JavaScript

Introduzione al linguaggio

Giovanni Iovino: giovanni.iovino@intecs.it

Premessa:

L'obiettivo del corso non è insegnare i principi di programmazione generali a chi non ha mai sviluppato. Il corso è rivolto agli sviluppatori che intendono usare il linguaggio di programmazione JavaScript (per la prima volta o meno) gli strumenti e le nozioni necessarie per comprendere alcuni meccanismi caratteristici del linguaggio che spesso sono causa di errori se non propriamente compresi.

Breve storia

- Originariamente introdotto da Netscape in Netscape Navigator con il nome di LiveScript
- Il nome è stato cambiato in JavaScript data la sintassi simile a Java (la sintassi è l'unico punto in comune)
- Standardizzato da ECMA con il nome di ECMAScript dal 1997:
    - Ultima versione stabile ECMA-262 2020 (https://262.ecma-international.org/11.0/)

Caratteristiche

le caratteristiche principali del linguaggio

- Linguaggio interpretato
- Sintassi c-like
- Tipi dinamici
- Orientato agli oggetti [prototype-based*]
- Multi paradigma - Procedurale, Funzionale, orientato agli eventi

Linguaggio interpretato

runtime environment

- Normalmente utilizzato come linguaggio di scripting all'interno di altri applicativi
  - Il programma ospite (es. browser) offre il runtime environment in cui è eseguito il codice JS offrendo i metodi per interagire con l'environment (es. accesso alle risorse di sistema)
- Per essere eseguito il codice js richiede un interprete o engine (es. V8 sviluppato da Google e utilizzato all'interno di Chrome)
- **Run-time evaluation:** a differenza di altri linguaggi interpretati permette l'esecuzione di codice rappresentato come una stringa

```
eval("var p = 10;");
```

Linguaggio interpretato

Just in Time compilation

- Molti interpreti JS supportano la compilazione JIT:
  - Il codice viene compilato a **runtime** e convertito in bytecodes prima di essere esguito
  - L'engine utilizza i bytecodes generati in modo ottimizzato per rendere l'esecuzione del codice più veloce

NOTA: il supporto alle specifiche dello standard, a JIT e le prestazioni del codice JS dipendono dall'interprete utilizzato!

Sintassi & Costrutti

Sintassi simile a quella del C e Java.

La maggior parte dei costrutti di C e Java sono disponibili in JavaScript

```
/* if - else */
if(condizione)
{
      //... codice
}
else
{
      //.. codice

}
```

```
/* while */
while(condizione)
{
      //... codice
}

/* do while */
do(condizione)
{
      //... codice
}while(condizione);
```

```
/* switch */
switch(expression)
{
  case x:
  //... codice
  break;

  case x:
  //... codice
  break;

  default:
  //... codice
}
```

```
/*
 Commento
 multilinea
*/

// commento in-line
```

```
/* for */
for(statement1; statement2; statement3)
{
      //... codice
}
```

```
/* try-catch */
try
{

  //... codice

}
catch(error)
{

  //... in caso di errore

}
```

Sintassi & Costrutti

Automatic semicolon insertion:

A differenza di C e JAVA in Javascript il parser inserisce automaticamente il punto e virgola al verificarsi di alcune condizioni:

- Se la linea di codice successiva "**interrompe**" quella precedente
- Quando la linea di codice successiva inizia con **}**
- Quando si raggiunge la fine del codice
- Se è presente il **`return`** nella linea di codice corrente
- Quando è presente un **`break`** nella linea di codice corrente
- Quando è presente **`throw`** nella linea di codice corrente
- Quando è presente **`continue`** nella linea di codice corrente

```
// Esempi
const a = 10; // OK

const b = 20 // OK regola 1

const c = "Ciao" // OK regola 1

const d = "Mondo"

[c, d, "come va?"].join(" ");

/*

 Errore viene interpretato come:

 const d ="ciao"[c, d, "come va?"].join(" ");

*/
```

Tipi

I tipi primitivi in JavaScript sono:

- **String** - Una sequenza di caratteri
- **Number** - interi o float
- **BigInt** (nuovo in ECMAScript 2020) - Interi con precisione arbitraria
- **Boolean** - true e false
- **null** - un oggetto speciale che indica un oggetto nullo
- **undefined** - usato per indicare valori non definiti
- **Object**
- **Symbol** (nuovo in ES6) - Oggetto le cui istanze sono uniche e immutabili

Tutte le variabili che non ricadono in un tipo tra quelli elencati sono di tipo Object.

