Laboratorio I
a.a. 2025/2026

L’identità segreta degli oggetti

Contenuti

- Valori e riferimenti
- Costruttori
- Funzioni e metodi
- Prototipi
- Classi

Valori e riferimenti

Abbiamo visto che JavaScript prevede valori di tipi base (es.: numeri, booleani, stringhe) e valori di tipi complessi (es.: array, oggetti, funzioni).

Nella maggior parte dei contesti, il valore effettivo a cui si fa riferimento tramite un identificatore non fa differenza.

Tuttavia, è importante ricordare che

- I tipi base hanno una **semantica per valore**
- I tipi complessi hanno una **semantica per riferimento**

Avete già incontrato questi termini a P&A!

Valori e riferimenti

Valori

Le variabili contengono direttamente il valore.

L'assegnamento copia il valore; l'uguaglianza confronta il valore.

```
var a=5
var b=5
var c=a

a==b → true
a==c → true
```

I numeri sono un tipo base: valori

Riferimenti

Le variabili contengono un riferimento a un'area della memoria dove è memorizzato il valore.

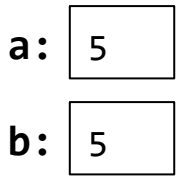
L'assegnamento copia il riferimento (non i contenuti dell'area di memoria); l'uguaglianza confronta il riferimento (non i contenuti dell'area di memoria).

```
var a=[1,2,3]
var b=[1,2,3]
var c=a

a==b → false
a==c → true
```

Gli array sono un tipo complesso: riferimenti

Valori e riferimenti

```
var a=5
var b=5
```

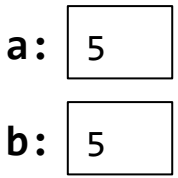



```
var c=a
```

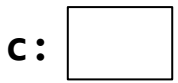
```
a==b → true
a==c → true
```

Valori e riferimenti

```
var a=5
var b=5
```



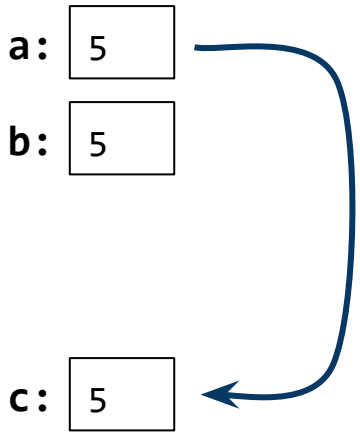

```
var c=a
```




```
a==b → true
a==c → true
```

Valori e riferimenti

```
var a=5
var b=5
```

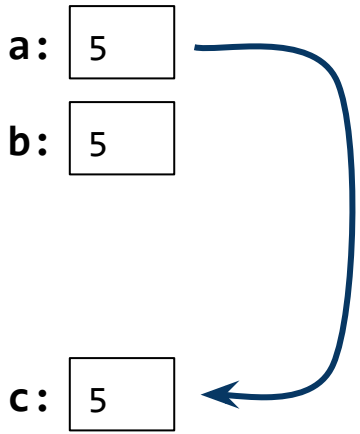



```
var c=a
```

```
a==b → true
a==c → true
```

Valori e riferimenti

```
var a=5
var b=5
```

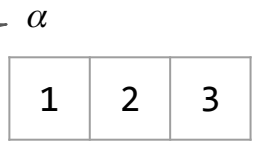



```
var c=a
```

```
a==b → true     (perché 5==5)
a==c → true     (perché 5==5)
```

Valori e riferimenti

α e β sono **riferimenti** (in pratica: indirizzi in memoria)




```
var a=5
var b=5
```

a: 5

b: 5

```
var c=a
```

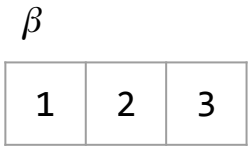
c: 5

```
a==b → true     (perché 5==5)
a==c → true     (perché 5==5)
```

```
var a=[1,2,3]
var b=[1,2,3]
```

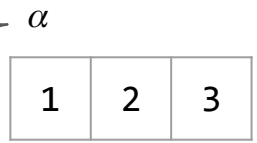
a:

b:



Valori e riferimenti

α e β sono **riferimenti** (in pratica: indirizzi in memoria)




```
var a=5
var b=5
```

a: 5

b: 5

```
var c=a
```

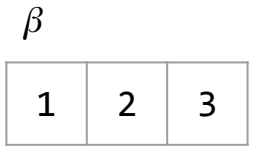
c: 5

```
a==b → true     (perché 5==5)
a==c → true     (perché 5==5)
```

```
var a=[1,2,3]
var b=[1,2,3]
```

a: ??

b: ??



