Laboratorio I
a.a. 2025/2026

Valori e riferimenti
Liste concatenate

Contenuti

- Valori e riferimenti
- Liste concatenate
  - Rappresentazione di liste tramite oggetti
  - Operazioni di base su liste
  - Esercizi

Valori e riferimenti

In JavaScript esistono **tipi di dato primitivi** (es. numeri, stringhe, booleani) e **tipi complessi** (es. oggetti, array, funzioni)

Quando si assegnano variabili, si passano parametri a una funzione o si restituiscono valori, è importante ricordare che:

- I **tipi primitivi** 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.

```
let a=5
let b=5
let c=a

a==b → true
a==c → true
```

I numeri sono un tipo primitivo: valori

Riferimenti

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

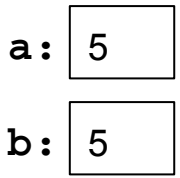
L'assegnamento copia il riferimento (non i contenuti dell'area di memoria); l'uguaglianza confronta il riferimento (non i contenuti dell'area di memoria).

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

a==b → false
a==c → true
```

Gli array sono un tipo complesso: riferimenti

Valori e riferimenti

```
let a=5
let b=5
```

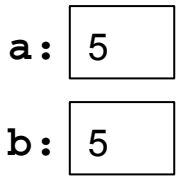



```
let c=a
```

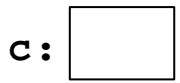
```
a==b → true
a==c → true
```

Valori e riferimenti

```
let a=5
let b=5
```



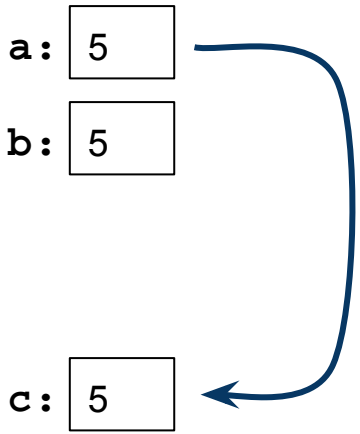

```
let c=a
```




```
a==b → true
a==c → true
```

Valori e riferimenti

```
let a=5
let b=5
```

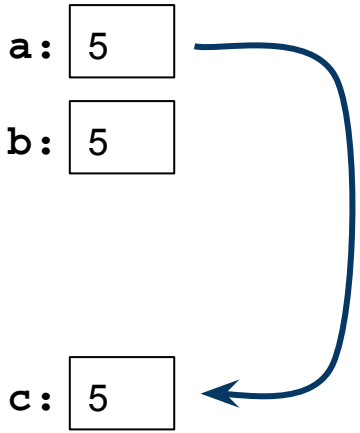



```
let c=a
```

```
a==b -> true
a==c -> true
```

Valori e riferimenti

```
let a=5
let b=5
```




```
let c=a
```

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

Valori e riferimenti

```
let a=5
let b=5
```

a: 5

b: 5

```
let 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)

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

a: α

b: β

α → 1 | 2 | 3

β → 1 | 2 | 3

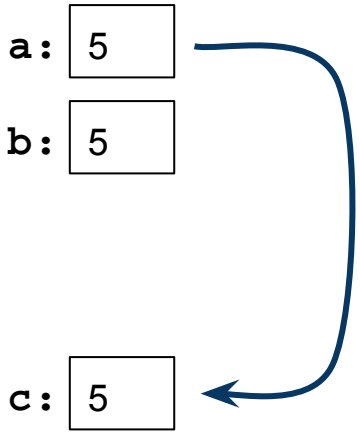
Ogni volta che valuto un letterale, si crea **una nuova copia** del valore

```
let c=a
```

```
a==b → false
a==c → true
```

