Tecnologie e applicazioni web

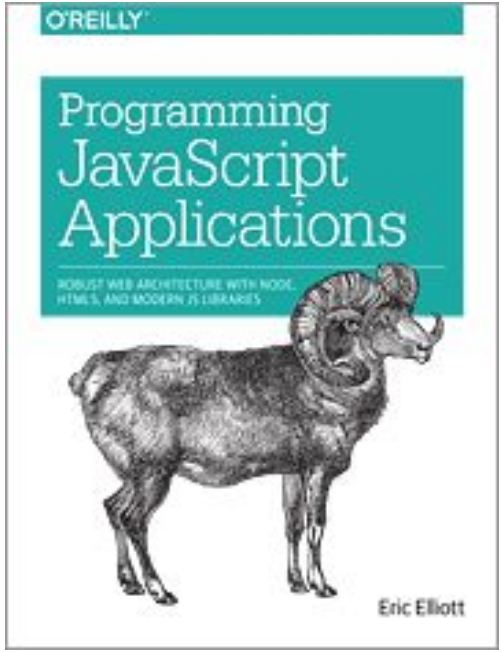
Javascript

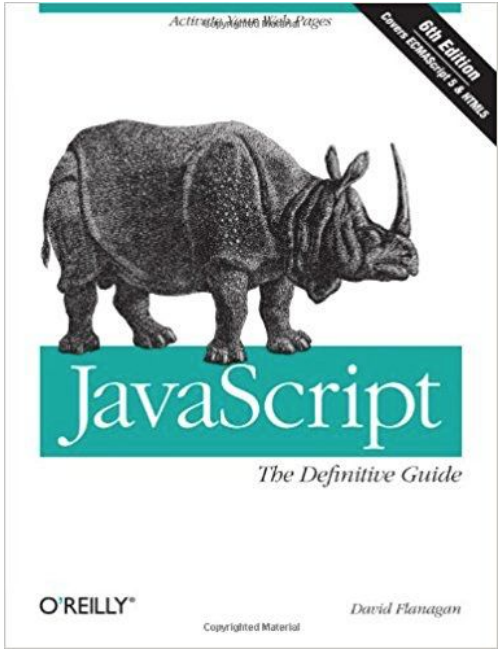
Filippo Bergamasco (filippo.bergamasco@unive.it)
http://www.dais.unive.it/~bergamasco/
DAIS - Università Ca'Foscari di Venezia
Anno accademico: 2017/2018

Atwood's Law: any application that can be written in JavaScript will eventually be written in JavaScript.

—Jeff Atwood

Libri suggeriti





Javascript: Caratteristiche

Javascript è un dialetto particolare (il più usato) del linguaggio standardizzato da parte dell'European Computer Manufacturer's Association noto come **ECMAScript**.

Praticamente tutti i motori Javascript implementati nei browsers supportano ECMAScript versione 5

Javascript: Caratteristiche

A dispetto del nome standard, “Javascript” è il nome comunemente usato dagli sviluppatori.

E’ un linguaggio **interpretato**, ad **alto livello**, **non-tipato** e **dinamico**. E’ particolarmente adatto allo sviluppo di software con paradigma object-oriented e funzionale.

Javascript: Caratteristiche

Il linguaggio definisce un insieme ristretto di funzionalità per lavorare con testi, espressioni regolari, date, array etc. ma non include API di alto livello (ad esempio per gestire l'input/output).

Le API più ad alto livello sono delegate all'ambiente che ospita l'interprete javascript che può essere il browser web oppure un processo stand-alone (come node.js)

I due pilastri di Javascript

1. **Ereditarietà basata su prototype**
   a. Programmazione object-oriented non basata su classi
   b. Oggetti che si riferiscono ad altri oggetti (OLOO: objects linked to other objects)

2. **Programmazione funzionale**
   a. Lambda / funzioni anonime
   b. Closures

Tipi di dati

I tipi di dati gestiti da Javascript sono di due tipi: **primitivi** e **oggetti**.