Valori e riferimenti

```
var a=5
var b=5
```

a: 5

b: 5

```
var c=a
```

c: 5

```
a==b → true     (perché 5==5)
a==c → true     (perché 5==5)
```

α e β sono **riferimenti** (in pratica: indirizzi in memoria)

```
var a=[1,2,3]
var b=[1,2,3]
```

a: α

b: β

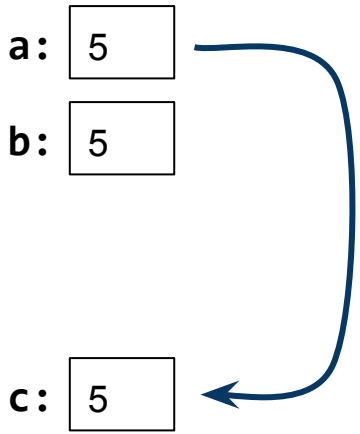
α → 1 | 2 | 3

β → 1 | 2 | 3

Ogni volta che valuto un letterale, si crea **una nuova copia** del valore

Valori e riferimenti

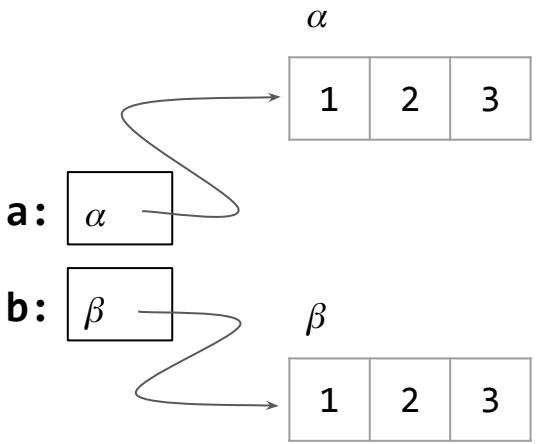
```
var a=5
var b=5
```




```
var c=a
```

```
a==b → true     (perché 5==5)
a==c → true     (perché 5==5)
```

```
var a=[1,2,3]
var b=[1,2,3]
```



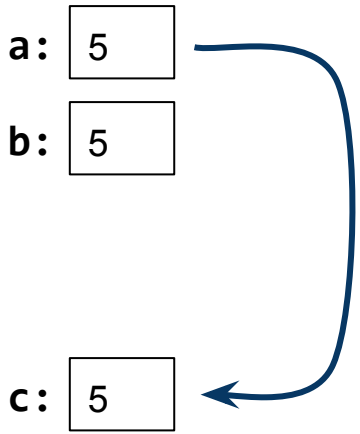

```
var c=a
```

c:

```
a==b → false
a==c → true
```

Valori e riferimenti

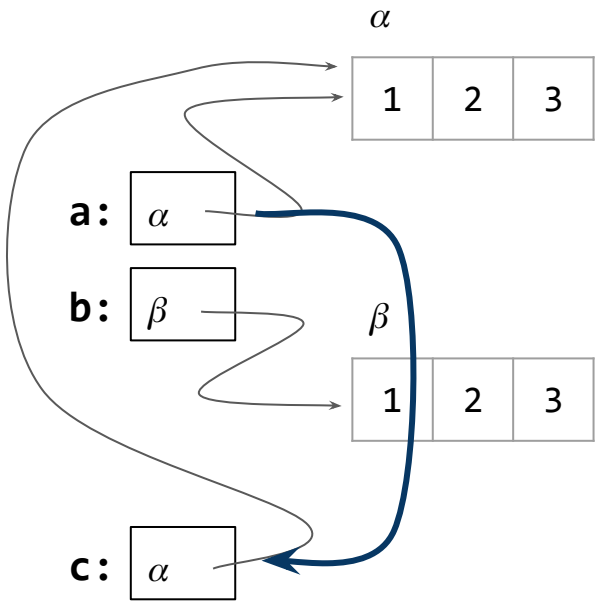
```
var a=5
var b=5
```




```
var c=a
```

```
a==b → true     (perché 5==5)
a==c → true     (perché 5==5)
```

```
var a=[1,2,3]
var b=[1,2,3]
```

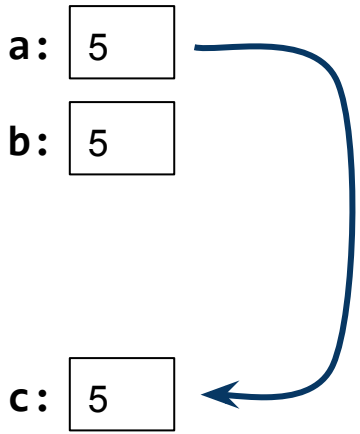



```
var c=a
```

```
a==b → false
a==c → true
```

Valori e riferimenti

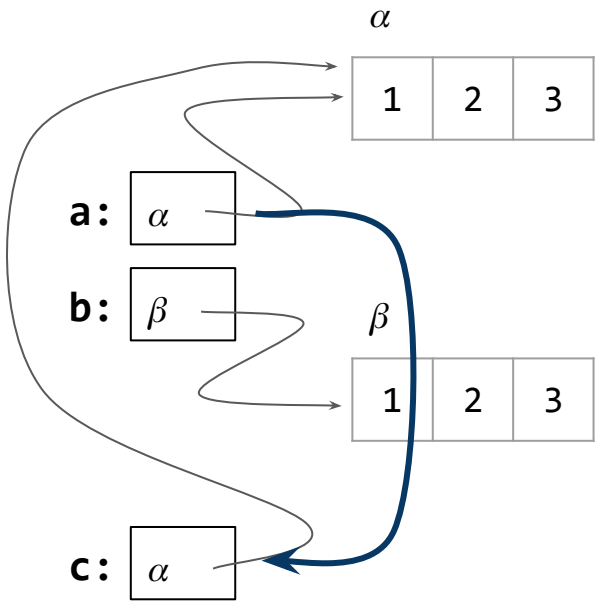
```
var a=5
var b=5
```




```
var c=a
```

```
a==b → true     (perché 5==5)
a==c → true     (perché 5==5)
```

```
var a=[1,2,3]
var b=[1,2,3]
```




```
var c=a
```

