Domande di teoria da vecchi esami

A cura di Giorgia Bosello.

Descrivere l'utilizzo di una funzione come metodo. In particolare indicando come si aggiunge un metodo ad un oggetto, come si invoca ed il ruolo dell'oggetto `this`.

In JavaScript, le funzioni sono oggetti che possono essere senza nome, oppure associate ad una variabile, come ad esempio:

```
let prova = function () {
    // do something
}
```

Un oggetto può creare metodi semplicemente aggiungendo una funzione alle sue properties, ad esempio, alla creazione di un oggetto (che ricordiamo può essere fatta in più modi:

1. `let obj = {}` aggiungendo properties "on the fly" semplicemente scrivendo il nome dell'oggetto appena creato la proprietà desiderata. Ad esempio, se vogliamo aggiungere ad un oggetto `car` la proprietà *cilindrata* a *1050*, faremo semplicemente `car.cilindrata = 1050`.
2. `let obj = Object.create()` passando a `create()` i parametri necessari o semplicemente lasciando vuoto.
3. Attraverso un costruttore tramite una funzione:

   ```
   function car () {
       this.cilindrata = 1050;
       this.cavalli = 200;
   }
   ```

   e poi facendo la chiamata `let obj = new car();`

) basta fare

```
obj.funct = function () {
    // do something
}
```

L'oggetto `this` è una variabile dinamica che cambia in base all'oggetto che sta chiamando il metodo e può, quindi, assumere "proprietari" diversi. Per esempio:

```
let person = {
    name: 'Tizio',
    age: 30,

    set age(newAge) {
        this.age = newAge; //this.age riferisce alla proprietà 'age' dell'oggetto 'person'.
    }
}
```

Descrivere l'utilizzo di una funzione come costruttore ed il ruolo del campo prototype

L'invocazione di un costruttore crea un nuovo oggetto vuoto che eredita le proprietà dal prototype del costruttore. Le funzioni costruttore servono per inizializzare oggetti e questo nuovo oggetto creato è usato nel contesto dell'invocazione, in modo che la funzione costruttore possa riferirsi con la keyboard `this`.

Solitamente, una funzione usata come costruttore non ha un valore di ritorno, tipicamente inizializzano un nuovo oggetto e lo ritornano implicitamente. Se, comunque, un costruttore utilizza un `return` per ritornare un oggetto, allora quell'oggetto diventa il valore di invocazione dell'espressione. Se il costruttore ritorna un valore vuoto o un valore primitivo, allora il `return` è ignorato e il nuovo oggetto viene usato come valore dell'invocazione.

Quando una funzione è usata come costruttore, il nuovo oggetto creato eredita le proprietà dall'oggetto prototype.

Spiegare il meccanismo di accesso in lettura e scrittura ai campi di un oggetto, con particolare riferimento al ruolo del prototype.

Per accedere alle proprietà di un oggetto si usano quelli che sono noti come *accessor properties*, ovvero i metodi `get` e `set`.

Quando un programma richiede il valore ad un accessor property, JavaScript invoca il metodo `get` (senza passargli argomenti). Il valore di ritorno di questo metodo diventa il valore dell'accessor property.

Quando un programma imposta il valore di un accessor property, JavaScript invoca il metodo `set`, passndogli il valore a destra dell'assegnamento. Il valore di ritorno del metodo `set` è ignorato.

Se un oggetto ha entrambi i metodi `get` e `set`, allora ha accesso in lettura e scrittura.

Il prototipo di un oggetto è la referenza ad un altro oggetto dal quale le proprietà sono ereditate.

Per esempio:

```
let person = {
    name: 'Tizio',
    age: 30,

    set age(newAge) {
        this.age = newAge;
    },

    get age() {
        return this.age;
    }
};
```

Spiegare il concetto di scope, come funzione e che cos'è una catena di scope e perchè è rilevante nell'ambito di definizione e chiamata di funzione.

Lo *scope* è l'idea di programmazione in cui alcune variabili sono accessibili/non accessibili da altre parti del programma.