Valori e riferimenti

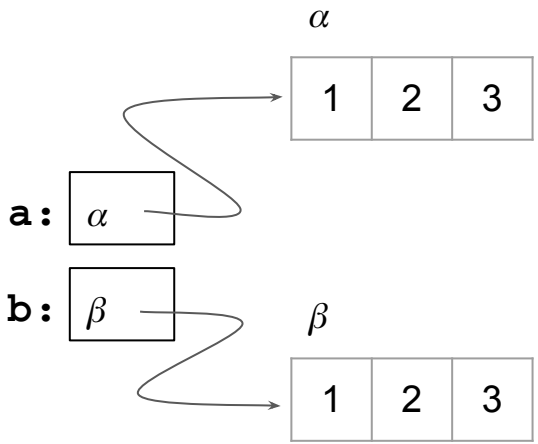
```
let a=5
let b=5
```




```
let c=a
```

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

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



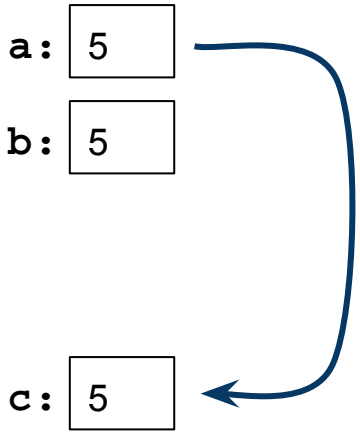

```
let c=a
```

c:

```
a==b → false
a==c → true
```

Valori e riferimenti

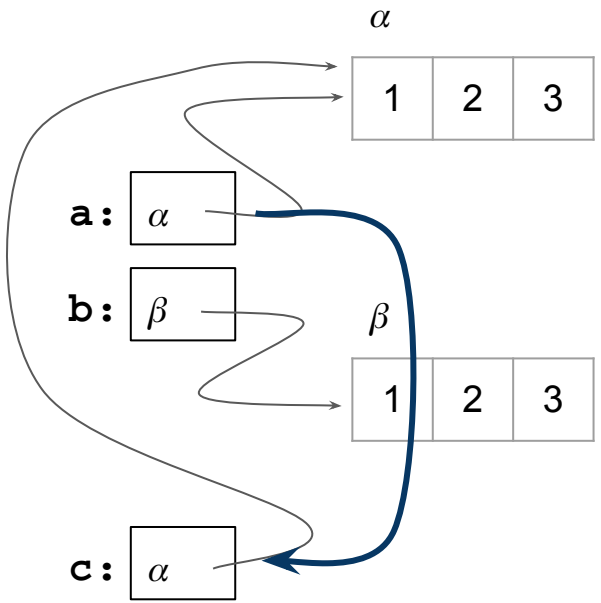
```
let a=5
let b=5
```




```
let c=a
```

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

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

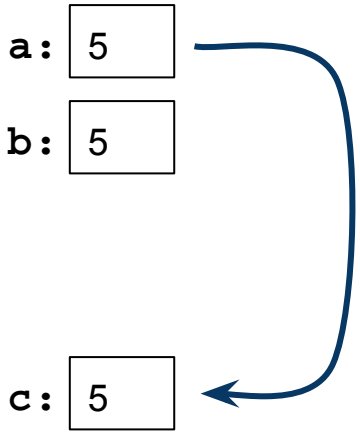



```
let c=a
```

```
a==b → false
a==c → true
```

Valori e riferimenti

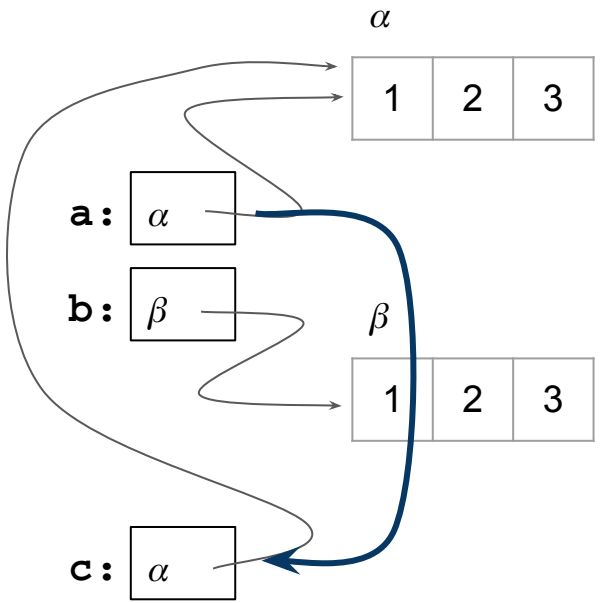
```
let a=5
let b=5
```




```
let c=a
```