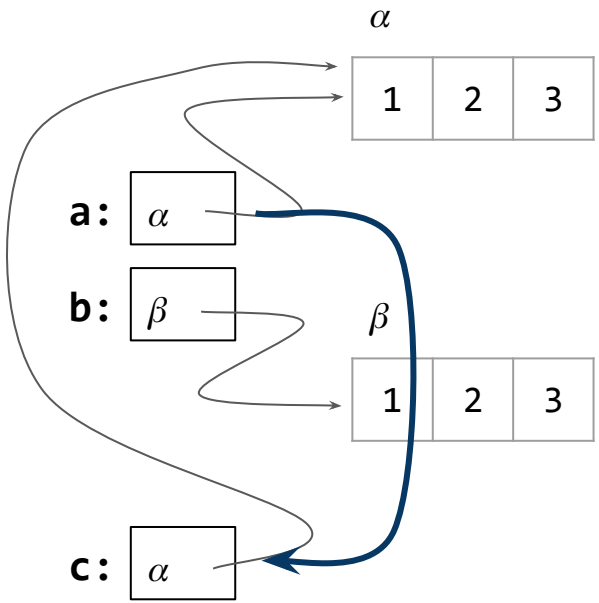
```
a==b → false    (perché α ≠ β)
a==c → true     (perché α = α)
```

Modifica di riferimenti

La semantica per riferimento permette la *condivisione* di oggetti in memoria. Modifiche apportate tramite un riferimento, sono visibili attraverso altri riferimenti allo stesso oggetto.

```
a → [1, 2, 3]
b → [1, 2, 3]
c → [1, 2, 3]

a[1]=0

a → [1, 0, 3]
b → [1, 2, 3]
c → [1, 0, 3]
```

```
var a=[1,2,3]
var b=[1,2,3]

var c=a
```




```
a==b → false    (perché α ≠ β)
a==c → true     (perché α = α)
```

Modifica di riferimenti

In particolare, questo è vero per gli **argomenti** passati alle funzioni.

JavaScript usa il **passaggio per valore** degli argomenti.
Per il corpo della funzione, il parametro formale è una variabile locale, inizializzata con una copia del valore del parametro attuale (oppure col valore di default, se indicato nella dichiarazione e se la chiamata non comprendeva quel parametro).

Questo implica che una funzione non può modificare il valore di una variabile usata nella sua invocazione.

Però se il valore è un riferimento, una funzione può modificare il valore di tipo complesso **riferito** dal riferimento!

Modifica di riferimenti

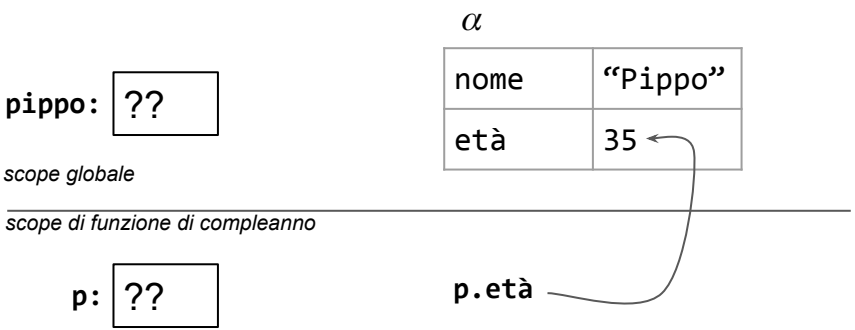
Esempio (modifica l'oggetto riferito)

```
function compleanno(p) {
  p.età++
}

var pippo= {nome: "Pippo", età: 35}
compleanno(pippo)
pippo → {nome: "Pippo", età: 36}
```

Modifica di riferimenti

Esempio (modifica l'oggetto riferito)

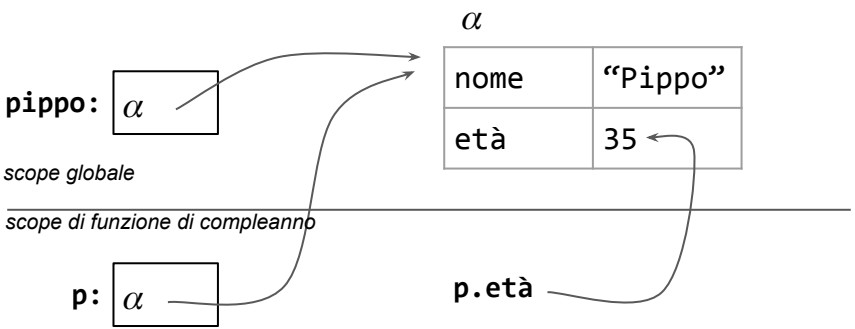
```
function compleanno(p) {
  p.età++
}

var pippo= {nome: "Pippo", età: 35}
compleanno(pippo)
pippo → {nome: "Pippo", età: 36}
```



