply**()



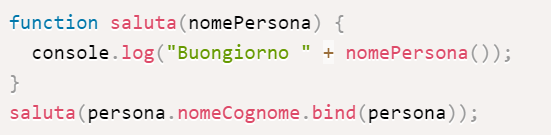
Il secondo metodo è **apply**(), del tutto simile a call() con la differenza che prevede due soli parametri: il primo è l’oggetto da associare a this mentre il secondo parametro è un array dei valori da passare alla funzione da invocare

Nel nostro caso possiamo quindi riscrivere la funzione saluta() in questo modo:



Un approccio alternativo all’utilizzo di call() e apply() per impostare correttamente l’oggetto this è l’uso del metodo **bind**(), introdotto nella versione 5 di ECMAScript.

Vediamo come sfruttare bind() nel nostro caso:



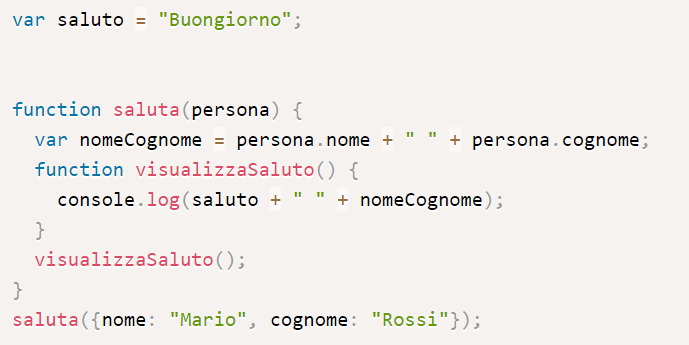
In questo caso, invece di intervenire sul codice della funzione saluta(), abbiamo passato ad essa come parametro non direttamente il metodo nomeCognome() ma una sua versione con this preimpostato su persona.

**33 Clousure e scope**

Tra i concetti fondamentali sulle funzioni abbiamo esaminato l’*ambito di visibilità* o **scope** delle variabili, distinguendo:

* *scope globale*, accessibilità estesa all’intero script;
* *scope locale*, accessibilità ristretta al solo codice di una funzione o di un blocco di codice.

Nella determinazione degli scope e quindi del contesto di esecuzione però intervengono altri fattori dipendenti dalla possibilità di definire funzioni all’interno di funzioni, di associare funzioni a variabili e di passarle o ottenerle da altre funzioni.



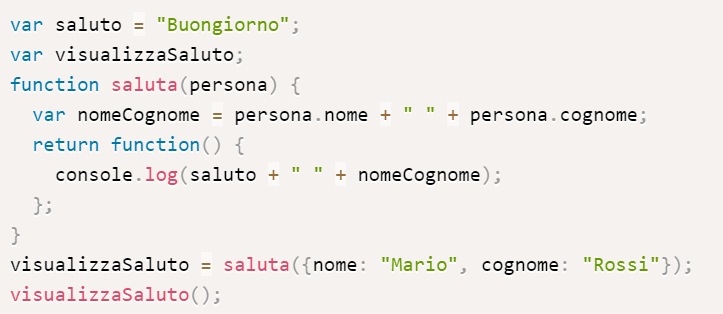
Già abbiamo visto il fatto che all’interno di una funzione è possibile fare riferimento ad una variabile globale, nel nostro caso la funzione visualizzaSaluto() accede alla variabile saluto definita globalmente

Un aspetto nuovo evidenziato dal codice è il fatto che la funzione visualizzaSaluto(), definita all’interno della funzione saluta(), possa accedere ad una variabile locale di quest’ultima, nello specifico nomeCognome.

Non si ha più, quindi, una semplice distinzione tra scope locale e globale, ma è possibile avere una vera e propria gerarchia di scope o **scope chain**. Una funzione, infatti, può accedere allo scope locale, allo scope globale ed allo scope accessibile dalla funzione in cui è stata definita (funzione esterna) , il quale può essere a sua volta il risultato della combinazione del proprio scope locale con lo scope della sua funzione esterna e così via.

La cosa ancora più interessante è il fatto che in JavaScript l’accesso allo scope della sua funzione esterna è consentito anche dopo che questa ha terminato la sua esecuzione.