I tipi primitivi sono: numeri, stringhe, boolean, **null**, **undefined**.

Tutto ciò che non è primitivo è un oggetto che contiene proprietà, ciascuna caratterizzata da un nome ed un valore.

Tipi di dati

Un dato può essere **mutabile** o **immutabile**. Gli oggetti in generale sono mutabili mentre i tipi primitivi non lo sono. Le stringhe sono immutabili sebbene possano essere viste come array di caratteri

L'interprete Javascript esegue automaticamente la garbage-collection degli oggetti che non sono più utilizzati (non più raggiungibili nello scope)

Numeri

Javascript non fa distinzione fra interi e numeri decimali. Tutti i numeri sono rappresentati in virgola mobile a 64 bit.

Javascript non ritorna alcun errore in caso di overflow, underflow o divisione per zero. Esistono oggetti speciali relativi ai corrispondenti valori fp: `NaN`, `Number.POSITIVE_INFINITY`, `Number.NEGATIVE_INFINITY`

Stringhe

Le stringhe si costruiscono con il letterale “ . . ” o ‘ . . . ’

L’operatore + applicato alle stringhe concatena il loro contenuto. Le stringhe forniscono alcuni metodi per la loro manipolazione:

`charAt(<n>), substring(<s>,<e>), slice(<s>,<e>), indexOf(<char>), split(<char>), toUpperCase()`

null / undefined

null e undefined sono tipi speciali utilizzati per indicare l'assenza di un certo valore.

null in realtà è **una keyword** si riferisce ad uno specifico **oggetto** globale (l'unico) di tipo null.
undefined è una variabile globale predefinita (in ES3 si poteva addirittura sovrascrivere il valore di undefined) di tipo undefined (anche in questo caso l'unica del suo tipo).

null / undefined

null è utilizzato per rappresentare la normale assenza di un valore “a livello di programma”. Ad esempio una variabile che contiene un valore al momento indefinito

undefined è utilizzata per rappresentare un’assenza di valore relativa ad una situazione potenzialmente errata, non attesa, o comunque a livello di sistema (es. variabile attesa come argomento di una funzione ma non inizializzata)

Oggetti

Un oggetto è una collezione non ordinata di proprietà, ciascuna formata da una coppia (nome,valore).

Dato che il nome di una proprietà è sempre una stringa, gli oggetti di fatto sono delle mappe (o dizionari) di tipo stringa->valore.

Gli oggetti sono **mutabili** e manipolabili soltanto per **riferimento**

Oggetti

Le proprietà di un oggetto possono essere lette o modificate attraverso l'operatore . (dot) oppure [ ].

Docente.nome = "Filippo"

Docente["nome"] = "Filippo"

Che differenza c'è tra le due modalità di accesso agli oggetti?

Oggetti

Docente.nome = “Filippo”

- “nome” è un identificativo che deve essere definito a priori, pertanto questa notazione permette di avere un numero finito di proprietà per un oggetto

Docente[“nome”] = “Filippo”

- La seconda opzione permette di utilizzare gli oggetti come array associativi, richiedendo come chiave una stringa o qualsiasi espressione castabile a stringa.

Oggetti

Possono essere creati in 3 modi:

1. Utilizzando un'espressione letterale

   ```
   var book = { “title”:”Javascript”, pages:200 }
   ```

2. Utilizzando la keyword **new**, seguita da un invocazione a funzione chiamata costruttore

   ```
   var a = new Array(); var b = new Date();
   ```

3. Invocando la funzione Object.create(<prototype>)

   ```
   var o = Object.create( {x:10, y:20} )
   ```

Oggetto globale

In Javascript esiste sempre un oggetto, chiamato oggetto globale, che esiste nello scope globale.