Modifica di riferimenti

Esempio (modifica l'oggetto riferito)

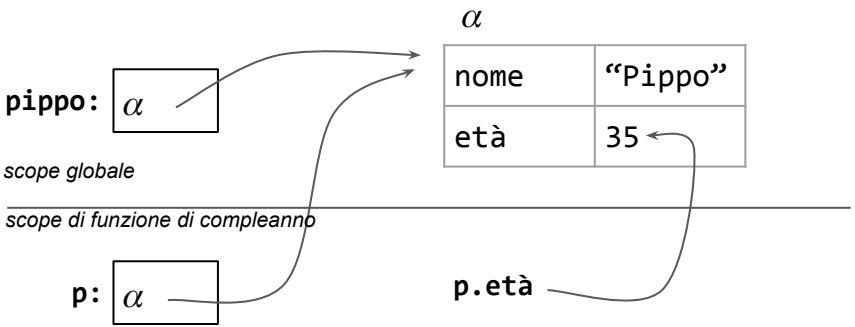
```
function compleanno(p) {
  p.età++
}

var pippo= {nome: "Pippo", età: 35}
compleanno(pippo)
pippo → {nome: "Pippo", età: 36}
```



Modifica di riferimenti

Esempio (modifica l'oggetto riferito)

```
function compleanno(p) {
  p.età++
}

var pippo= {nome: "Pippo", età: 35}
compleanno(pippo)
pippo → {nome: "Pippo", età: 36}
```



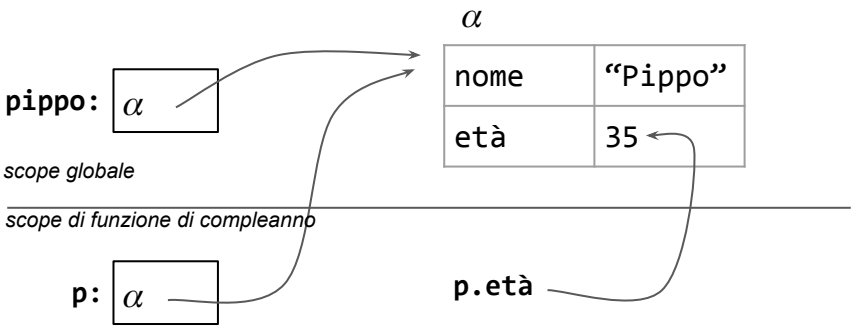
Esempio (modifica il parametro formale)

```
function compleanno(p) {
  p = {nome: p.nome, età: p.età+1}
}

var pippo= {nome: "Pippo", età: 35}
compleanno(pippo)
pippo → {nome: "Pippo", età: 35}
```

Modifica di riferimenti

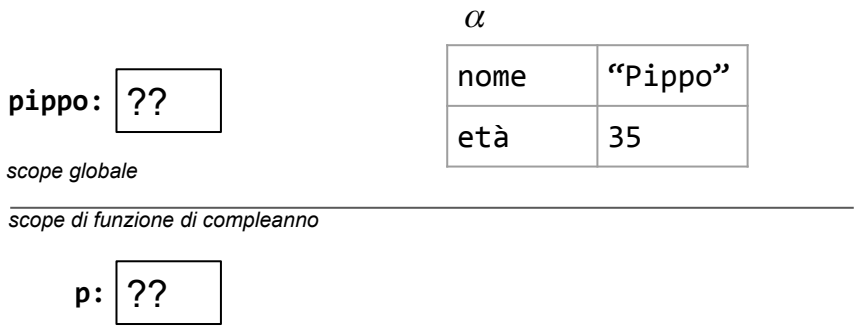
Esempio (modifica l'oggetto riferito)

```
function compleanno(p) {
  p.età++
}

var pippo= {nome: "Pippo", età: 35}
compleanno(pippo)
pippo → {nome: "Pippo", età: 36}
```



Esempio (modifica il parametro formale)

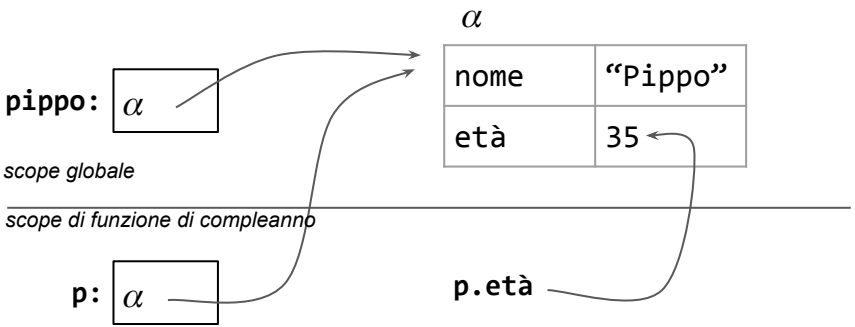
```
function compleanno(p) {
  p = {nome: p.nome, età: p.età+1}
}

var pippo= {nome: "Pippo", età: 35}
compleanno(pippo)
pippo → {nome: "Pippo", età: 35}
```



