

Laboratorio I

a.a. 2025/2026

Visibilità delle dichiarazioni: lo scoping

Contenuti


- Le dichiarazioni in JavaScript
- Visibilità delle dichiarazioni
 - Scope di visibilità
 - Annidamento, *shadowing* e *hoisting*
 - Differenze fra **var**, **let**, **const** e **function**
- Closure
- Soluzione degli esercizi della lezione precedente

Dichiarazione e inizializzazione

La **dichiarazione** di una variabile consiste nel definire il suo **nome** (identificatore)

L'**inizializzazione** di una variabile consiste nell'assegnarle un valore per la prima volta

Ricordate: una **dichiarazione** di funzione è un modo diverso (ma equivalente) di scrivere una **dichiarazione** di variabile, e **inizializzarla** con un valore di tipo funzione

<pre>function nome(a₁, ... a_n) { /* corpo */ }</pre>		<pre>var nome = (a₁, ... a_n) => { /* corpo */ }</pre>
----------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------

Dichiarazione e inizializzazione

La **dichiarazione** può essere fatta sia con che senza **inizializzazione**:

var x

dichiarazione

dichiarazione
var x = 5

inizializzazione

Le variabili non inizializzate hanno valore **undefined**.

Visibilità delle dichiarazioni

Un nome dichiarato è **visibile** in certe parti del programma, più o meno ampie a seconda di *dove* è stato dichiarato e di *quale tipo* di dichiarazione viene usata.

JavaScript prevede tre **scope** o **ambiti di visibilità**:

- Scope globale
- Scope di funzione
- Scope di blocco

```
console.log("benvenuto")
n = 1

function map(f,a) {
  let r=[]
  for (var i=0;i<a.length;i++) {
    r.push(f(n,a[i]))
    n= 1-n
  }
  return r
}

function mul(a,b) { return a*b }

const aa = [4,7,12,3,22,1,6]
var t=map(mul,aa)
console.log(t)
```

Visibilità delle dichiarazioni

Le **dichiarazioni** fatte fuori da qualunque funzione avvengono nello **scope globale**.

Queste dichiarazioni sono **visibili** in tutto il programma, sia fuori che dentro le funzioni o i blocchi.

Le variabili non dichiarate dal programma, ma usate senza dichiarazione, si intendono implicitamente dichiarate nello scope globale.

- Scope **globale**
- Scope di funzione
- Scope di blocco

```
console.log("benvenuto")  
n = 1
```

```
function map(f,a) {  
  let r=[]  
  for (var i=0;i<a.length;i++) {  
    r.push(f(n,a[i]))  
    n= 1-n  
  }  
  return r  
}
```

```
function mul(a,b) { return a*b }
```

```
const aa = [4,7,12,3,22,1,6]  
var t=map(mul,aa)  
console.log(t)
```

Visibilità delle dichiarazioni

Ogni funzione definisce il proprio **scope di funzione**.

I parametri formali si intendono dichiarati nello scope di funzione (ma il nome della funzione no: è nello scope globale!).

Le **dichiarazioni** fatte all'interno di una funzione con **var** sono **visibili** nello scope di quella funzione, ma non fuori.

- Scope globale
- Scope di **funzione**
- Scope di blocco

```
console.log("benvenuto")  
n = 1
```

```
function map(f,a) {  
  let r=[]  
  for (var i=0;i<a.length;i++) {  
    r.push(f(n,a[i]))  
    n= 1-n  
  }  
  return r  
}
```

```
function mul(a,b) { return a*b }
```

```
const aa = [4,7,12,3,22,1,6]  
var t=map(mul,aa)  
console.log(t)
```

Visibilità delle dichiarazioni

Ogni blocco (0 o più comandi racchiusi fra {...}) definisce il proprio **scope di blocco**.
In particolare, il corpo di una funzione è un blocco.
Un blocco può essere contenuto in un altro blocco.
Le dichiarazioni all'interno di un `for` si considerano fatte nel suo blocco¹.

Le **dichiarazioni** fatte all'interno di un blocco con `let` e `const` sono **visibili** nello scope di quel blocco, ma non fuori.

- Scope globale
- Scope di funzione
- Scope di **blocco**

```
console.log("benvenuto")
```

```
n = 1
```

```
function map(f,a) {  
  let r=[]  
  for (var i=0;i<a.length;i++) {  
    r.push(f(n,a[i]))  
    n= 1-n  
  }  
  return r  
}
```

```
function mul(a,b) { return a*b }
```

```
const aa = [4,7,12,3,22,1,6]
```

```
var t=map(mul,aa)
```

```
console.log(t)
```

¹ anche `if` e `while`, ma uso rarissimo!

Annidamento e *shadowing*

Notate che gli scope sono contenuti uno nell'altro:

- Lo scope **globale** contiene 0 o più scope di **funzione** e 0 o più scope di **blocco**
- Ogni scope di **funzione** contiene 0 o più scope di **funzione**, e 1 o più scope di **blocco**
- Ogni scope di **blocco** contiene 0 o più scope di **funzione** e 0 o più scope di **blocco**

Quando JavaScript incontra una dichiarazione di un identificatore, definisce una nuova variabile con quel nome e inserisce il nome nel blocco corrispondente.

Quando JavaScript incontra un riferimento a un identificatore, lo interpreta come un riferimento alla dichiarazione più interna con lo stesso nome, partendo dalla posizione in cui incontra il riferimento.

Annidamento e *shadowing*

```
var x=1

function f() {
  console.log(x)
  for (var i=0;i<10;i++) {
    let x=i*2
    g(x+1)
  }

  function g(n) {
    let f = x=>2*x
    console.log(f(n))
  }
}

console.log(x)
f()
```

Proviamo a trovare le “corrispondenze” fra usi e dichiarazioni:

- La prima occorrenza di x in f?
- Le occorrenze di i in f?
- La seconda occorrenza di x in f?
- L'occorrenza di g in f?
- Le occorrenze di f in g?
- Le occorrenze di x in g?
- Le occorrenze di n in g?
- L'ultima occorrenza di x e di f nel prog?

Annidamento e *shadowing*

```
var x=1

function f() {
  console.log(x)
  for (var i=0;i<10;i++) {
    let x=i*2
    g(x+1)
  }

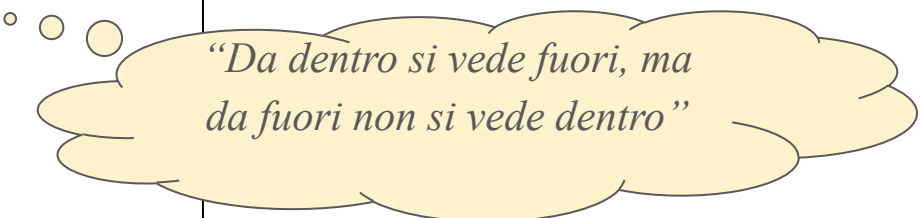
  function g(n) {
    let f = x=>2*x
    console.log(f(n))
  }
}

console.log(x)
f()
```

La regola del “dall’interno all’esterno” fa sì che sia possibile ri-usare lo stesso nome all’interno di uno scope senza troppi problemi.

La variabile esterna non è più visibile (quella interna con lo stesso nome la “copre”, *shadowing*)

I nomi dichiarati dentro uno scope sono **locali** o **privati** allo scope.



“Da dentro si vede fuori, ma da fuori non si vede dentro”

Sollevamento o *hoisting*

Le dichiarazioni, ovunque si trovino all'interno di uno scope, sono implicitamente spostate all'inizio del loro scope (*hoisting*).

<https://developer.mozilla.org/>

JavaScript Hoisting refers to the process whereby the interpreter allocates memory for variable and function declarations prior to execution of the code. Declarations that are made using **var** are initialized with a default value of **undefined**.

Declarations made using **let** and **const** are not initialized as part of hoisting. Until the line in which they are initialized is executed, any code that accesses these variables will throw an exception.

Sollevamento o *hoisting*

Le dichiarazioni, ovunque si trovino all'interno di uno scope, sono implicitamente spostate all'inizio del loro scope (*hoisting*).

Attenzione: le inizializzazioni però rimangono dove sono scritte, quindi la variabile prima della riga in cui è dichiarata sarà visibile, ma non inizializzata (`undefined` nel caso di `var`)

Eccezione: per le funzioni dichiarate con `function`, anche l'inizializzazione si considera fatta all'inizio dello scope (vedi esempio precedente con `g()`)

Uso comune: mettiamo da subito le dichiarazioni all'inizio dello scope, così non avremo brutte sorprese...

var, let, const e function

Riassumendo:

1. JavaScript ha scope **globale**, di **funzione**, e di **blocco**
2. C'è un solo scope **globale**, ma gli scope di **funzione** e di **blocco** possono essere annidati uno nell'altro
3. Le dichiarazioni con **let** e **const** hanno scope di **blocco**, o di **funzione**, o **globale** (vale lo scope più interno in cui sono scritte)
4. Le dichiarazioni con **var** hanno scope di **funzione** o **globale** (vale lo scope più interno in cui sono scritte)
5. Le variabili non dichiarate hanno scope **globale**
6. Le dichiarazioni con **function** si comportano come **var**, ma si “portano dietro” l'inizializzazione in caso di *hoisting*

Closure

Function è un tipo di dato come tutti gli altri in Javascript

Possiamo usare una funzione in un'espressione o comando

Passare una funzione come parametro a un'altra funzione

Restituire una funzione da un'altra funzione

Closure: Lo scope attivo durante la dichiarazione della funzione rimane visibile alla funzione restituita anche se la funzione padre finisce

```
function f2(f){  
  return x => 2*f(x);  
}  
  
var myF=f2(Math.sqrt);  
  
myF(9); // -> 6
```

Q & A

Esercizi

Scrivere le seguenti funzioni:

- **unioneSenzaOutlier(A,B,epsilon)**: dati due insiemi di numeri, calcola la media di tutti i numeri in **A** e **B** e restituisce l'insieme contenente tutti gli elementi in **A** e **B** la cui distanza dalla media è inferiore a **epsilon**
- **immagine(f,D)**: Data una funzione **f** e un insieme **D**, restituisce l'insieme "immagine" di **f** per il dominio **D**, ovvero l'insieme $\{ f(x) \mid x \in D \}$
- **componi(f,g)**: Date due funzioni **f** e **g**, restituisce la funzione che calcola $f(g(x))$
- **poly(a,b,c)**: Dati tre numeri **a**, **b** e **c**, restituisce la funzione che calcola il polinomio $ax^2 + bx + c$