Consideriamo il seguente esempio derivato dal codice precedente:



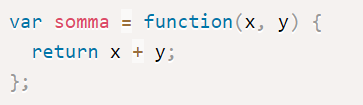
quando la funzione restituita viene invocata, la funzione saluta() (la sua funzione esterna) ha terminato la sua esecuzione e quindi il suo contesto di esecuzione non esiste più. Nonostante ciò è ancora possibile accedere alla variabile nomeCognome presente nel suo scope locale.

Il principio di base su cui si fonda questo meccanismo stabilisce che ogni variabile che era accessibile quando una funzione è stata definita rimane "racchiusa" nello scope accessibile dalla funzione. Questo meccanismo è detto **closure**.

**34 arrow function**

Si tratta di funzioni anonime con una sintassi molto concisa ed alcune specifiche caratteristiche che analizziamo in questa lezione.

Consideriamo ad esempio una funzione anonima che implementa la somma, come quella che segue:



Possiamo implementarla utilizzando la **sintassi delle arrow function** nel seguente modo:



La sua invocazione è del tutto identica a quella di una funzione anonima assegnata ad una variabile:



**varianti sintattiche di arrow function:**







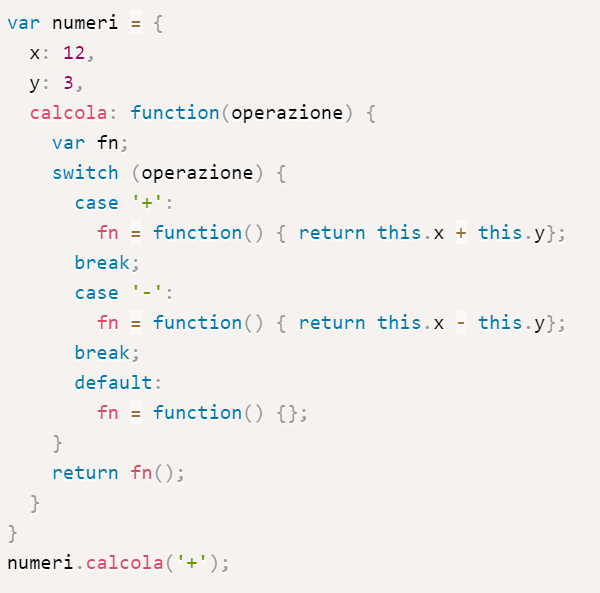


Data la sintassi compatta delle arrow function, esse si prestano molto bene ad essere utilizzate come callback

Ma le arrow function non introducono soltanto una diversa espressione sintattica nella definizione delle funzioni. Esse hanno anche delle implicazioni semantiche che occorre tener presente

L’aspetto più importante da tener presente è che la parola chiave **this all’interno di una arrow function** non rappresenta il contesto di esecuzione a runtime come avviene per le funzioni standard. Essa rappresenta sempre lo scope in cui la funzione stessa viene definita.

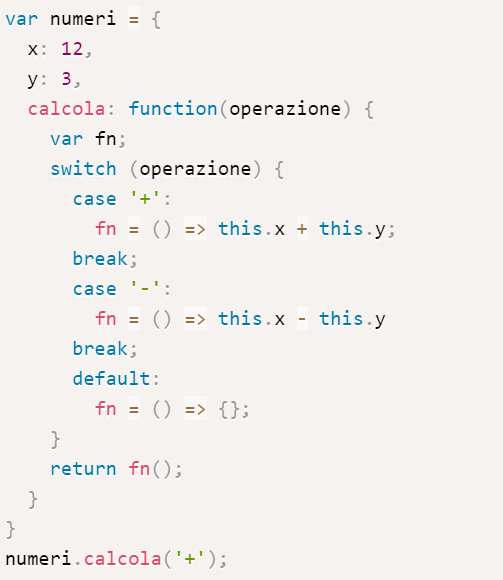
Proviamo a spiegare questo concetto con un esempio rappresentato dal seguente codice:



Il risultato dell’esecuzione del metodo non è, come a prima vista ci aspetteremmo, la somma dei due valori numerici, ma il valore NaN. Il motivo dovrebbe essere abbastanza chiaro: la parola chiave this utilizzata nelle funzioni anonime definite all’interno del metodo calcola() non sta indicando l’oggetto numeri, ma il loro contesto di esecuzione.

l’approccio classico per aggirare questo problema consiste nel definire una variabile all’interno dello scope del metodo calcola() ed assegnargli il valore di this.