Lo scope può essere *globale* o *locale*. Una variabile creata nel *global scope* è accessibile da ogni parte del programma, mentre una variabile creata nel *local scope* è accessibile solo dal blocco in cui è stata creata. Anche i parametri delle funzioni contano come variabili locali e sono definite solo all'interno del corpo della funzione.

Lo *scope* di una variabile è la regione del programma in cui essa è stata definita.

Per esempio:

```
let scope = "global"; // Declare a global variable
function checkscope() {
    let scope = "local"; // Declare a local variable
    return scope; // Return the local value, not the global one
}

checkscope(); // "local"
console.log(scope); // "global"
```

Spiegare il concetto di chiusura ed il suo ruolo nel definire lo scope di una funzione della sua dichiarazione

Le funzioni sono eseguite usando la variabile scope che era in uso quando sono state definite, non la variale scope che è in uso quando sono state invocate. Per poter implementare il *"lexical scoping"*, l'oggetto delle funzioni in JavaScript deve includere non solo il codice della funzione, ma anche una referenza alla corrente **catena di scope**.

Questa combinazione dell'oggetto di una funzione e uno scope (legami con le variabili) nel quale le variabili della funzione sono risolte, è chiamato **chiusura** nella "letteratura informatica".

Tecnicamente, tutte le funzioni in JavaScript sono *chiusure*: sono oggetti e hanno una catena di scope associata con loro. Molte funzioni sono invocate usando la stessa catena di scope che era in uso quando la funzione era stata definita e non importa che fosse coinvolta una chiusura.

Per esempio:

```
let scope = "global scope";           // A global variable
function checkscope() {
    let scope = "local scope";        // A local variable
    function f() { return scope; }    // Return the value in scope here
    return f();
}
checkscope()                          // => "local scope"

let scope = "global scope";           // A global variable
function checkscope() {
    let scope = "local scope";        // A local variable
    function f() { return scope; }    // Return the value in scope here
    return f();
}
checkscope()()                        // What does this return?
```

La funzione nidificata `f()` era stata definita sotto una catena di cope nel quale la variabile scope era legata al valore "local scope". Quel legame è ancora in effetto quando `f` è eseguita, da qualsiasi parte sia eseguita. Quindi l'ultima linea di codice sopra ritorna "local scope", non "global scope".

Che cos'è il duck-typing

Per duck-typing si intende quella programmazione non convenzionale per cui si tende a guardare cosa un oggetto fa (che metodi ha) piuttosto di quale sia la sua classe. Questo tipo di approccio è comune in linguaggi come Python e Ruby ed è rappresentato da questa frase: *Quando vedo un uccello che cammina come un'anatra, nuota come un'anatra e starnazza come un'anatra, chiamo quell'uccello Anatra.* In JavaScript interpretiamo questo aforismo come "se un oggetto può camminare, nuotare e starnazzare come un'Anatra, allora lo possiamo trattare come un'Anatra, anche se non eredita il prototipo oggetto della classe Anatra."

Differenze fra SVG e Canvas

Entrambi sono elementi introdotti con il "nuovo" HTML5, ed entrambi permettono di "disegnare" nella nostra pagina web. Una differenza sostanziale è che il canvas ci permette di disegnare immagini bitmap, mentre gli SVG sono immagini vettoriali. Le immagini bitmap sgranano se se ne modificano le dimensioni, mentre le immagini vettoriali no.

CANVAS	SVG
Buon rapporto tra i browser più importanti	Non sempre supportato da tutti i browser
Manipolazione dei pixel delle immagini più approfondito	Ottimo per manipolare immagini vettoriale
Difficile gestire gli eventi	Gestione degli eventi più semplice
Più performante	Meno efficiente
Difficile da usare se non usi librerie JS apposite	È possibile disegnare e gestire immagini complesse grazie anche a prodotti di larga diffusione (Adobe Illustrator, ad esempio)

Costrutto for/in in JavaScript

Il costrutto for/in in javascript è uno dei modi per eseguire un ciclo. Il codice si presenta così:

```
for (<variabile> in <oggetto>) {
    //istruzioni
```

```
}
```

È molto utile per conoscere le proprietà di un oggetto, è possibile anche scorrere un array:

```
for (index in array) {
    array[index]
}
```

Esempio:

```
for(var i = 0; i < a.length; i++) // Assign array indexes o variable i
    console.log(a[i]);  // Print the value of each array element

for(var p in o) // Assign property names of o to variable p
    console.log(o[p]); // Print the value of each property
```

È diverso da un ciclo for, in quanto un ciclo for/in, se valuta **null** o **undefined** salta il loop e va al prossimo statement.