L'interfaccia dell'oggetto globale dipende dal contesto di esecuzione dello script:

- Browser: l'oggetto globale è una **window**
- Worker: L'oggetto globale è un **WorkerGlobalScope**
- Node.js: L'oggetto globale è un oggetto di tipo "**global**"

Prototype

In Javascript, ciascun oggetto 0 contiene un riferimento ad un altro oggetto (oppure null in rari casi) chiamato **prototype**.

L'oggetto 0 eredita tutte le proprietà del suo prototype: Quando ci si riferisce ad una proprietà di un dato oggetto, l'interprete prima verifica se quella è presente nell'oggetto stesso. Altrimenti, risale la catena di di prototype in cerca della proprietà

Prototype

Gli oggetti creati con un'espressione letterale hanno come prototype un oggetto chiamato `Object.prototype`

Gli oggetti creati con **new** hanno come prototype la stessa proprietà prototype della funzione invocata come costruttore

Ex: var a = new Array();

Avrà come prototype Array.prototype

Prototype

Quando si legge una proprietà di un oggetto viene seguita la catena di ereditarietà dei prototype

Quando si scrive una proprietà di un oggetto **o**:

- Se la proprietà è read-only l'assegnamento non è permesso
- Se la proprietà è ereditata, l'assegnamento crea comunque una nuova proprietà nell'oggetto **o**, non modificando gli oggetti nella catena (il meccanismo è fondamentale perche permette l'override)

.

Funzioni

Le funzioni sono uno dei building blocks di ogni applicazione. In Javascript sono particolarmente importanti perché:

- Javascript possiede funzioni first-class
- Le funzioni sono oggetti
- Le funzioni possono essere definite a runtime
- Supporta le chiusure (closures)

Definire funzioni

Vi sono due modi per definire una funzione:

- Function declaration

```
function sum(a,b) { return a+b; }
```

- Function expression

```
var sum = function(a,b) { return a+b; }
```

Definire funzioni

Function declaration:

- Non permette la dichiarazione condizionale
- Provoca l'**Hoisting** della funzione
- Permette di dare un nome alla funzione stessa, utile per esempio in fase di debug

.

Definire funzioni

Function expression:

- Permette la dichiarazione condizionale
- Non comporta l'**Hoisting**. La dichiarazione segue quindi la logica attesa del programma
- Non è possibile dare un nome alla funzione, che resta pertanto anonima. Si può accedere alla funzione soltanto attraverso l'identificativo della variabile a cui è stata assegnata

.

Scope e Hoisting

Function scope:

Le variabili sono visibili nel codice della funzione che le contiene e in tutte le nested functions in essa contenute.

Hoisting:

Tutte le variabili (e funzioni) dichiarate in una funzione sono visibili in tutto il corpo della funzione

Funzioni Lambda

In Javascript, una funzione lambda è una funzione anonima (dichiarata mediante una function expression) che è utilizzata all'interno del programma come se si trattasse di un valore.

Attenzione: Funzioni anonime e lambda non sono sinonimi. Una funzione anonima non necessariamente è lambda

Funzioni Lambda

Ex:

```
function(a) { return -a; }
```

E' una funzione lambda. La funzione è utilizzata semanticamente come se si trattasse di un dato o valore e ha l'effetto di negare il parametro in input.

.

Invocare funzioni

Il codice contenuto in una funzione è eseguito non quando la funzione viene definita ma quando essa viene invocata. Può avvenire in 4 modi:

- Come un invocation expression: `f(a)`
- Come un method invocation se la funzione è contenuta all'interno di un oggetto:

  `o.f(a) ; o["f"](a)`
- Come una constructor invocation, se preceduta da `new`:

  `var o = new Object()`
- Come invocazione indiretta usando `call()` o `apply()`

this

La keyword **`this`** permette di conoscere il “contesto di esecuzione di una funzione”.

- Fuori da qualsiasi funzione, **`this`** si riferisce all’oggetto globale (ex. window nel browser)
- All’interno di una funzione invocata come espressione:
  - **`this`** si riferisce all’oggetto globale se non è impostato lo strict mode
  - **`this`** è undefined se ci si trova in strict mode

this

La keyword **`this`** permette di conoscere il “contesto di esecuzione di una funzione”.

- All’interno di una funzione invocata come metodo, **`this`** si riferisce all’oggetto di cui la funzione è metodo
  - NOTA: questo comportamento non è per nulla influenzato dal come e dal dove la funzione sia stata definita, soltanto da come viene invocata

this

La keyword **`this`** permette di conoscere il “contesto di esecuzione di una funzione”.

- Nella prototype chain di un oggetto, this si riferisce sempre all’oggetto su cui il metodo è stato chiamato, non all’oggetto che contiene effettivamente quel metodo
  - Questo è un esempio interessante di ereditarietà basata su prototype tipica del linguaggio

this

La keyword **`this`** permette di conoscere il “contesto di esecuzione di una funzione”.

- Se la funzione è usata come costruttore (con la keyword **new**), this si riferisce all’oggetto appena creato
- Se la funzione eredita da Function.prototype e viene invocata con `call()` o `apply()`, this assume il valore dell’oggetto passato come primo argomento del metodo `call()` o `apply()`.

Closures

Javascript utilizza uno *scoping lessicale*: le funzioni sono eseguite utilizzando lo scope delle variabili che era presente **quando sono state definite**, non quello di **quando vengono invocate**.

La combinazione di una funzione e dello scope in cui le variabili della funzione vengono risolte è detto closure (chiusura)

Closures

Solitamente, le funzioni sono invocate nello stesso scope che era valido quando son state definite. In questo caso non ci rendiamo conto che tutte le funzioni Javascript sono chiusure (portano con se lo scope in cui son state definite)

Le cose interessanti si verificano quando la funzione viene invocata in uno scope diverso da quello di definizione (quasi sempre in una nested function)

Closures

```
var scope = "global scope";
function checkscope() {
   var scope = "local scope";
   function f() { return scope; }
   return f();
}
checkscope() // cosa ritorna?
```

Closures

```
var scope = "global scope";
function checkscope() {
   var scope = "local scope";
   function f() { return scope; }
   return f;
}
checkscope()() // cosa ritorna?
```

Closures

```
var scope = "global scope";
function checkscope() {
   var scope = "local scope";
   function f() { return scope; }
   return f;
}
checkscope()() // cosa ritorna?
```

var scope=”local scope” non cessa di esistere dopo la definizione di checkscope() (anche se è locale per la funzione) perché resta legata allo scope della funzione f()

Closures

- Le funzioni hanno accesso alle variabili della funzione in cui sono contenute anche se quella funzione ritorna e quindi cessa di esistere

- Le funzioni salvano dei riferimenti alle variabili della funzione in cui sono contenute e non i valori attuali. Questo permette ad esempio di usare le funzioni per simulare variabili private

L'oggetto arguments

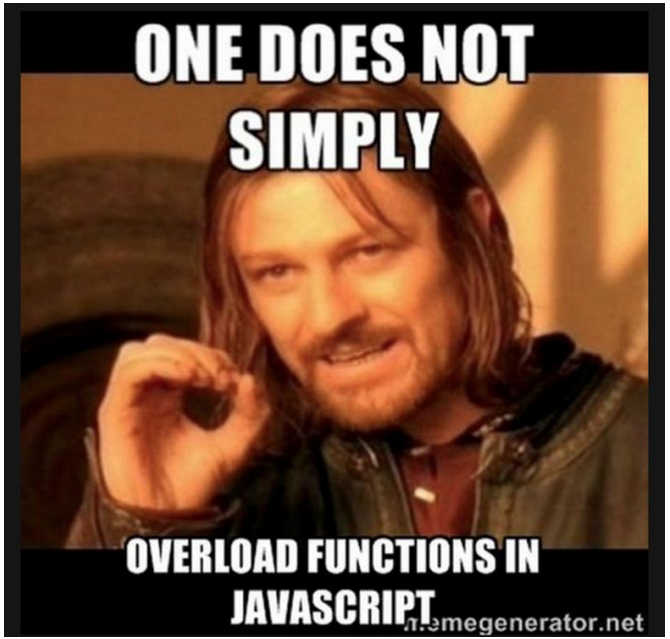
In Javascript è possibile passare un numero di parametri superiore o inferiore rispetto a quelli previsti dalla firma della funzione.