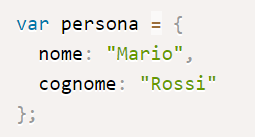
L’alternativa è il ricorso alle arrow function. Possiamo infatti riscrivere il codice precedente come mostrato di seguito per ovviare all’inconveniente:



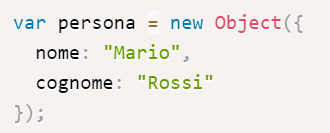
in questo caso this rappresenterà il contesto al momento della definizione delle funzioni, e cioè l’oggetto numeri stesso.

**35 Gli oggetti**

possiamo definire un oggetto nel seguente modo:



Oppure:



Possiamo aggiungere in un secondo tempo una nuova proprietà senza particolari formalità:



Possiamo anche assegnare ad un oggetto una proprietà con un nome che non segue le regole sintattiche per variabili e identificatori, come nel seguente esempio:

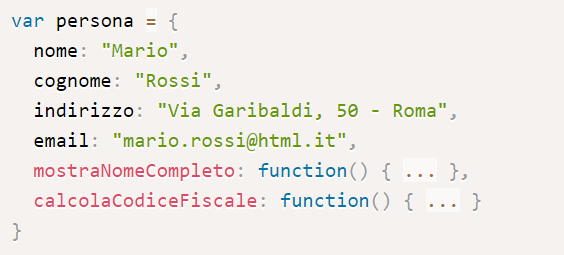


La definizione di un metodo infatti non è altro che l’assegnazione di una funzione ad una proprietà:



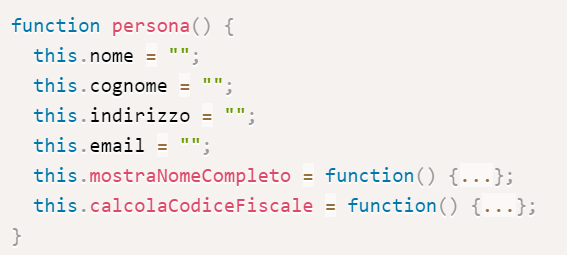
**36 Costruttori, creare oggetti in Javascript**

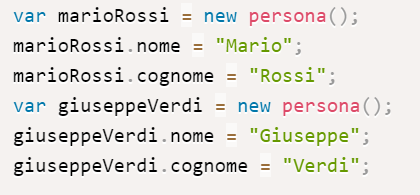
Immaginiamo di aver bisogno di più oggetti dello stesso tipo, ad esempio di più oggetti persona, che condividono la stessa struttura:



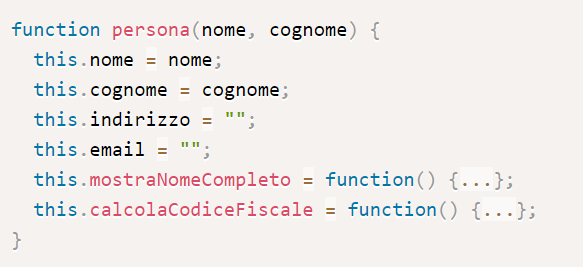
Utilizzando la notazione letterale saremmo costretti a ripetere la definizione per ciascun oggetto che vogliamo creare, cosa indispensabile per gli elementi identificativi della persona ma inutilmente ripetitiva per le parti costanti come ad esempio i metodi.

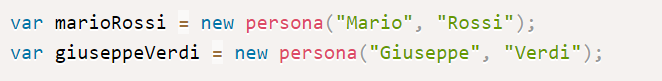
Per evitare quindi di dover ridefinire da zero oggetti che hanno la stessa struttura possiamo ricorrere ad un **costruttore**. Un costruttore non è altro che una normale funzione JavaScript invocata mediante l’operatore **new**. Vediamo ad esempio come creare un costruttore per l’oggetto persona:





nella definizione di un costruttore possiamo prevedere la presenza di parametri che possiamo utilizzare nell’inizializzazione del nostro oggetto.