Descrivere lo scopo ed il funzionamento generale della funzione $ della libreria jQuery.

La libreria jQuery è una liberia che va inclusa nel file di programmazione JavaScript. Essa permette di semplificare la vita al programmatore con semplici righe di codice. Permette la manipolazione del DOM, dal CSS all'HTML in maniera semplice ed esaustiva, oltre a fornire metodi per le chiamate AJAX. Scrivere jQuery o scrivere `$` nelle chiamate di funzione è la stessa cosa. È un modo più veloce di scrivere `jQuery()`. jQuery può essere eseguito in modalità compatibilità se un'altra liberia sta usando `$`, basta chiamare la funzione `jQuery.noConflict();`. `$` è anche comunemente usato come selettore di funzione in JS.

In jQuery, la funzione `$` fa molto più che selezionare "cose": * Può essere passata a un selettore per ottenere una collection di elementi dal DOM. * Può essere passata ad una funzione da eseguire quando il documento è pronto (simile a `body.onload()`, ma meglio). * Può essere passata ad un documento o elementi del DOM che si vogliono raggruppare in un oggetto jQuery.

Esempi:

```
let nodes = $("selector"); // come ad esempio '#prova'
let nodes = $(element); // con element un elemento del DOM
let nodes = $(HTML text);
$(function() {
    //do something
});
```

Una volta ottenuti i vari nodes è possibile utilizzare una vasta quantità di metodi resi disponibili dalla libreria, come append(), replaceWith() e tanti altri.

Spiegare il funzionamento del metodo `getElementsByTagName` di document. Qual è la differenza fra il risultato ottenuto con `document.getElementsByTagName("p")` e con `document.querySelectorAll("p")`?

Il metodo `getElementsByTagName` crea un array di sola lettura con all'interno tutti gli oggetti basati dal loro tag name. `querySelectorAll` può usare qualsiasi selettore e ciò da molta più flessibilità e potere, ma è più nuovo ed ha un minor supporto dei browser. `getElementsByTagName` è probabilmente più veloce, dato che è più semplice. Non vi è alcuna differenza fra il risultato ottenuto dalla chiamata `document.getElementsByTagName("p")` e dalla chiamata `document.querySelectorAll("p")`

Per esempio:

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8">
    <title>Yandex</title>
</head>
<body>
    <a href="((http://unsitoqualunque.it))">Boh1</a>,
    <a href="((http://unsitoqualunque.com))">Boh2</a>
</body>
```

```
<script>

let elems1 = document.getElementsByTagName('a'), // return 2 elements, elems1.length = 2
    elems2 = document.querySelectorAll("a");  // return 2 elements, elems2.length = 2

document.body.appendChild(document.createElement("a"));

console.log(elems1.length, elems2.length);  // now elems1.length = 3!
                                            // while elems2.length = 2
</script>
</html>
```

Perchè querySelectorAll ritorna elementi *non-live*, ovvero, se il documento DOM subisce delle modifiche, la NodeList non viene aggiornata. Nel nostro caso, non vi è alcuna differenza fra il risultato ottenuto dalla chiamata `document.getElementsByTagName("p")` e dalla chiamata `document.querySelectorAll("p")`; entrambi ritornano un array di paragrafi.

Nota:

- `document.getElementsByClassName()` is an HTMLCollection, and is live.
- `document.getElementsByTagName()` is an HTMLCollection, and is live.
- `document.getElementsByName()` is a NodeList and is live.
- `document.querySelectorAll()` is a NodeList and is *not* live.

HTMLCollections appear to always be live.