- Se vengono passati meno parametri, quelli mancanti avranno valore `undefined`
- Se vengono passati più parametri, questi sono accessibili attraverso un oggetto chiamato `arguments`

.

Function overloading

In Javascript non è possibile avere funzioni con lo stesso nome e diversi parametri. Quello che avviene è che quella definita per ultima sovrascrive la prima



Function overloading

Due possibili soluzioni:

1. Utilizzare l'oggetto `arguments` per verificare il numero e il tipo dei vari parametri (Poco manutenibile al crescere dei parametri perchè richiede enormi liste di if-then-else)
2. Utilizzare un oggetto con coppie chiave-valore per descrivere nome e valore dei parametri

   (ex: http://api.jquery.com/jquery.ajax)

Classi

Nella programmazione ad oggetti è fondamentale poter definire delle **classi** di oggetti che condividono certe proprietà

In Javascript, le classi non sono definite in modo esplicito ma sono rese possibili dal meccanismo di ereditarietà basato su prototype:

Se due oggetti ereditano delle proprietà da uno stesso oggetto prototype, diciamo che i due oggetti sono due istanze della stessa classe

Costruttori

Il modo idiomatico per creare delle classi è attraverso la definizione di una funzione che agisca da **costruttore** e che inizializzi le proprietà dei vari oggetti della classe.

Utilizzando la keyword `new`, la proprietà “prototype” della funzione costruttore sarà usata per settare la proprietà “prototype” del nuovo oggetto creato.

- Tutti gli oggetti creati con la stessa funzione costruttore avranno lo stesso prototype e pertanto saranno istanze della stessa classe

Costruttori

Attenzione: L'identità di una classe è data dall'oggetto prototype e non dalla funzione costruttore che viene usata per istanziare gli oggetti di quella classe:

- Due costruttori diversi possono avere lo stesso oggetto prototype e pertanto creare oggetti della stessa classe

L'operatore `instanceof` permette di stabilire se due oggetti appartengono alla stessa classe (ie. se hanno lo stesso prototype)

Classi

Una caratteristica interessante delle classi in Javascript è che sono estendibili dinamicamente:

Un oggetto eredita tutte le proprietà dal suo prototype, anche se il prototype viene modificato dopo che l'oggetto è stato istanziato.

Questo permette di "iniettare" nuove funzionalità a tutti gli oggetti che, durante l'esecuzione, sono stati già istanziati. E' possibile in questo modo anche modificare il funzionamento degli oggetti built-in come le stringhe

.

Proprietà private

Uno degli scopi della programmazione object oriented è di incapsulare (nascondere) lo stato di un oggetto e permetterne l'accesso soltanto attraverso i metodi

In Javascript non è possibile modificare la visibilità di metodi e proprietà ma si può simulare questo comportamento con le chiusure

.

Sottoclassi

Una classe B che estende una classe A è una classe che eredita tutti i suoi metodi. Inoltre, B può aggiungere ulteriori metodi o sovrascriverli (override).

In Javascript non vi è una sintassi esplicita per definire classi e sottoclassi. La funzionalità deve essere implementata manualmente con una inizializzazione particolare del prototype di B

Sottoclassi

Se B è sottoclasse di A, allora B.prototype deve essere figlio (o un erede) di A.prototype. In questo modo tutti i metodi del prototype di A saranno ereditati anche dalla classe B.

Sfruttando i metodi `call()` o `apply()` è possibile cambiare il contesto di esecuzione di un metodo e quindi chiamare metodi delle superclassi. (simile alla keyword `super` di altri linguaggi)

.

Composition

La composizione di classi è concettualmente più semplice da implementare:

- Si aggiunge la “superclasse” come proprietà (privata o pubblica) all’interno della “sottoclasse”
- Si implementano le funzionalità aggiuntive e forwardano manualmente tutti i metodi “da ereditare”

La composizione di classi andrebbe sempre favorita rispetto alla classica class inheritance. Perchè?

Problemi dell'ereditarietà

Accoppiamento stretto

Le sottoclassi di una gerarchia dipendono in modo stretto dalle loro superclassi, conoscendone in molti casi i dettagli interni

Gerarchie non flessibili

Le gerarchie formate dall'ereditarietà classica creano delle tassonomie fisse che non modellano correttamente il problema. Spesso questo causa duplicazione del codice tra superclasse e sottoclasse

Problemi dell'ereditarietà

Ereditarità multipla è complessa

L'ereditarietà multipla è complessa da implementare e spesso inconsistente con quella singola

Gorilla/Banana problem

Può accadere che ci siano funzionalità della superclasse che non ha senso ereditare. L'ereditarietà classica non prevede di poter scegliere quali funzionalità ereditare

Mixins

La flessibilità di Javascript ci permette di sfruttare la composition in modo ancora più efficiente se immaginiamo di usare degli oggetti come semplici contenitori di funzionalità che possono essere “prese in prestito” e mixate per creare nuovi oggetti.

Questi oggetti sono detti **mixins** e uniscono i vantaggi dell’utilizzo di interfacce con quelli derivanti dall’ereditarietà tra classi (riutilizzo del codice)

.

Moduli

Uno dei motivi di organizzare il codice in classi è quello di renderlo “modulare” e poterlo quindi riutilizzare in diversi contesti

Possiamo immaginare un concetto più generico di **modulo** che contiene non solo la definizione di oggetti, ma anche funzioni e in generale codice da eseguire

Un modulo espone soltanto un set ristretto di API pubbliche

Principi base dei moduli

Essere specializzati

Un modulo dovrebbe avere uno specifico scopo. Tutte le parti del modulo dovrebbero concorrere soltanto a risolvere il problema per cui il modulo è stato creato

Indipendenti

Un modulo dovrebbe usare il meno possibile altri moduli. Invece di invocare direttamente le API di altri moduli dovrebbe comunicare attraverso dei mediatori (command pattern)

Principi base dei moduli

Scomponibili

Un modulo dovrebbe poter essere usato e testato in completo isolamento rispetto ad altri moduli

Componibili

Moduli diversi devono poter cooperare e coesistere insieme per sviluppare applicazioni differenti

Sostituibili

Un modulo deve poter essere sostituito con un altro purchè esponga la stessa interfaccia

Module design pattern

Per implementare il concetto di modulo nel browser possiamo sfruttare le closures e le IIFE (Immediately Invoked Function Expressions).

- Il modulo è incapsulato in un IIFE
- L'interfaccia pubblica del modulo è definita per assegnamento

Svantaggio: Per funzionare, ciascun modulo deve esporre almeno una variabile globale

CommonJS

Per ovviare al problema, possiamo passare una variabile esistente che sarà estesa con tutte le funzionalità rese pubbliche dal modulo

Questa modalità è simile a quella implementata in Node.js e in generale “standardizzata” dal comitato CommonJS.

CommonJS

```
// mathfunctions.js
module.exports = {
  sum: function(a, b) { return a + b; }
};

// app.js
var mathfunctions = require('./mathfunctions);
mathfunctions.sum(1, 2);
```

CommonJS: problemi

Lo svantaggio principale è che la chiamata `require()` è sincrona al caricamento del codice del modulo.

Pertanto, questo meccanismo non può essere implementato nel browser dove il caricamento del codice Javascript deve avvenire necessariamente in modo asincrono

AMD

Il comitato CommonJS ha definito anche una modalità asincrona per il caricamento dei moduli detta “Asynchronous Module Definition” (AMD)

```
// mathfunctions.js
define("mathfunctions", function() {
  return {
    sum: function(a, b) { return a + b; }
  };
});

// app.js
define("app", ["mathfunctions"], function(mathfunctions) {
  console.log(mathfunctions.sum(1, 2)); // => 3
});
```

Nome del modulo

Dipendenze da caricare in modo asincrono

AMD

Il meccanismo di caricamento AMD è implementato nella famosa libreria Require.js:

http://requirejs.org

Ed utilizzato in genere in sinergia con il gestore di pacchetti front-end Bower:

https://bower.io

Problema: Il load asincrono di tutti i moduli può aumentare sensibilmente il tempo di caricamento del codice front-end di una SPA

Browserify + npm

Per evitare l'attesa dovuta al caricamento asincrono di tutti i moduli, un'alternativa è fornita dal progetto Browserify:

http://browserify.org

Permette di creare in modo automatico un unico "bundle" da includere nelle pagine di front-end. Il meccanismo di caricamento dei moduli è poi lo stesso di CommonJS

Operatori di uguaglianza

Una caratteristica spesso contestata di Javascript è quella di prevedere due diversi operatori di uguaglianza: `==` e `===`.

Entrambi ritornano `true` o `false` a seconda che i due operandi siano rispettivamente uguali o diversi. Il primo si riferisce ad una definizione più “rigorosa” di uguaglianza, il secondo ad una più accomodante che effettua una politica aggressiva di type coercion

A === B

1. Se A e B hanno tipi diversi, ritorna false
2. Se A e B sono entrambi null o undefined, ritorna true
3. Se A e B sono entrambi true o entrambi false, ritorna true
4. Se entrambi sono NaN, ritorna false (NaN non è mai uguale a nessun valore, incluso se stesso)
5. Se sono due numeri uguali, ritorna true

A === B

6. Se A e B sono stringhe, e hanno esattamente la stessa sequenza di bytes, allora ritorna true. (Le stringhe unicode non vengono normalizzate)
7. Se A e B si riferiscono allo stesso oggetto, array o funzione, ritorna true. (Gli oggetti sono sempre confrontati per riferimento, un oggetto è sempre uguale a se stesso e mai ad un qualsiasi altro oggetto, nemmeno se contiene le stesse proprietà)

A == B

1. Se A e B hanno lo stesso tipo, ritorna A===B
2. Se A è null e B è undefined, o viceversa, ritorna true
3. Se uno dei valori è stringa e l'altro è numero, converte la stringa in numero e ritorna A==B
4. Se A o B sono true o false, converte true in 1, false in 0 e ritorna A==B
5. Se uno degli operandi è un oggetto, lo si converte in stringa e ritorna A==B

A == B

6. Se tutte le operazioni precedenti falliscono, ritorna false

Ad esempio, in Javascript “1” == true:

1. `1 == true // converte la stringa “1” in numero`
2. `1 == 1 // converte true in 1`

eval()

Uno dei vantaggi derivanti dal fatto di essere un linguaggio interpretato è quello di poter interpretare al volo stringhe contenenti codice Javascript per produrre un valore

Ex:

```
eval(“3+2”)  // => 5
```

...può essere utile in alcuni rari contesti

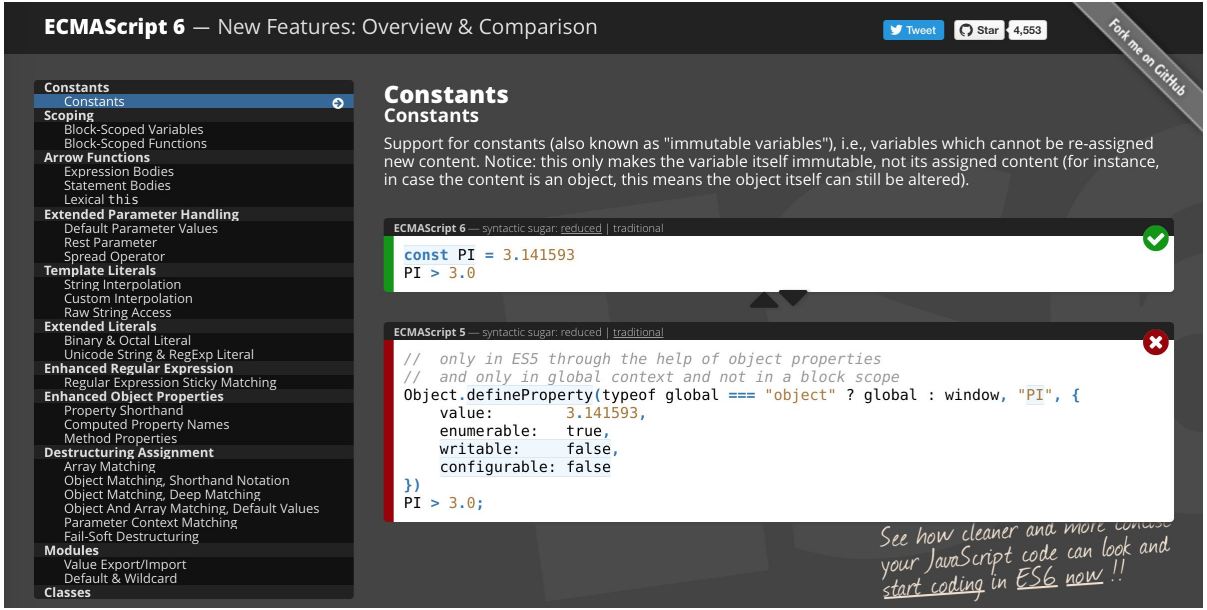
eval()

La cosa interessante è che `eval()` usa l'ambiente (con tutte le variabili) del codice che lo invoca:

```
var x = 1;
eval( “x=2”); // modifica il valore di x
```

Per il fatto che eval ha accesso alle variabili esterne, anche globali, questo ha un impatto forte sull'ottimizzatore, che non può operare sulle funzioni che invocano `eval()`

ES 6 - Features importanti



http://es6-features.org/

ES 6 - Variabili

In ES 6 è possibile definire variabili il cui scope è il blocco di codice corrente senza che queste siano visibili prima della loro dichiarazione (hoisting)

Si utilizza la keyword **`let`** al posto di **`var`**

ES 6 - Funzioni

In ES 6 è possibile evitare l'hoisting di una funzione se la si racchiude all'interno di un blocco di codice

Si può utilizzare l'operatore **=>** (arrow function) per creare funzioni anonime (o espressioni lambda) senza utilizzare la keyword **function:**

```
( <args> ) => <expression/function body>
```

ES 6 - Funzioni

Le arrow function supportano il cosiddetto “lexical this”. Vale a dire che la keyword **`this`** all’interno di una arrow function fa riferimento allo stesso oggetto **`this`** del codice che contiene la arrow function.

Senza le arrow functions lo stesso funzionamento va invece “simulato” usando le chiusure.

Classi

ES6 introduce delle “funzioni speciali” per semplificare la sintassi di creazione degli oggetti e gestire l'ereditarietà:

```
class Polygon {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }

  get area() { return this.calcArea() }

  calcArea() { return this.height * this.width; }
}
```

Classi

La sintassi non introduce un nuovo modello di eredità class-oriented a Javascript, che resta basata sul meccanismo dei prototype

La nuova sintassi prevede anche meccanismi espliciti per gestire l'ereditarietà (extends) e l'ereditarietà a partire da espressioni, come avviene in stile mixins.

ES6 - Compatibilità

La definizione dello standard ECMAScript versione 6 è stata resa definitiva nel 2015 ed è attualmente implementata nelle versioni aggiornate dei browser.

In Node.js è supportato dalla versione 6.5 in poi

Per una lista degli ambienti compatibili:

http://kangax.github.io/compat-table/es6/