Modifica di riferimenti

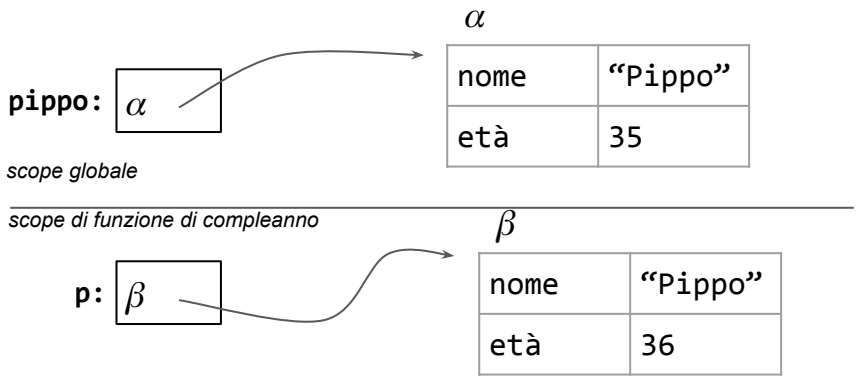
Esempio (modifica l'oggetto riferito)

```
function compleanno(p) {
  p.età++
}

var pippo= {nome: "Pippo", età: 35}
compleanno(pippo)
pippo → {nome: "Pippo", età: 36}
```



Esempio (modifica il parametro formale)

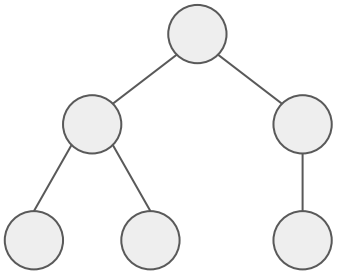
```
function compleanno(p) {
  p = {nome: p.nome, età: p.età+1}
}

var pippo= {nome: "Pippo", età: 35}
compleanno(pippo)
pippo → {nome: "Pippo", età: 35}
```



Riferimenti nelle strutture dati

Il concetto di **riferimento** si applica anche alle strutture dati.

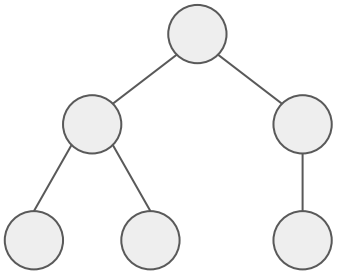
Finora abbiamo lavorato con strutture dati in cui ogni oggetto appariva **in un solo posto**.
Per esempio: gli alberi godono di questa proprietà, ogni nodo compare come valore solo del campo **`sx`** o del campo **`dx`** del padre (oppure: solo una volta nell'array **`figli`**, nel caso di alberi *k*-ari).



Riferimenti nelle strutture dati

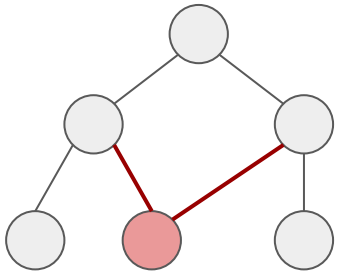
Il concetto di **riferimento** si applica anche alle strutture dati.

Finora abbiamo lavorato con strutture dati in cui ogni oggetto appariva **in un solo posto**.
Per esempio: gli alberi godono di questa proprietà, ogni nodo compare come valore solo del campo `sx` o del campo `dx` del padre (oppure: solo una volta nell'array `figli`, nel caso di alberi *k*-ari).



Una volta chiarito l'uso dei riferimenti, possiamo definire strutture dati in cui **lo stesso** oggetto compaia in più campi. **Per esempio: grafi**!

Importante: dire *lo stesso* oggetto non è come dire un oggetto *uguale*!
Nel primo caso si parla di **identità**, nel secondo di **uguaglianza**.



Un esempio: Grafi

Usiamo la definizione classica: $G = \langle N,E \rangle$ dove

N è un insieme di Nodi

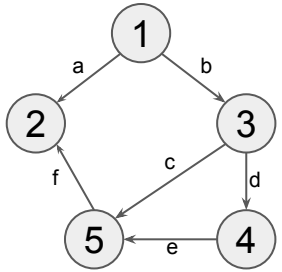
E è un insieme di archi, $E \subseteq N \times N$

Rappresentiamo un grafo come un oggetto con un campo nodi e un campo archi: **`{ nodi: N, archi: E}`**

Rappresentiamo un nodo come un oggetto (con un campo val per ospitare un valore o etichetta): **`{ val: x }`**

Rappresentiamo un arco come un oggetto con un campo **`da`** e un campo **`a`** (modelliamo un grafo diretto): **`{ da: `**n_1**`, a: `**n_2**` }`**

Un esempio: Grafi



Per esprimere il fatto che gli archi *b*, *c* e *d* insistono sullo stesso nodo 3 (proprio lui, non un possibile altro nodo etichettato 3), è indispensabile fare affidamento sui riferimenti.

La via più semplice è di usare delle variabili per riferire i vari nodi e archi:

```
var n1={val: 1}, n2={val: 2}, n3={val: 3}, n4={val: 4}, n5={val: 5}

var a={da: n1, a: n2}, b={da: n1, a: n3}, c={da: n3, a: n5},
    d={da: n3, a: n4}, e={da: n4, a: n5}, f={da: n5, a: n2}

var G={nodi: [n1, n2, n3, n4, n5], archi: [a, b, c, d, e, f]}
```

Un esempio di algoritmo su grafi

Vogliamo scoprire il *nodo di grado massimo* all'interno di un grafo.

```
function nodoGradoMassimo(G) {
  var nmax, gmax=0;
  for (var n of G.nodi) {
    var gn=0;
    for (var e of G.archi) {
      if (e.da == n) gn++;
      if (e.a == n) gn++;
    }
    if (gn>gmax) {
      gmax=gn;
      nmax=n;
    }
  }
  return nmax;
}
```

Qui vogliamo davvero ==

Restituiremo proprio **quel** nodo, non una copia

"Scusi prof! Ma questo algoritmo è

- ☒ poco efficiente
- ☒ non ricorsivo
- ☒ più lungo di una riga
- ☒ poco efficiente l'ha già detto qualcuno?
- ☒ pieno di punti e virgola
- ☒ altro (indicare sotto)

Come mai?"

Lo so, lo so… ma qui ci stiamo concentrando sui riferimenti, ok?

I costruttori

Quando si usano oggetti complicati, è usuale (e comodo) usare una funzione per **costruire** l'oggetto, spesso partendo da alcuni valori che lo descrivono.

Queste funzioni usano il fatto - che già conosciamo - che ogni volta che viene valutato un letterale di tipo oggetto, in effetti viene allocato un nuovo oggetto, con un suo riferimento diverso da tutti quelli già esistenti.

Spesso queste funzioni prevedono argomenti di default, oppure effettuano dei calcoli o dei controlli prima di inizializzare l'oggetto.

Chiamando sempre `arco(n1,n2)` siamo sicuri che **tutti** gli archi del nostro programma abbiano lo stesso formato.

```
function arco(n1,n2) {
  return {da: n1, a: n2}
}
```

```
function persona(n, e) {
  return {
        nome: n,
        età: e
  }
}
```

In un certo senso, la funzione usata per costruire un oggetto "cattura" l'idea di *tipo* dell'oggetto… 🤔

Proprietà di tipo funzione

Sappiamo da tempo che un oggetto può avere proprietà (ovvero: coppie `<chiave, valore>`) con valori di tipo funzione.

Lo sappiamo, vero? Scrivete un oggetto con una proprietà che ha come valore una funzione….

```
var o={
    inc: ????,
    dec: ????,
    ....
}
```

Proprietà di tipo funzione

Sappiamo da tempo che un oggetto può avere proprietà (ovvero: coppie `<chiave, valore>`) con valori di tipo funzione.

Conosciamo due modi di denotare un valore funzione: con `=>` o con **`function`**. Tuttavia, per il caso particolare di funzioni definite come proprietà di un oggetto, esiste una terza sintassi abbreviata:

```
var o={
    inc: x=>x+1,
    dec: function(x) { return x-1},
    zero(x) { return x-x}
}
```

Equivale a:
zero: x => { return x-x }

Proprietà di tipo funzione

Finora abbiamo usato con grande tranquillità funzioni “speciali” come **`console.log()`**, **`Math.max()`** e così via.

Secondo voi, cosa sono **`console`**, **`Math`** ecc. ?

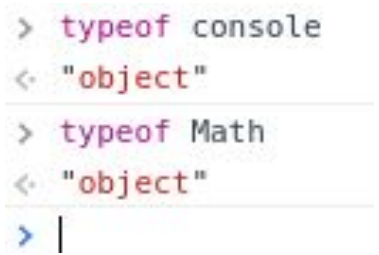
Proviamo a vedere cosa restituisce `typeof`….

Proprietà di tipo funzione

Finora abbiamo usato con grande tranquillità funzioni “speciali” come **`console.log()`**, **`Math.max()`** e così via.

Ora è chiaro di cosa si tratta: **`console`**, **`Math`** ecc. sono **variabili globali** *pre-dichiarate nell’ambiente di esecuzione*.

Si tratta di variabili di tipo oggetto, come possiamo verificare Facilmente. E le nostre funzioni speciali non sono altro che proprietà di questi oggetti, con valori di tipo funzione.



Naturalmente, anche gli oggetti definiti da noi possono avere proprietà di tipo funzione.

Proprietà di tipo funzione

```
> console.log(console)
▼ console {debug: ƒ, error: ƒ, info: ƒ, log: ƒ,
  ▶ assert: ƒ assert()
  ▶ clear: ƒ clear()
  ▶ context: ƒ context()
  ▶ count: ƒ count()
  ▶ countReset: ƒ countReset()
  ▶ debug: ƒ debug()
  ▶ dir: ƒ dir()
  ▶ dirxml: ƒ dirxml()
  ▶ error: ƒ error()
  ▶ group: ƒ group()
  ▶ groupCollapsed: ƒ groupCollapsed()
  ▶ groupEnd: ƒ groupEnd()
  ▶ info: ƒ info()
  ▶ log: ƒ log()
    memory: (...)
  ▶ profile: ƒ profile()
  ▶ profileEnd: ƒ profileEnd()
  ▶ table: ƒ table()
```

```
> console.log(Math)
▼ Math {abs: ƒ, acos: ƒ, acosh: ƒ, asin:
    E: 2.718281828459045
    LN2: 0.6931471805599453
    LN10: 2.302585092994046
    LOG2E: 1.4426950408889634
    LOG10E: 0.4342944819032518
    PI: 3.141592653589793
    SQRT1_2: 0.7071067811865476
    SQRT2: 1.4142135623730951
  ▶ abs: ƒ abs()
  ▶ acos: ƒ acos()
  ▶ acosh: ƒ acosh()
  ▶ asin: ƒ asin()
  ▶ asinh: ƒ asinh()
  ▶ atan: ƒ atan()
  ▶ atan2: ƒ atan2()
  ▶ atanh: ƒ atanh()
  ▶ cbrt: ƒ cbrt()
  ▶ ceil: ƒ ceil()
  ▶ clz32: ƒ clz32()
  ▶ cos: ƒ cos()
```

Proprietà di tipo funzione

Possiamo anche aggiungere le *nostre* funzioni a questi oggetti, come si fa normalmente per impostare proprietà di qualunque altro oggetto:

```
Math.fattoriale = n=>{ ... }

console.poly = p=>{ ... }

console.formula = f=>{ ... }
```

Queste nuove funzioni possono poi essere usate normalmente in tutto il resto del programma, esattamente come quelle predefinite

Questa cosa funziona molto bene per impressionare gli amici che usano linguaggi meno dinamici...

Funzioni e metodi

Le funzioni di `Math` sono *funzioni pure* — non hanno **effetti collaterali**, si limitano a calcolare un risultato che dipende solo dagli argomenti.

È però assai più comune il caso in cui una funzione (nel senso JavaScript, non matematico) debba **modificare** l'oggetto a cui si riferisce.

Esempio:

```
function compleanno(p) {
  p.età++
}
```

Potremmo anche assegnare questa funzione a una proprietà di un oggetto persona, per esempio

`pippo.compleanno = p=>{ p.età++}` …

Però poi come dovremmo scrivere per incrementare l'età????

Funzioni e metodi

Le funzioni di **`Math`** sono *funzioni pure* — non hanno **effetti collaterali**, si limitano a calcolare un risultato che dipende solo dagli argomenti.

È però assai più comune il caso in cui una funzione (nel senso JavaScript, non matematico) debba **modificare** l'oggetto a cui si riferisce.

Esempio:

```
function compleanno(p) {
  p.età++
}
```

Potremmo anche assegnare questa funzione a una proprietà di un oggetto persona, per esempio

`pippo.compleanno = p=>{ p.età++}` …

Però poi dovremmo scrivere codice come **`pippo.compleanno(pippo)`** per incrementare l'età!

Funzioni e metodi

JavaScript prevede che nel corpo di queste funzioni si possa usare una variabile speciale che contiene *un riferimento all'oggetto di cui la funzione è proprietà*:

`this`

In questo modo, possiamo scrivere

```
persona.compleanno = function(????) {????}
```

Quando una funzione è fornita come proprietà di un oggetto, e al suo interno riferisce l'oggetto, si usa chiamarla **metodo** dell'oggetto

Funzioni e metodi

JavaScript prevede che nel corpo di queste funzioni si possa usare una variabile speciale che contiene *un riferimento all'oggetto di cui la funzione è proprietà*:

`this`

In questo modo, possiamo scrivere

```
persona.compleanno = function() {this.età++}
```

E poi per invocare **`compleanno`**?

Quando una funzione è fornita come proprietà di un oggetto, e al suo interno riferisce l'oggetto, si usa chiamarla **metodo** dell'oggetto

Funzioni e metodi

JavaScript prevede che nel corpo di queste funzioni si possa usare una variabile speciale che contiene *un riferimento all'oggetto di cui la funzione è proprietà*:

`this`

In questo modo, possiamo scrivere

```
persona.compleanno = function() {this.età++}
```

E poi per invocare `compleanno`?

```
pippo.compleanno();   andrea.compleanno();
```

Quando una funzione è fornita come proprietà di un oggetto, e al suo interno riferisce l'oggetto, si usa chiamarla **metodo** dell'oggetto

Funzioni e metodi

In questo modo, possiamo scrivere

```
persona.compleanno = function() {this.età++}
```

E poi per invocare `compleanno`?

```
pippo.compleanno();   andrea.compleanno();
```

Ma come possiamo fare per essere sicuri che ***tutti*** gli oggetti di questo *tipo* abbiano il metodo **`compleanno()`** sempre definito? Invece quindi di *assegnarlo a mano* ogni volta?

Metodi e costruttori

È utile assicurarsi che *tutte* le persone abbiano un metodo compleanno, senza doverlo assegnare “a mano” tutte le volte che ci serve.

Fortunatamente, possiamo usare le funzioni costruttori per assicurarci che tutti gli oggetti di un certo *tipo* (in senso lasco) abbiano i metodi che ci servono.

Più proprietà e metodi vogliamo aggiungere a un oggetto, più è utile definire una funzione per assicurarsi che tutti gli oggetti dello stesso *tipo* siano costruiti esattamente allo stesso modo!

```
function persona(n, e) {
  return {
        nome: n,
        età: e,
        compleanno() {
            this.età++
        }
  }
}


var pippo=persona(“Pippo”, 35)
```

L’operatore **new**

L’operazione di creare un nuovo oggetto e assegnare le sue proprietà e i suoi metodi all’interno di una funzione costruttore, in modo da costruire oggetti tutti con la stessa struttura, **è molto comune**.

Tanto comune, in effetti, che JavaScript prevede un modo speciale per farla:
→ l’operatore `new`.

L’operatore **new**

L’operazione di creare un nuovo oggetto e assegnare le sue proprietà e i suoi metodi all’interno di una funzione costruttore, in modo da costruire oggetti tutti con la stessa struttura, **è molto comune**.

Tanto comune, in effetti, che JavaScript prevede un modo speciale per farla:
→ l’operatore **new**.

```
var pippo = new Persona(“Pippo”, 35)
```

L’operatore `new` **crea** un nuovo oggetto vuoto, **chiama** la funzione costruttore passando come `this` l’oggetto vuoto appena creato, più i parametri indicati, e **restituisce** l’oggetto inizializzato

```
function Persona(n, e) {
  this.nome=n
  this.età=e
  this.compleanno=function() {
            this.età++
        }
}
```

Non c’è più un letterale di tipo oggetto per creare il nuovo oggetto: la creazione viene fatta da **new**

Non c’è un **return**: **new** implicitamente restituisce l’oggetto creato da **new** e inizializzato da `Persona()`

L'operatore **new**

L'operazione di creare un nuovo oggetto e assegnare le sue proprietà e i suoi metodi all'interno di una funzione costruttore, in modo da costruire oggetti tutti con la stessa struttura, **è molto comune**.

Tanto comune, in effetti, che JavaScript prevede un modo speciale per farla:
→ l'operatore **new**.

```
var pippo = new Persona("Pippo", 35)
```

L'operatore `new` **crea** un nuovo oggetto vuoto, **chiama** la funzione costruttore passando come `this` l'oggetto vuoto appena creato, più i parametri indicati, e **restituisce** l'oggetto inizializzato

Convenzione: le funzioni-costruttore destinate a essere usate con **new** iniziano per lettera maiuscola.

```
function Persona(n, e) {
  this.nome=n
  this.età=e
  this.compleanno=function() {
            this.età++
        }
}
```

Non c'è più un letterale di tipo oggetto per creare il nuovo oggetto: la creazione viene fatta da **new**

Non c'è un **return**: **new** implicitamente restituisce l'oggetto creato da **new** e inizializzato da `Persona()`

Il mistero del metodo mancante

Quanto abbiamo visto finora spiega come sono realizzate molte delle funzioni predefinite di JavaScript… ma non tutte!

- `"Nel mezzo del cammin di nostra vita".split(" ")`
- `var a=[1, 2, 3]; a.push(4);`

Se guardiamo fra le proprietà di **a**, non troviamo il metodo **push()**. Allo stesso modo, se frughiamo all'interno di una stringa, non troviamo il metodo **split()**...

I prototipi

In realtà, il meccanismo con cui JavaScript accede alle proprietà di un oggetto è più complesso di come l'abbiamo usato finora.

1. Ogni oggetto ha un **prototipo** (che è un altro oggetto), tranne il prototipo dell'oggetto `Object`, che non ha prototipo.
2. Quando si vuole leggere il valore di una proprietà di un oggetto, si guarda se l'oggetto ha la chiave cercata.
   a. Se la chiave è presente, il valore è quello della chiave nell'oggetto
   b. Se la chiave non è presente, e l'oggetto ha un prototipo, si cerca la proprietà nel prototipo
   c. Se la chiave non è presente, e l'oggetto non ha un prototipo, il risultato è `undefined`
3. Quando si vuole scrivere il valore di una proprietà di un oggetto, la chiave e il valore vengono inseriti nell'oggetto (eventualmente sovrascrivendo il valore precedente)

Il mistero del metodo mancante

Ecco dove stavano i metodi mancanti: nella *catena dei prototipi* dei nostri oggetti!

Possiamo scoprire chi è il prototipo di un oggetto *o* accedendo alla sua proprietà "speciale" **`o.__proto__`** (scritta con due _ prefissi e due suffissi), oppure invocando **`Object.getPrototypeOf(o)`**

Provate a guardare come sono fatti i prototipi di **`"a"`**, di **`3`**, di **`{a:1}`** o di **`()=>0`**

pa è l'oggetto prototipo di tutti gli array

Q & A