IL PICCOLO LIBRO DI
{ JAVASCRIPT }

Tutto quello che dovresti sapere sul linguaggio JavaScript
(ma non hai mai osato chiedere)

{ Valentino Gagliardi }

Il piccolo libro di JavaScript

Valentino Gagliardi

Questo libro è in vendita presso https://leanpub.com/piccolo-javascript-05b687d204abda5e

Questa versione è stata pubblicata il 2026-07-27

Questo è un libro di Leanpub. Leanpub permette ad autori ed editori un processo di pubblicazione agile. La Pubblicazione Agile consite nel pubblicare un ebook in corso d'opera, utilizzando strumenti leggeri e molte iterazioni per ottenere un feedback dai lettori, al fine di assicurare un libro giusto e attraente una volta completato.

Twitta questo libro!

Per favore, aiuta Valentino Gagliardi a diffondere la voce su questo libro su Twitter!

Il tweet consigliato per questo libro è:

Sto studiando JavaScript con il piccolo libro di JavaScript di @gagliardi_vale

L'hashtag suggerito per questo libro è #PiccoloLibroJavaScript.

Scopri cosa dicono gli altri su questo libro cliccando su questo link per cercare questo hashtag su Twitter:

#PiccoloLibroJavaScript

A Caterina, che mi ha spinto ad arrivare dove non avrei mai immaginato. A mio nonno Valentino che mi ha insegnato l'umiltà.

Indice

Capitolo 5. Tutto è un oggetto (o quasi)

JavaScript e l'illusione delle classi

JavaScript è un linguaggio di programmazione basato sulle classi? A giudicare da questo codice sembrerebbe di si:

```
class Person {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log("Hello " + this.name);
  }
}
```

La sintassi è molto simile a quella delle classi in altri linguaggi di programmazione. Come Python:

```
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        return 'Hello' + self.name
```

O come PHP:

```
class Person {
    public $name;

    public function __construct($name){
        $this->name = $name;
    }

    public function greet(){
        echo 'Hello ' . $this->name;
    }
}
```

Tuttavia la keyword `class` in JavaScript è una novità del linguaggio introdotta con ES6 (2015). In JavaScript infatti non esistono classi vere e proprie. Questo fatto pone un interrogativo: qual è il costrutto primordiale del linguaggio? In Python ad esempio tutte le classi ereditano da una classe `object`. C'è qualcosa di simile in JavaScript ma `Object` in JavaScript è una semplice struttura dati di tipo chiave/valore:

```
var obj = { chiave: 'valore' };
```

Nel capitolo 2 abbiamo introdotto i tipi di dato fondamentali di JavaScript, le primitive. Nell'elenco non figura `Object` che è un tipo a sè e `Function` che è un oggetto anch'esso, con la caratteristica però di poter essere chiamato come una funzione. Gli oggetti JavaScript sono l'equivalente di un dizionario Python o di una hash table in Perl. Come vedi non si tratta di classi eppure molti degli elementi di JavaScript sembrano discendere da `Object`. Come gli array ad esempio. Crea un array nella console del browser:

```
var arr = [1,2,3,4,5]
```

e controlla di cosa si tratta con `typeof`:

```
typeof arr
"object"
```

Anche le funzioni in JavaScript sono oggetti. Se creiamo una funzione e la ispezioniamo nella console del browser noteremo che ha dei metodi:

```
var a = function(){ return false; }
a.toString();
```

Output:

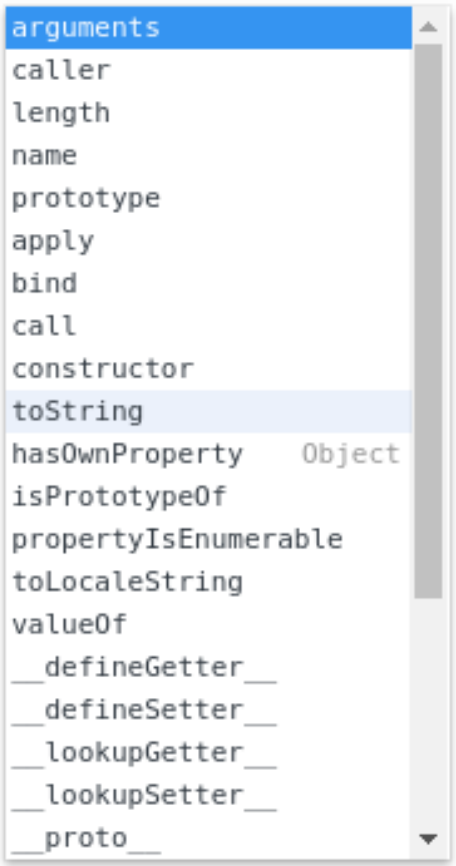
```
"function(){ return false; }"
```

Da dove viene fuori il metodo `.toString()`? Anche `Object` ha un metodo chiamato `.toString()`: per qualche strano motivo la nostra funzione ha gli stessi metodi di `Object`.

```
Object.toString();
```

La cosa si fa ancora più strana se guardiamo a quanti metodi e proprietà sono collegati alla nostra funzione:

```
> var a = function(){ return false; }
<- undefined
> a.arguments
```



I metodi "ereditati" dalla nostra funzione

Chi ha aggiunto questi metodi? Abbiamo detto che anche le funzioni in JavaScript sono oggetti e questo può darci qualche indizio. Forse `Function` è un lontano parente di `Object`? `Object` difatti è il padre di tutti gli oggetti in JavaScript. Ma per il momento accontentati di questo assaggio e ritorniamo alle classi. Occhio all'immagine sopra: tra le varie proprietà quella che dovrebbe attirare di più la tua attenzione è `prototype`.

TIP

Il metodo `.toString` dell'oggetto function non è la stessa versione di `Object.toString()`. `Function` infatti lo sovrascrive con una versione custom.

Abbiamo detto che JavaScript non è un linguaggio OOP. Ma abbiamo prove per dimostrarlo? Riprendiamo in mano la nostra classe Person:

```
class Person {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log("Hello " + this.name);
  }
}
```

Se la keyword `class` è stata aggiunta al linguaggio solo nel 2015, come abbiamo fatto finora a creare blocchi di funzionalità in JavaScript, organizzando ad esempio un oggetto `Person` con un metodo `greet()`? Per incapsulare gli oggetti e le funzionalità legate a ogni oggetto non facevamo altro che usare funzioni. Supponiamo di voler creare un oggetto `Person` e questo oggetto debba avere un metodo associato. La soluzione che potrebbe venire in mente di getto è:

```
function Person(name) {
  var newPerson = {};
  newPerson.name = name;
  newPerson.greet = function() {
    console.log("Hello " + newPerson.name);
  };
  return newPerson;
}
```

Prenditi qualche minuto per esaminare il codice e comprenderne il risultato di ritorno. La funzione `Person` crea un nuovo oggetto denominato `newPerson`. Assegna la proprietà `name` al nuovo oggetto insieme a un metodo denominato `greet()` che opera su `newPerson`. Per usare il nuovo oggetto potremmo chiamare la funzione `Person` in questo modo:

```
var me = Person("Valentino");
```

Questo ci consente di avere accesso al metodo `greet()`:

```
me.greet();
```

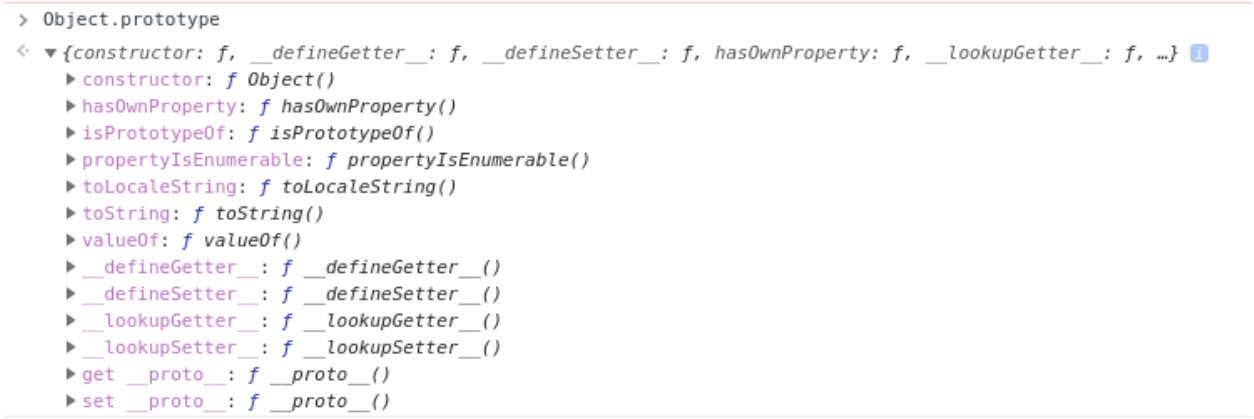
La soluzione sopra è ingenua ma non ottimale (vedremo dopo come renderla performante). Per il momento però il codice ci permette di svelare il mistero di `class`: non esistono classi in JavaScript. E' con le funzioni che in JavaScript possiamo generare nuovi oggetti con dei metodi associati. La keyword `class` è un'illusione. Ma qual è il problema nel codice visto sopra? Continua a tenere a mente `prototype` (che abbiamo visto prima), ci servirà a breve. Per come il motore JavaScript tiene in memoria variabili e funzioni (vedi capitolo 3 "Come funzionano i motori JavaScript?") viene logico immaginare che per 100 nuovi oggetti `Person` il motore dovrà copiare 100 volte il metodo `greet()` su ognuno. Ricorda che JavaScript gira nei browser, dobbiamo fare molta attenzione a come utilizziamo la memoria. C'è una soluzione migliore per creare copie di oggetti e metodi?

JavaScript e Object.prototype

Immagina che il nostro programma dovrà creare 200 oggetti `Person` (stiamo sviluppando una chat e si collegheranno molte persone). Con questa soluzione il motore JavaScript è costretto a creare 200 copie di `greet()`:

```
function Person(name) {
  var newPerson = {};
  newPerson.name = name;
  newPerson.greet = function() {
    console.log("Hello " + newPerson.name);
  };
  return newPerson;
}
```

Improponibile. C'è un modo per dichiarare il metodo `greet()` solo una volta all'interno del nostro programma? `prototype` ci viene in aiuto. Questo oggetto, collegato automaticamente di default a qualsiasi altro oggetto JavaScript contiene a sua volta diversi metodi:



I metodi contenuti all'interno di Object.prototype

Nella figura vediamo `Object.prototype`, scelto non a caso. `Object` infatti è il genitore di molti altri oggetti JavaScript. Perché `prototype` è così importante in JavaScript? Piuttosto che creare 200 copie di `greet()` come nel seguente esempio:

```
function Person(name) {
  var newPerson = {};
  newPerson.name = name;
  newPerson.greet = function() {
    console.log("Hello " + newPerson.name);
  };
  return newPerson;
}
```

possiamo dichiarare il metodo `greet()` una sola volta in un oggetto a parte (abbiamo già visto a inizio libro che gli oggetti JavaScript possono contenere funzioni):

```
var personMethods = {
  // il metodo greet ora è indipendente
  greet: function() {
    console.log("Hello " + this.name);
  }
};

function Person(name) {
  var newPerson = {};
  newPerson.name = name;
  // il metodo greet ora è indipendente
  // ma come colleghiamo newPerson a personMethods?
  return newPerson;
}
```

Nota l'utilizzo di `this.name` sopra: nel prossimo capitolo vedremo di cosa si tratta. Per il momento ti basti sapere che è necessario affinchè `greet()` punti su `newPerson`. Ma resta un dubbio: come colleghiamo l'oggetto `newPerson` a `personMethods`? Prima abbiamo appurato che le funzioni JavaScript, così come gli array, sono dei particolari tipi di oggetto che in qualche modo rimangono collegati al genitore `Object`. Possiamo sfruttare questa caratteristica di JavaScript per collegare `newPerson` a `personMethods`? Ci sono vari modi per creare un nuovo oggetto in JavaScript tra cui la forma letterale, quella più comune:

```
var newPerson = {};
```

Ma esiste anche un altra forma, ed è quella che ci consente di sfruttare la natura *prototipale* di JavaScript:

```
var newPerson = Object.create();
```

`Object.create()` è un metodo nativo di JavaScript, e crea per noi un nuovo oggetto: questo nuovo oggetto può essere legato a un genitore che passeremo come parametro durante la creazione. Nel nostro caso vogliamo che `personMethods` sia il genitore di `newPerson`:

```
var personMethods = {
  greet: function() {
    console.log("Hello " + this.name);
  }
};

function Person(name) {
  // ora newPerson è collegato a personMethods
  var newPerson = Object.create(personMethods);
  newPerson.name = name;
  return newPerson;
}
```

Per usare il nuovo oggetto chiamiamo la funzione `Person` come prima:

```
var she = Person("Caterina");
```

A questo punto abbiamo accesso al metodo `greet()`:

```
she.greet();
```

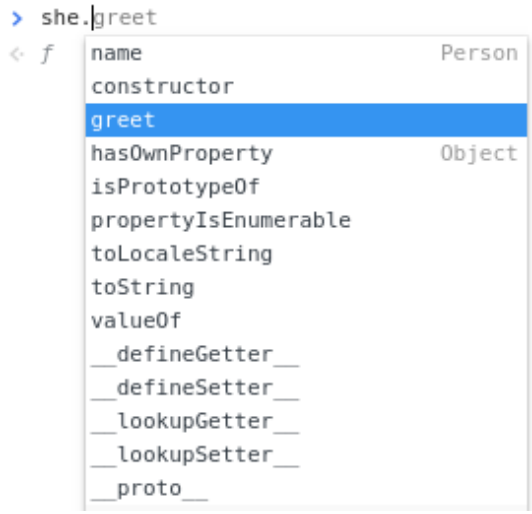
Come vedi non esiste nessun `greet()` sul nuovo oggetto `she`. L'unico oggetto in cui abbiamo dichiarato `greet()` è `personMethods`. Ma in qualche modo il motore JavaScript è in grado di risalire la catena dei prototipi, come si dice in gergo. Passo passo il motore JavaScript fa un'operazione di ricerca:

STEP 1: cerca il metodo `greet()` all'interno dell'oggetto `she` ma non lo trova e quindi inizia a risalire la catena dei prototipi.

STEP 2: individua il metodo `greet()` sull'antenato di `she`, ovvero sull'oggetto `personMethods` (che è il costruttore di `she`).

STEP 3: esegue il metodo `greet()` sull'oggetto `she`.

Possiamo confermare le nostre supposizioni esaminando l'oggetto `she` nel browser (copia nella console tutto il codice qui sopra e crea un nuovo oggetto come ho fatto io):



L'oggetto she

Nota come l'oggetto `she` "eredita" il metodo `greet()` che abbiamo specificato su `personMethods`. Ci sono anche altri modi per analizzare il genitore di un particolare oggetto, come `Object.getPrototypeOf()`:

```
Object.getPrototypeOf(she);
```

che ritorna nel nostro caso `{ greet: ƒ }`. O ancora, con `__proto__`:

```
she.__proto__
```

che ritorna sempre `{ greet: ƒ }`. `greet()` si riferisce al metodo che abbiamo definito all'interno dell'oggetto `personMethods`.

`__proto__` oppure `getPrototypeOf`?

`__proto__` è un sinonimo per `getPrototypeOf()` ed è stato standardizzato solo nel 2015 all'interno di ES6. In ogni caso è sempre preferibile evitarlo perché non tutti i browser sono d'accordo su questa convenzione.

Ricorda, la chiave di tutto è `Object.create()` e fra poco vedremo come ottimizzare ancora il nostro codice per arrivare alla forma standard e più concisa: quella delle funzioni *costruttore*.

Che differenza c'è tra metodi e funzioni?

All'inizio del libro abbiamo definito le funzioni come blocchi di codice riutilizzabili e stand-alone, non legati ad alcun oggetto. Per metodo si intende invece una funzione che opera su e per un determinato oggetto JavaScript.

JavaScript e la keyword new

Per capire a cosa serve la keyword `new` in JavaScript dobbiamo partire di nuovo da zero. Mettiamoci nei panni di una sviluppatrice JavaScript alle prime armi: Laura. Laura programma in C++ ma si ritrova coinvolta in un nuovo progetto, questa volta scritto in JavaScript. Ha poco tempo per studiare il linguaggio e ogni giorno deve produrre codice, anche se al primo colpo non sarà l'implementazione perfetta. Laura sa che in JavaScript non ci sono classi quindi evita di usare la keyword `class`, vuole vederci chiaro. Laura inizia a farsi alcune domande. Inizia a studiare la documentazione e scopre che le classi in JavaScript non sono altro che funzioni. Con queste informazioni in mano inizia a programmare in JavaScript le sue prime funzioni costruttore, ovvero funzioni che ritornano oggetti e metodi. Un giorno viene incaricata di creare una nuova funzionalità per l'applicazione: una classe che crea persone, completa di metodi per modificare il nickname. Facile! Una prima soluzione potrebbe essere questa:

```
function Person(name, nick) {
  var newPerson = {};
  newPerson.name = name;
  newPerson.nick = nick;
  newPerson.changeNick = function(nick) {
    newPerson.nick = nick;
  };
  return newPerson;
}
```

Il codice sembra funzionare come dovrebbe:

```
var laura = Person("Laura", "javaScripter89");
console.log(laura.nick);
"javaScripter89"
```

Ma presto vengono fuori i problemi. Sembra che i primi test sull'applicazione dimostrano che alla soglia dei 100 utenti (100 oggetti `Person`) il browser diventa lentissimo e va in crash. JavaScript è

un linguaggio per i browser: dobbiamo prestare molta attenzione alle performance. Dopo una prima analisi con il team il responso è chiaro: la funzione `Person` non è ottimizzata. Per ogni nuovo utente il motore JavaScript deve creare un nuovo oggetto. Oltre all'oggetto il motore è costretto anche a creare il metodo `changeNick()` 100 volte. Come possiamo rendere questo codice più performante? Laura tenta la strada più logica alla ricerca di una soluzione: separare il metodo `changeNick()` dalla funzione `Person`. Laura scopre infatti che può racchiudere tutti i metodi all'interno di un oggetto JavaScript. Questo oggetto JavaScript può rimanere collegato a tutti gli oggetti di tipo `Person` in modo che questi ultimi abbiano accesso ai metodi. Ecco come sarebbe un primo refactoring del codice. Creiamo prima di tutto un oggetto per contenere i metodi:

```
var personMethods = {
  changeNick: function(nick) {
    this.nick = nick;
  }
};
```

e poi facciamo qualche piccola modifica alla funzione `Person`:

```
function Person(name, nick) {
  var newPerson = Object.create(personMethods);
  newPerson.name = name;
  newPerson.nick = nick;
  return newPerson;
}
```

Il codice sembra molto più pulito. Ma la cosa più importante è che abbiamo spostato il metodo `changeNick()` in un oggetto indipendente. In questo modo il motore JavaScript non è più costretto a creare ogni volta la stessa funzione in memoria. Ci sono anche alcune novità nel codice La più evidente è `Object.create()`, che come abbiamo già visto è un metodo nativo e si occupa di:

- creare un legame dall'oggetto fornito come argomento verso un nuovo oggetto figlio
- ritornare il nuovo oggetto creato

In altre parole l'oggetto `newPerson` rimane legato all'oggetto `personMethods`. Questo comportamento corrisponde a quanto visto nella sezione precedente con `prototype`. Sfruttando la natura a prototipi di JavaScript possiamo legare un oggetto radice a tutti i figli, che avranno così accesso a un bagaglio di metodi condiviso. Tutto torna e cosa più importante il codice continua a funzionare. Ma dov'è in tutto questo la keyword `new` di cui parlavamo a inizio sezione? Se ricordi abbiamo visto che ogni oggetto JavaScript si ritrova con una proprietà attaccata, chiamata `prototype`. Abbiamo anche visto che `prototype` è a sua volta un oggetto, usato come bagaglio per contenere metodi a disposizione di tutti gli oggetti. Sembra che `personMethods` sia superfluo visto che abbiamo già un oggetto pronto a contenere i metodi in comune. Forse possiamo fare qualcosa del genere?

```
Person.prototype.changeNick = function(nick) {
  this.nick = nick;
};
```

Se anteponiamo la keyword `new` alla nostra funzione `Person` avremo quella che in gergo viene definita *constructor call*. Possiamo semplificare anche `Person` eliminando ogni riferimento ad `Object.create`:

```
function Person(name, nick) {
  this.name = name;
  this.nick = nick;
}
```

Questo perché `new` fa il lavoro per noi:

- ritorna il nuovo oggetto dalla funzione costruttore: possiamo omettere `return newPerson`
- collega il nuovo oggetto a un `prototype` comune: possiamo omettere `Object.create(personMethods)`
- punta nel modo corretto il `this`: possiamo omettere `newPerson.name = name`

Alla fine del refactoring avremo il seguente codice:

```
function Person(name, nick) {
  this.name = name;
  this.nick = nick;
}

Person.prototype.changeNick = function(nick) {
  this.nick = nick;
};

var laura = new Person("Laura", "javaScripter89");

laura.changeNick("javaScripter85");
```

Se dovessi definire `prototype` direi che è come uno zaino sulle spalle di ogni oggetto JavaScript. In questo zaino possiamo conservare tutti i metodi che prevediamo di assegnare ai nuovi oggetti. I vantaggi sono evidenti: sfruttando la natura a prototipi di JavaScript possiamo ottimizzare a costo zero l'utilizzo della memoria.

Nota questa sintassi che è alla base del funzionamento di `new`:

```
new Person("Laura", "javaScripter89");
```

Questa forma in JavaScript è denominata *constructor function*: indica la possibilità di usare funzioni funzioni per costruire altri oggetti. La keyword `new` potrebbe risultare familiare agli sviluppatori Java o C#. Nei linguaggi OOP puri `new` è sinonimo di istanza della classe. In JavaScript invece non è niente di tutto questo e non esiste neanche ereditarietà tra le classi nel senso stretto del termine: tutto quello che abbiamo in JavaScript sono oggetti collegati ad altri oggetti.

E cosa mi dici del pattern OLOO?

Il lettore più esperto potrebbe obiettare che le funzioni non sono l'unico modo per mimare le classi in JavaScript. Gli oggetti infatti possono assolvere allo stesso compito, senza complicare di proposito il codice. Nella serie "You don't know JS" Kyle Simpson propone un'alternativa denominata OLOO pattern che sfrutta solo gli oggetti e il collegamento tra di loro per organizzare il codice, eliminando completamente `new`. Tuttavia la forma constructor function con `new` è quella più familiare per la maggior parte degli sviluppatori JavaScript, e bisogna anche tenere a mente che il pattern OLOO ha implicazioni in termini di performance (la costante creazione di nuovi oggetti è molto spesso penalizzante).

Qualche parola su toString

All'inizio del capitolo abbiamo parlato del metodo `toString()`. Se chiamato su un valore ne ritorna la rappresentazione testuale. Funziona su qualsiasi tipo di dato. Numeri:

```
var nine = 9
nine.toString();
"9"
```

Array:

```
[1,2,3].toString();
"1,2,3"
```

Funzioni:

```
var a = function(){ return false; };
a.toString();
"function(){ return false; }"
```

Booleani:

```
true.toString();
"true"
```

Ma funziona anche su `object`? Proviamo:

```
var obj = { name: "Jacopo" };
obj.toString()
"[object Object]"
```

L'output in questo caso è `"[object Object]"`, una stringa buffa quanto famosa che sono sicuro avrai già visto in qualche meme su JavaScript. Ogni oggetto JavaScript ha un metodo denominato `toString()` collegato ad `Object.toString()`. Alcuni oggetti come gli array implementano una versione personalizzata di `toString()`, in altre parole sovrascrivono `toString()` in modo da restituire una versione in formato stringa di se stessi. Lo stesso vale per le funzioni e per i numeri. Se però chiamiamo `toString()` su un qualsiasi oggetto che non ha sovrascritto il `toString()` originale allora entra in azione `Object.toString()` che per default ritorna la stringa `"[object TipoDiOggetto]` dove TipoDiOggetto è il tipo di oggetto su cui stiamo operando. Qualora volessimo implementare e quindi sovrascrivere `toString()` per i nostri oggetti possiamo ricorrere a prototype:

```
function Person(name, nick) {
  this.name = name;
  this.nick = nick;
}

// implementiamo la nostra versione di toString

Person.prototype.toString = function() {
  return this.name + " " + this.nick;
};

var laura = new Person("Laura", "javaScripter89");

laura.changeNick("javaScripter85");

laura.toString()
"Laura javaScripter85"
```

Sommario del capitolo

In questo capitolo abbiamo fatto luce sull'illusione delle classi JavaScript. Abbiamo imparato che:

- in JavaScript non esistono classi nel vero senso del termine. Quasi tutto è un oggetto.
- anche le funzioni stesse sono oggetti e possiamo utilizzare le *constructor function* come fabbriche per altri oggetti.
- `prototype` è il jolly per agganciare metodi condivisi sui nostri oggetti.
- la keyword `new` è una sintassi di convenienza per snellire la creazione e il legame tra nuovi oggetti JavaScript.

Metti alla prova le tue skills

- Che cos'è una constructor call?
- Che cos'è una constructor function?
- Che cosa significa prototype?
- Sapresti descrivere gli step che avvengono nel motore quando chiamiamo una funzione con `new`?