Un caso particolare sono le **funzioni**, che vedremo in seguito.

```
// Esempi
const a = 20.05; // Number float

const b = 20; // Number int

const c = “Ciao”; // String

const d = true; // Boolean

var o = {  // Object

   p1: “Prop1”, // Property di tipo String
   p2: 20 // Property di tipo Number

};

var ar = [1, 2, 3]; // Object
```

JS

Tipi - typeof

Per controllare il tipo di una variabile a runtime possiamo utilizzare l'operatore **typeof** che restituisce una **stringa** rappresentante il tipo della variabile a cui applichiamo l'operatore.

sintassi: **typeof** *instance*

oppure: **typeof(***instance***)**

I valori restituiti per ogni tipo sono:

- **String** - "string"
- **Number** - "number"
- **BigInt** - "bigint"
- **Boolean** - "boolean"
- **null** - "object"
- **undefined** - "undefined"
- **Object** - "object"
- **Symbol**- "symbol"
- **Function** - "function"
- Tutti gli altri oggetti - "object"

```
// Esempi
const a = 20.05; // Number float

typeof a; // "number"
```

Null è un oggetto:

```
// null vs undefined
var a; // Declaration

typeof a; // "undefined"


a = null;

typeof a; // "object"


a = undefined;

typeof a; // "undefined"
```

Le funzioni hanno un tipo:

```
const f = function(){
        // ...
}

typeof f; // "function"
```

Standard built-in objects

Sono oggetti predefiniti e accessibili a livello globale, di uso comune in JavaScript.

Hanno proprietà e metodi built-in che servono ad eseguire operazioni comuni per l'accesso e la modifica dei dati che contengono.

Oggetti fondamentali:

Oggetti generali di Javascrpt, sui quali sono basati tutti gli altri oggetti. Rappresentano oggetti, funzioni ed errori.

- Object
- Function
- Boolean
- Symbol
- Error
- EvalError
- InternalError
- RangeError
- ReferenceError
- SyntaxError
- TypeError
- URIError

Numeri e date:

- Number
- Math
- Date

Elaborazione del testo:

- String
- RegExp

Collezioni ordinate:

- Array
- Int8Array
- Uint8Array
- Uint8ClampedArray
- Int16Array
- Uint16Array
- Uint16Array
- Int32Array
- Uint32Array
- Float32Array
- Float64Array

Collezioni chiave-valore:

- Map
- Set
- WeakMap
- WeakSet

ed altri...

JS

Standard built-in objects - Array

A titolo di esempio vediamo come si usano gli Array.

Per la lista di tutte le proprietà e i metodi degli oggetti Array fare riferimento a:

https://developer.mozilla.org/it/docs/Web/JavaScript/Reference/Global_Objects/Array

In generale le reference guide sono le vostre migliori amiche

```
var a = [0,1,3,4]; // Declaration
let b = a.length; // 4
a.push(5); // a = [0,1,2,3,4,5]

b = a.pop(); // b = 5, a = [0,1,2,3,4]
a.splice(0,1); // a = [1,2,3,4]
a.splice(1,0, 5); // a = [1,2,5,3,4]
a = a.sort(); // a = [1,2,3,4,5]
let str = a.join(“, “); // str = “1, 2, 3, 4, 5”
a = a.reverse();
```

JS

Variabili

Le variabili in JavaScript **hanno un tipo,** ma a differenza di altri linguaggi fortemente tipizzati, Javascript utilizza i **tipi per valore** e non per variabile.

Le variabili possono essere dichiarate come:

- **var** - per variabili sia locali che globali
- **let** - per variabili locali
- **const** - per variabili locali immutabili

Va prestata particolare attenzione all'uso di variabili non dichiarate esplicitamente a causa dell'**auto-global variable declaration** che avviene in JavaScript.

```
var a; // Declaration

let b; // Declaration

b = 10; // OK

const c = 100;
c = 200; // Errore - c è una costante


n = 1000; // OK!! Auto global var declaration!


function f(){
  pippo = "ciao";
}

console.log(pippo); // Error pippo is not defined

f(); //chiamata a f()

console.log(pippo); // "ciao"
/*
pippo assegnata in f e non dichiarata esplicitamente
risulta essere una variabile globale a causa dell'
auto dichiarazione delle variabili globali!
*/


let x = 10; // Declaration as number

b = "Ciao"; // x now is a string
```

Strict mode

Introdotto in ES5 serve a dichiarare che si vuole usare una versione “**ristretta**” di JavaScript. Per ristretta si intende una serie di regole che cambiano la semantica di Javascript:

- Alcuni errori che normalmente non generano eccezioni in strict mode generano un’eccezione
- Aiuta a prevenire errori che impediscono una corretta ottimizzazione del codice. Il codice scritto in strict mode può essere più veloce.

Si attiva inserendo `‘use strict’` nel codice. Si può applicare anche solo a parti specifiche del codice.

```
‘use strict’
//Tutto il codice nel modulo è in strict mode
let b; // Declaration
b = 10; // OK

n = 1000; // Error! n non è definita! No auto-global var
```

```
// Solo il codice di f è in strict mode
function f(){
  ‘use strict’
  pippo = “ciao”;
}

n = 1000; // OK
```

Coercizione dei tipi

Avviene quando si converte il tipo di una variabile in un altro tipo. Può essere:

- **Esplicita:** Avviene a seguito di un'operazione di cambio di tipo volontaria
- **Implicita:** Avviene in modo automatico a seconda delle operazioni che si effettuano sulle variabili.

Va prestata particolare attenzione alla implicit coercion, che può rivelarsi uno strumento utile quanto causa di errori difficili da individuare in fase di debug del codice. Le regole di coercizione implicita non sono generiche ma dipendono dai tipi primitivi, quindi da usare con consapevolezza.

```
let a = 10; // Declaration
let b = "10"; // Declaration

let c = Number(b); // Explicit coercion


let d = a*b; // Implicit coercion d = 100

let e = a+b; // Implicit coercion d = "1010"
```

Vero e Falso

In JavaScript le seguenti espressioni sono vere quando usate come espressioni booleane:

- "hello" - stringhe non vuote
- 5 - numeri diversi da 0
- true
- [ ], [1, "2", 3] - Array vuoti o meno
- { }, { a: 42 } - Oggetti vuoti o meno
- funzioni

Mentre le seguenti sono valutate come false:

- "" - stringhe vuote
- 0, -0, NaN - numeri non validi e zero
- null, undefined
- false

```
var a = {
  p1 = 10;
  p2 = [20];
}

if(a.p1){
 console.log(a.p1);
}

if(a.p3){
 console.log(a.p3);
}
```

== oppure ===

In JavaScript esistono due operatori di uguaglianza che possono essere utilizzati per il confronto di due espressioni:

- **a ==b** Esegue il confronto tra i valori di a e b, si applica la coercizione dei tipi. Ritorna true se i due valori sono uguali
- **a ===b** Esegue il confronto tra i valori e i tipi di a e b, non si applica la coercizione dei tipi. Ritorna true se entrambi i valori e i tipi sono uguali

```
var a = 10;
var b = “10”;

if(a == b){ // true
  ...
}
if(a === b){ // false
  ...
}
```

Funzioni

- Le funzioni in JavaScript sono oggetti speciali
- Possiamo assegnare una funzione ad una variabile
- Possiamo passare una funzione come argomento di un'altra funzione

Normalmente le funzioni per essere eseguite devono essere invocate.

In Javascript si possono eseguire le funzioni senza invocarle tramite l'uso delle **Immediately Invoked Function Expressions (IIFEs).**

Essendo di fatto funzioni le IIFEs possono anche restituire valori

```
function f(){
  console.log("ciao");
}
f();
const somma = function(a, b){
  return a+b;
}
var x = somma(2,8);

const diff = function(a, b){
  return a-b;
}
const operation = function(a, b, op){
  return op(a,b);
}
var y = operation(2,8, somma); //y = 10
y = operation(y,3, diff); //y = 7

//typeof operation -> "function"
```

```
console.log("ciao");

//IIEF

(function f(){

  console.log("mondo");

})();


/* eseguendo il codice verrà stampato
"ciao mondo" perché f è invocata
immediatamente
*/
```

```
console.log("ciao");

//Arrow functions

const a = (a,b)=>{
  return a+b;
}


/* Nota: Le funzioni arrow hanno un
comportamento diverso a livello di
scoping
*/
```

JS

Hoisting

In Javascript è possibile accedere alle variabili prima che vengano dichiarate, grazie al meccanismo di **hoisting.**

E' possibile a causa di come gli interpreti eseguono il codice JavaScript:

- Durante la fase di compilazione le dichiarazioni delle variabili **var** e delle **funzioni** vengono sollevate in cima al codice e **inizializzate a undefined**
- Contrariamente le variabili dichiarate come **let** o come **const** e le **classi** vengono '*sollevate*' **ma non sono inizializzate.** Vengono inizializzate solo dopo che viene processata la riga di codice che le dichiara.

```
// console.log(b) darebbe  undefined
b = 10; // OK - Non è auto-global declaration
var b; // Declaration
```

```
f(); // OK - f è hoisted in fase di compilazione
function f(){
  // ...
}
```

```
f(); // Errore - f è hoisted ma è undefined quindi non è
     // una funzione
var f = function(){
  // ...
}
```

```
b = 10; // Errore b non è definita
let b=0; // Declaration
```

```
function f(){
  b=1000;  //OK f() è chiamata dopo la dichiarazione di b
}
let b=0; // Declaration
f();
```

JS

Hoisting

il meccanismo di **hoisting** è eseguito in base allo scope**.**

```
x = 5; // OK
f();  // OK

function f(){ // f sollevata in cima al modulo
  x = 10; // OK
  var x; // Declaration sollevata in cima a f()
}

var x; // Declaration sollevata in cima al modulo

// quanto vale x?
```

JS

Scope

Lo scope è il contesto in cui esistono e sono valutate le espressioni e le variabili.

Il lexical-scoping è il meccanismo che avviene in fase di compilazione (fase di lexing) per stabilire lo scope di una variabile.

In JavaScript lo scope può essere a livello di **funzione** o a livello di **blocco**, ma lo scope a livello di blocco non si comporta come in altri linguaggi!

```
var x = 5;

function f(){
  var y = 10; // Function scope
}
for(var 0=1;i <10; i++){
  var z = i;

}

if(1){
  var x = 10;
  console.log(x);
}


console.log(z); // 9
console.log(x); // 10
console.log(i); // 10
console.log(y); // Error - y is not defined
```

JS

Let’s make block-scope

La keyword **let** introdotta in ES6 per dichiarare le variabili localmente ci aiuta ad implementare codice con block-scoping utile.

Un comportamento analogo si ha con la keyword **const.**

```
var x = 5; // Che tipo di scope è?

function f(){
  var y = 10; // Function scope
}
for(let i=1;i <10; i++){ // Block scope
  let z = i; // Block scope

}

if(1){
  let x = 10; // Block scope
  console.log(x);
}


console.log(z); // Error - z is not defined
console.log(x); // 5
console.log(i); // Error - i is not defined
console.log(y); // Error - y is not defined
```

JS

Closure

Una closure è la combinazione di una funzione e dello stato nella quale è stata creata (ambito lessicale).

Gli scope sono annidati, la funzione inner può accedere a x, a y e a k.

Viceversa f non può accedere a yin.

```
var x = 5;

var k = 5;


function f(){

  var x = 10; // Function scope
  var y = 10; // Function scope


  function inner(){

    var yin = 2; // Function scope

    console.log(k + x + y + yin);
  }

  inner(); // stampa 27

}

f();
```

JS

Closure

Sappiamo che le funzioni sono oggetti. Cosa accade se f ritorna la funzione inner?

Closure!

Ogni istanza di inner ritornata da f ha uno stato proprio, con una closure sullo scope di f

```
function f(val){
  var y = val; // Function scope
  function inner(){
    var yin = 2; // Function scope
    console.log(y + yin);
  }
  return inner;
}
const f1 = f(10);
const f2 = f(0);
f1(); // stampa 12
f2(); // stampa 2
```