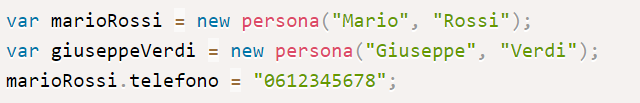


Per cercare di ovviare a dimenticanze accidentali dell’operatore new si può ricorrere ad una convenzione abbastanza diffusa che prevede l’**uso del maiuscolo per la prima lettera del nome di un costruttore**.



**37 Prototipi ed ereditarietà**

Nella creazione di oggetti possiamo partire da una struttura comune definita da un costruttore per poi personalizzarla in base alle nostre esigenze.



Ma come fare per modificare la struttura di tutti gli oggetti creati tramite un costruttore?

possiamo sfruttare una delle carattersitiche più interessanti della programmazione ad oggetti di JavaScript: il **prototype**.



Questo assegnamento fa sì che tutti gli oggetti creati tramite il costruttore persona() abbiano istantaneamente tra le loro proprietà anche la proprietà telefono valorizzata al valore indicato.

In JavaScript, il prototipo di un oggetto è una sorta di riferimento ad un altro oggetto. Gli oggetti che creiamo tramite la semplice applicazione della notazione letterale hanno come prototipo Object.

Quando creiamo un oggetto tramite un costruttore, il suo prototipo è l’oggetto prototype del costruttore.

Il meccanismo su cui si basa l’**erediterietà prototipale** (*prototypal inheritance*) di JavaScript è abbastanza semplice: se una proprietà non si trova in un oggetto viene cercata nel suo prototipo.

Il prototipo di un oggetto può a sua volta avere un altro prototipo. In questo caso la ricerca di una proprietà o di un metodo risale la catena dei prototipi fino ad arrivare all’oggetto Object, il prototipo base di tutti gli oggetti.

38 Creare oggetti con Object.create()

Una alternativa ai metodi visti finora per la creazione di oggetti è quella di fare ricorso al metodo **create()** dell’oggetto base Object.

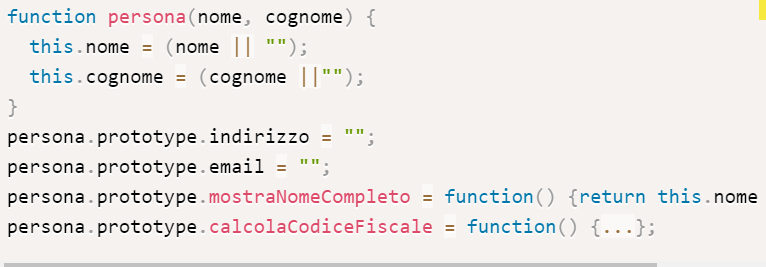
Nella sua forma minima il metodo create() consente di creare un oggetto specificando il prototipo di riferimento:



In questo esempio abbiamo specificato null come prototipo di riferimento dell’oggetto persona che abbiamo creato. Il risultato è un oggetto senza proprietà né metodi, nemmeno quelli ereditati dall’oggetto base Object, come avremmo ottenuto creando un oggetto vuoto tramite notazione letterale ({}). Per ottenere un oggetto equivalente all’oggetto vuoto dobbiamo scrivere il seguente codice:



Riconsideriamo quindi la nostra definizione dell’oggetto persona definendo opportunamente il suo prototipo:





Naturalmente in questo caso abbiamo creato un oggetto senza utilizzare il costruttore e quindi non abbiamo impostato direttamente le due proprietà nome e cognome. Il fatto è che, avendo tenuto le due proprietà fuori dalla definizione del prototipo, esse non sono presenti nella struttura dell’oggetto marioRossi. Possiamo rendercene conto invocando il metodo mostraNomeCompleto(), che restituirà un undefined undefined invece che una stringa vuota come ci aspetteremmo.

L’**inizializzazione di un oggetto** creato con Object.create() avviene tramite il secondo parametro opzionale del metodo:

