Laboratorio I
a.a. 2025/2026

L’identità segreta degli oggetti (cont.)

Contenuti

- Nelle puntate precedenti...
  - Valori e riferimenti
  - Costruttori
  - Funzioni e metodi
- Prototipi
- Classi
- Esempi

Valori e riferimenti

In JavaScript,

- **Tipi base** (booleani, numeri, stringhe) → semantica **per valore**
- **Tipi complessi** (oggetti, array, funzioni) → semantica **per riferimento**

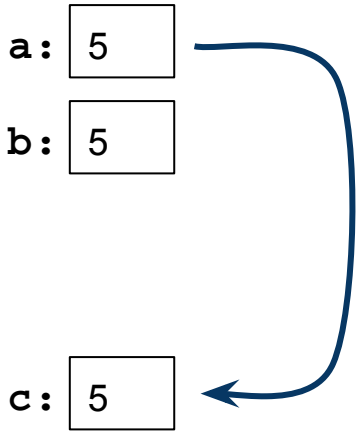
La semantica per riferimento consiste essenzialmente nell'usare **indirizzi in memoria** come valori, e nell'avere operazioni di indirezione (in JavaScript, sempre implicita) per accedere ai valori contenuti nella memoria all'indirizzo dato.

Alcuni esempi di operazioni di indirezione:

- oggetto.chiave — va in memoria a cercare la chiave e restituisce il contenuto
- oggetto[chiave] — come sopra
- array[indice] — come sopra
- funzione(argomenti) — va in memoria a recuperare la definizione della funzione e inizia a eseguirla

Valori e riferimenti

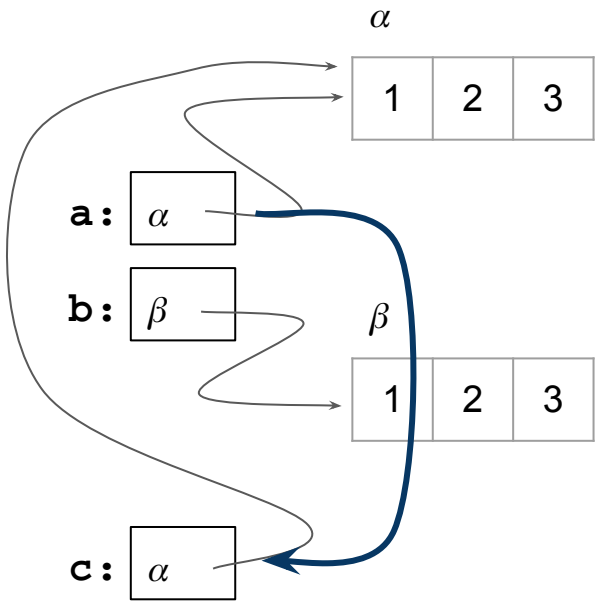
```
var a=5
var b=5
```




```
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]
```



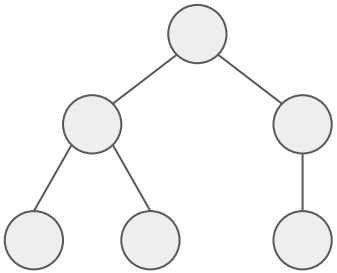

```
var c=a
```

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

Riferimenti nelle strutture dati

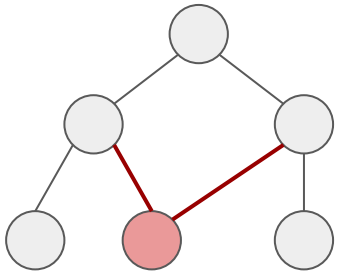
Il concetto di **riferimento** si applica anche alle strutture dati.

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

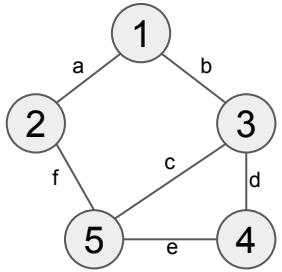
E è un insieme di archi, $E \subseteq N \times N$

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]}
```

Costruttori e metodi

- È comodo far **costruire** gli oggetti, tutte le volte che serve, da una sola funzione (così siamo sicuri che abbiano sempre i campi giusti)
  - Queste funzioni sono dette **costruttori**
- È anche comodo che un oggetto abbia delle proprietà con valori di tipo funzione (anzi, in genere ne avrà parecchie!)
- Queste funzioni possono poi essere chiamate con *oggetto*.*nome*(*argomenti*)
  - Le funzioni memorizzate all'interno degli oggetti sono dette **metodi**
  - Quando una funzione è invocata tramite una proprietà di un oggetto, ha a disposizione un parametro in più, implicito, che si chiama **this** e contiene un riferimento all'oggetto su cui è stata invocata

Esempio

Parametri di default

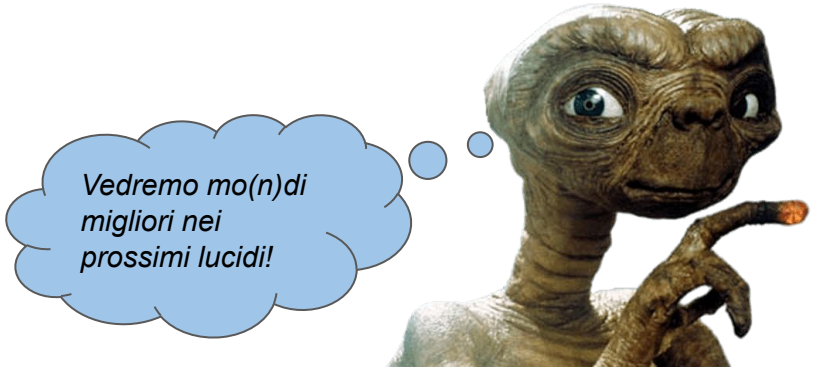
```
function grafo(N=[], E=[]) {
  let grafo={}
  grafo.nodi=N
  grafo.archi=E
  grafo.size = ()=>grafo.nodi.length
  grafo.figli = (n)=>grafo.archi.filter(({da})=>da==n).map(({a})=>a)
  grafo.fan = (n)=>grafo.archi.filter(({da,a})=>(da==n||a==n))
  return grafo
}
```

Letterale: alloca un nuovo oggetto

Argomento con destrutturazione

Restituisce il nuovo oggetto con i campi `nodi`, `archi`, `size`, `figli`, `fan`

- Attenzione! Si tratta di un **esempio** ma non è il modo migliore di fare le cose!



Esempio (alternativa più breve)

```
function grafo(N=[], E=[]) {
  return {
    nodi:N,
    archi:E,
    size() {return this.nodi.length},
    figli(n) {return this.archi.filter(({da})=>da==n).map(({a})=>a)},
    fan(n) {return this.archi.filter(({da,a})=>(da==n||a==n))}
  }
}

var G=grafo([n1, n2, n3, n4, n5], [a, b, c, d, e, f])
```

- Attenzione! Ancora <u>non</u> è il modo migliore di fare le cose!

L’operatore new (primo sguardo)

L’operatore **`new`**, premesso a una chiamata di funzione costruttore, consente di esprimere il costruttore usando **this** anziché un nuovo oggetto

```
var G = new Grafo([n1, n2, n3, n4, n5], [a, b, c, d, e, f])
```

Ricordiamo il funzionamento di **`new`**:

1. Alloca un nuovo oggetto (vuoto), chiamiamolo *o*
2. Invoca la funzione Grafo passando gli argomenti e implicitamente **`this`**= *o*
   a. La funzione Grafo userà this per arricchire *o* aggiungendo metodi e altre proprietà
3. Restituisce *o*
   a. Nel nostro caso, *o* viene poi assegnato a G, che da questo momento “è” un grafo

I prototipi

Il meccanismo dei **prototipi** spiega come mai alcuni oggetti sembrano avere tante proprietà (specialmente metodi) che *non sono "dentro" l'oggetto*.

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)

I prototipi

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`

Provate anche a guardare `o.__proto__.__proto__` e così via…

`pa` è l'oggetto prototipo di tutti gli array

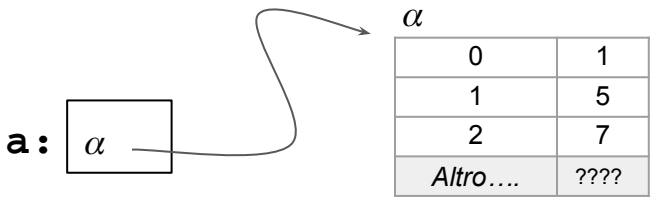
```
> var pa=[1,2,3].__proto__
undefined
> pa
▼[constructor: ƒ, concat: ƒ, copyWithin: ƒ, fill: ƒ,
  ▶concat: ƒ concat()
  ▶constructor: ƒ Array()
  ▶copyWithin: ƒ copyWithin()
  ▶entries: ƒ entries()
  ▶every: ƒ every()
  ▶fill: ƒ fill()
  ▶filter: ƒ filter()
  ▶find: ƒ find()
  ▶findIndex: ƒ findIndex()
  ▶flat: ƒ flat()
  ▶flatMap: ƒ flatMap()
  ▶forEach: ƒ forEach()
  ▶includes: ƒ includes()
  ▶indexOf: ƒ indexOf()
  ▶join: ƒ join()
  ▶keys: ƒ keys()
  ▶lastIndexOf: ƒ lastIndexOf()
   length: 0
  ▶map: ƒ map()
  ▶pop: ƒ pop()
  ▶push: ƒ push()
  ▶reduce: ƒ reduce()
  ▶reduceRight: ƒ reduceRight()
  ▶reverse: ƒ reverse()
  ▶shift: ƒ shift()
```

Nota: ***__proto__ non è standard*** (è *deprecato*, ma resta per retro-compatibilità);

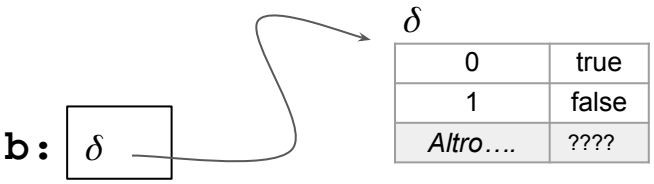
Object.getPrototypeOf() è standard.

La catena dei prototipi

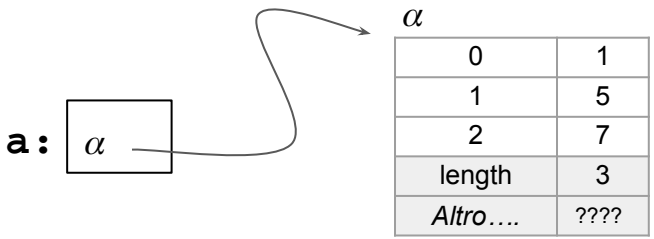
```
var a=[1,5,7]
```



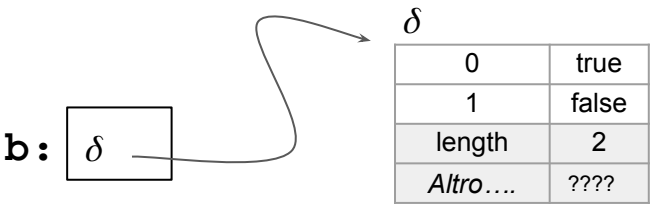

```
var b=[true,false]
```



La catena dei prototipi

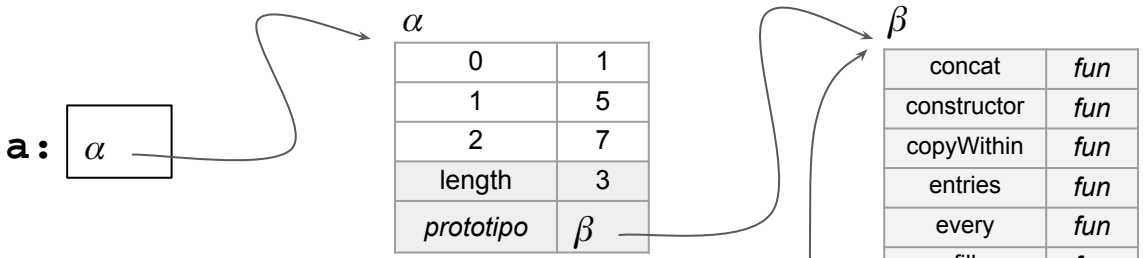



```
var b=[true,false]
```

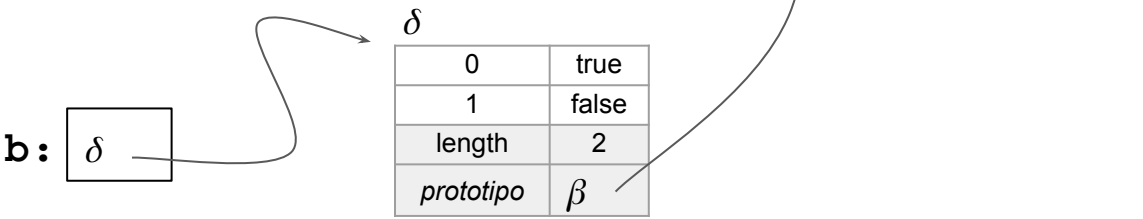


La catena dei prototipi

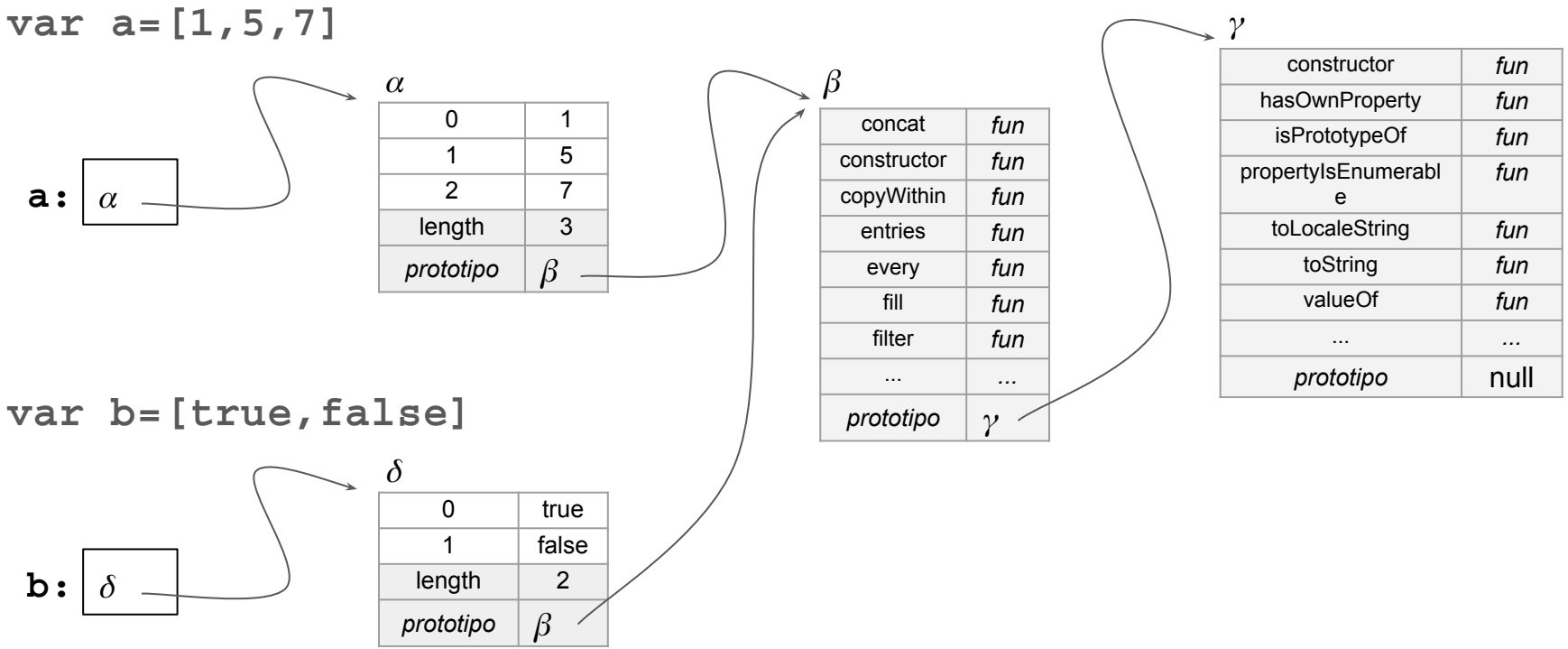
```
var a=[1,5,7]
```




```
var b=[true,false]
```



La catena dei prototipi



Prototipi ed enumerazione

Notate che l'enumerazione delle proprietà di un oggetto, fatta con

```
for (k in o) { … }
```

Restituisce solo le *chiavi proprie* dell'oggetto, quindi non quelle che vengono trovate nei prototipi.

Lo stesso vale per *Object.keys(o)*, *Object.entries(o)*, ecc.

In questo modo, le chiavi presenti nel prototipo sono leggibili se le accedete, ma non sono enumerabili (ciò è particolarmente comodo per array e dizionari)

Prototipi e costruttori

Osservazione

Se aggiungiamo un metodo a un particolare oggetto (diciamo, *pippo*), il metodo sarà disponibile solo per quell'oggetto.

Se aggiungiamo un metodo al *prototipo* di un oggetto, (diciamo, *Persona*), il metodo sarà disponibile per tutti gli oggetti che hanno lo stesso prototipo.

Prototipi e costruttori

Doppio salto carpiato

Tutte le funzioni (e in particolare: le **funzioni costruttore**) hanno una proprietà che si chiama `prototype`, inizializzata automaticamente dal linguaggio quando si dichiara una funzione, che contiene un oggetto pronto per fare da prototipo per gli oggetti inizializzati dalla funzione.

L'operatore `new` assegna automaticamente come prototipo dell'oggetto creato, il `prototype` della sua funzione costruttore.

Ciò crea un vero **legame permanente** tra gli oggetti e i loro costruttori… praticamente un tipo!

Prototipi e costruttori

Doppio salto carpiato

Tutte le funzioni (e in particolare: le **funzioni costruttore**) hanno una proprietà che si chiama **`prototype`**, inizializzata automaticamente dal linguaggio quando si dichiara una funzione, che contiene un oggetto pronto per fare da prototipo per gli oggetti inizializzati dalla funzione.

L'operatore **`new`** assegna automaticamente come prototipo dell'oggetto creato, il **`prototype`** della sua funzione costruttore.

Ciò crea un vero **legame permanente** tra gli oggetti e i loro costruttori… praticamente un tipo!

Definiamo un costruttore

```
> function Persona(n,e) { this.nome=n; this.età=e}
<· undefined
> Persona.prototype
<· ▶ {constructor: ƒ}
> var pippo=new Persona("Pippo",35)
<· undefined
> pippo.__proto__
<· ▶ {constructor: ƒ}
> pippo.__proto__ == Persona.prototype
<· true
>
```

Bene; il prototipo di pippo è il valore del prototype del suo costruttore!

Prototipi e costruttori

Questa caratteristica suggerisce un modo diverso per definire costruttori in JavaScript, che lo avvicina ad altri linguaggi (quelli basati su classi):

1. Aggiungiamo le proprietà che descrivono lo stato di un oggetto **all'oggetto** creato
2. Aggiungiamo i metodi che descrivono il suo comportamento **al prototipo della funzione costruttore**

In questo modo, i *campi* dell'oggetto sono enumerabili, i suoi metodi no.

Esempi di costruttore con prototipo

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

var pippo=new Persona(“Pippo”, 35)
pippo.compleanno()
pippo → {nome: “Pippo”, età: 36}
```

Un mondo migliore è possibile

Le versioni più recenti di JavaScript hanno aggiunto un modo diverso di **dichiarare** le **classi**

Si tratta comunque ***solo*** *di sintassi* più facile: in realtà, tutta la componente **orientata agli oggetti** di JavaScript è basata sul concetto di prototipo

1. Gli oggetti sono creati da `new` tramite funzioni-costruttori
2. Le funzioni-costruttori definiscono implicitamente il prototipo di ogni oggetto creato
3. Tutte le parti "**a comune**" di oggetti della stessa famiglia vengono implementate dai prototipi; le parti "**proprie**" sono implementate da ogni oggetto individualmente

La dichiarazione `class`

```
class Persona {
  constructor(nome,età) {
    this.nome=nome
    this.età=età
  }

  compleanno() {
    this.età++
  }
}

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

La sintassi con la dichiarazione `class` consente di definire funzione-costruttore e metodi di un oggetto di particolare tipo in maniera molto compatta.

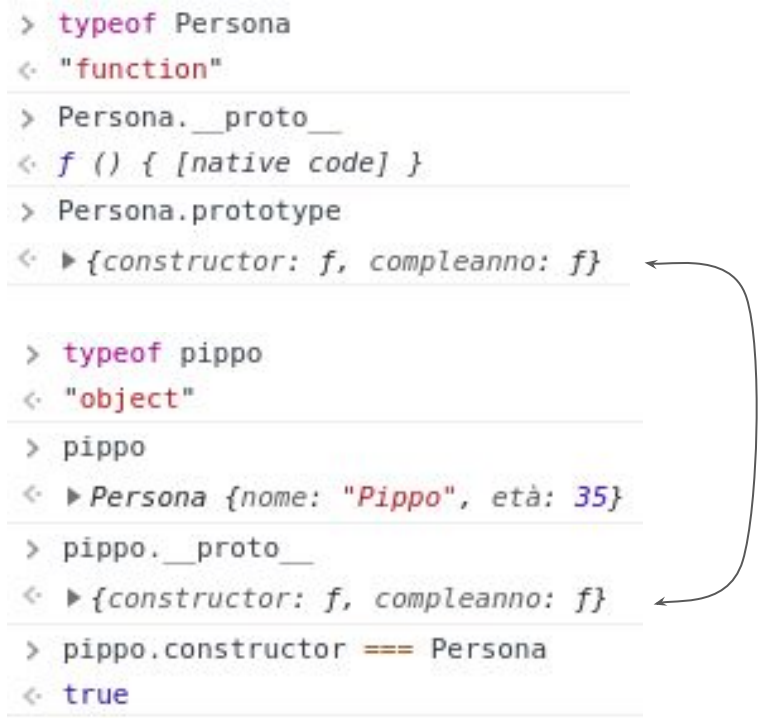
All'interno dei metodi (incluso il costruttore), l'oggetto che viene manipolato (o creato) è riferito da this.

Tecnicamente, `Persona` **non è** una "classe" (**un concetto che non esiste in JavaScript**), ma una **funzione**, come abbiamo visto in precedenza.

La creazione di nuovi oggetti avviene dunque con `new` come già sappiamo.

La dichiarazione `class`

```
class Persona {
  constructor(nome,età) {
    this.nome=nome
    this.età=età
  }

  compleanno() {
    this.età++
  }
}

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



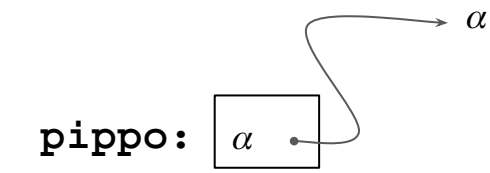
La dichiarazione `class`

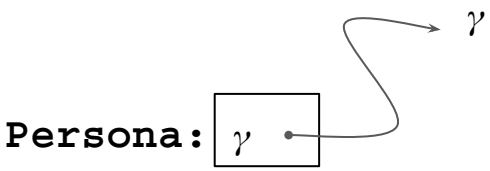
```
class Persona {
  constructor(nome,età
    this.nome=nome
    this.età=età
  }

  compleanno() {
    this.età++
  }
}

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

La funzione di stampa è abbastanza furba da riconoscere che **`pippo`** non è un semplice dizionario, ma è stato creato da **`Persona`** (lo vede da **`constructor`**), e quindi stampa anche il nome della classe davanti al contenuto del dizionario.

```
> typeof Persona
"function"
> Persona.__proto__
ƒ () { [native code] }
> Persona.prototype
▶ {constructor: ƒ, compleanno: ƒ}

> typeof pippo
"object"
> pippo
▶ Persona {nome: "Pippo", età: 35}
> pippo.__proto__
▶ {constructor: ƒ, compleanno: ƒ}
> pippo.constructor === Persona
true
```

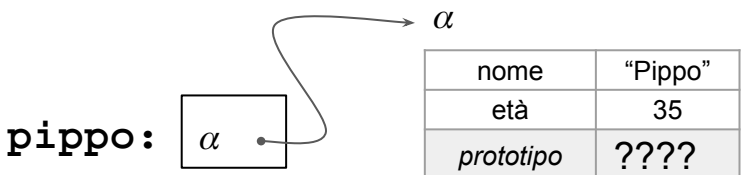
La dichiarazione `class`

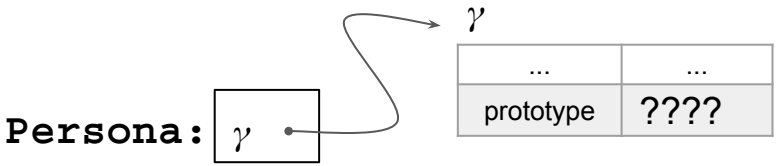





```
> typeof Persona
"function"
> Persona.__proto__
ƒ () { [native code] }
> Persona.prototype
▶ {constructor: ƒ, compleanno: ƒ}

> typeof pippo
"object"
> pippo
▶ Persona {nome: "Pippo", età: 35}
> pippo.__proto__
▶ {constructor: ƒ, compleanno: ƒ}
> pippo.constructor === Persona
true
```

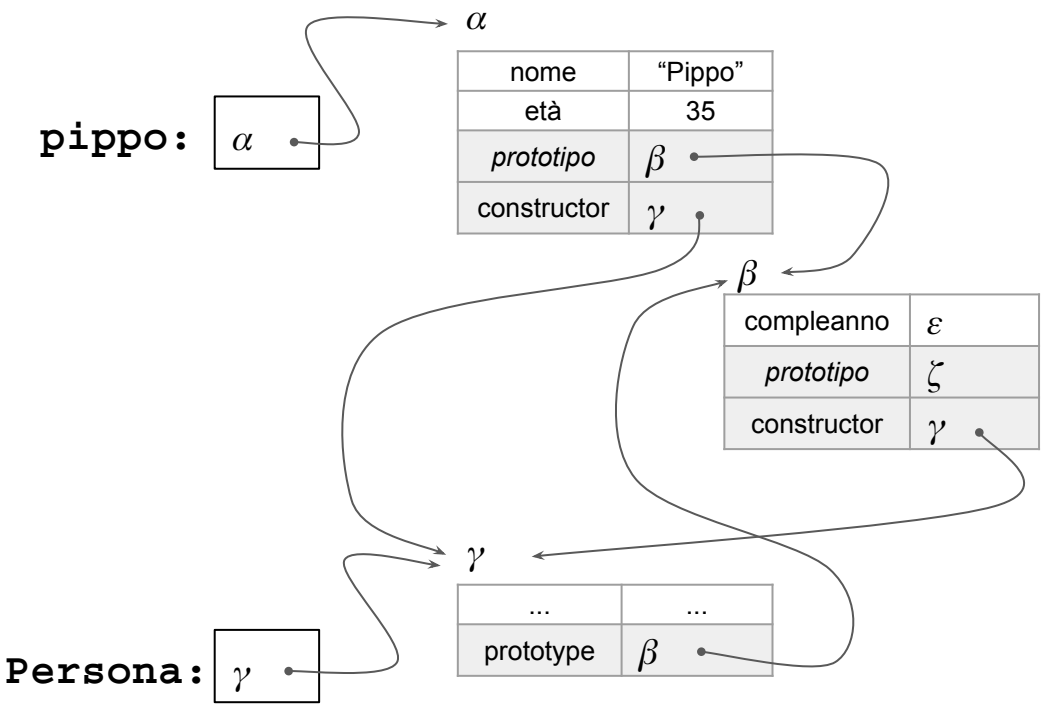
La dichiarazione `class`






```
> typeof Persona
< "function"
> Persona.__proto__
< ƒ () { [native code] }
> Persona.prototype
< ▶ {constructor: ƒ, compleanno: ƒ}

> typeof pippo
< "object"
> pippo
< ▶ Persona {nome: "Pippo", età: 35}
> pippo.__proto__
< ▶ {constructor: ƒ, compleanno: ƒ}
> pippo.constructor === Persona
< true
```

La dichiarazione `class`




```
> typeof Persona
"function"
> Persona.__proto__
ƒ () { [native code] }
> Persona.prototype
▶ {constructor: ƒ, compleanno: ƒ}

> typeof pippo
"object"
> pippo
▶ Persona {nome: "Pippo", età: 35}
> pippo.__proto__
▶ {constructor: ƒ, compleanno: ƒ}
> pippo.constructor === Persona
true
```

LUCIDO LASCIATO INTENZIONALMENTE VUOTO PER PERMETTERVI DI RIFLETTERE SUL MOMENTO IN CUI AVETE SCELTO DI FARE INFORMATICA… :)

La programmazione Object-Oriented

Ora abbiamo (quasi) tutti gli strumenti per cominciare a scrivere programmi “seri”.

Nelle prossime lezioni approfondiremo questi concetti

- Esempi ed esercizi
- Metodi statici e di istanza
- Ereditarietà
- Controllo statico dei tipi → TypeScript

Q & A

La catena dei prototipi