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

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




```
let c=a
```

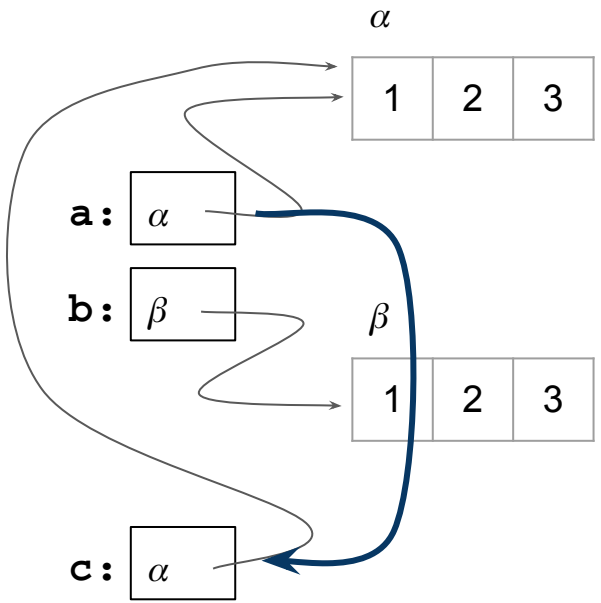
```
a==b → false    (perché α≠β)
a==c → true     (perché α=α)
```

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]
```

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

let 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: ogni parametro formale è una variabile locale, inizializzata con una copia del parametro attuale

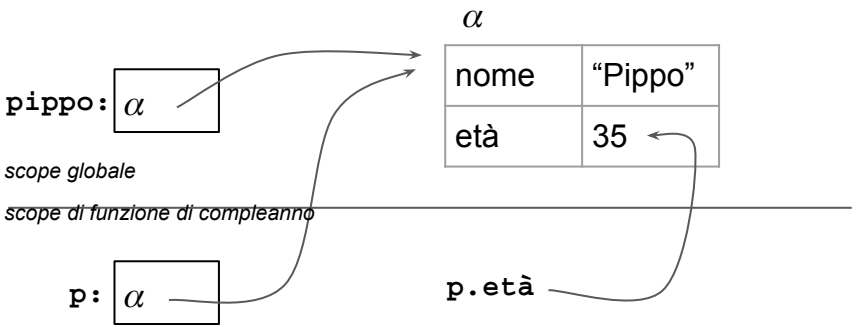
Questo significa che:

- Una funzione non può modificare direttamente la variabile usata nell'invocazione
- Tuttavia, **se il valore passato è un riferimento**, la funzione può modificare l'oggetto o l'array a cui quel riferimento punta

Modifica di riferimenti

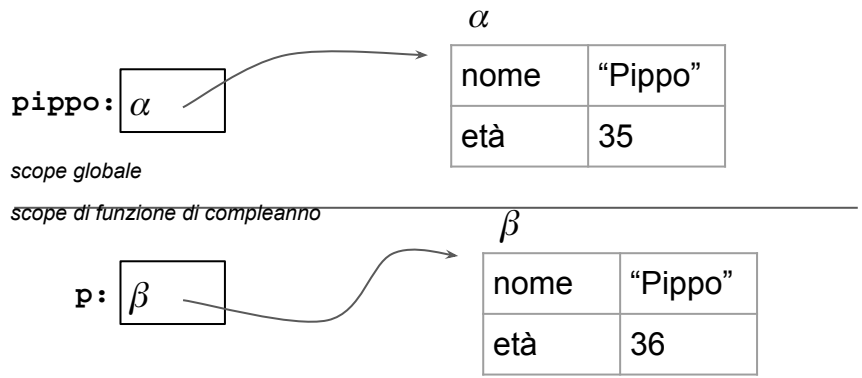
Esempio (modifica l'oggetto riferito)

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

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



Esempio (modifica il parametro formale)

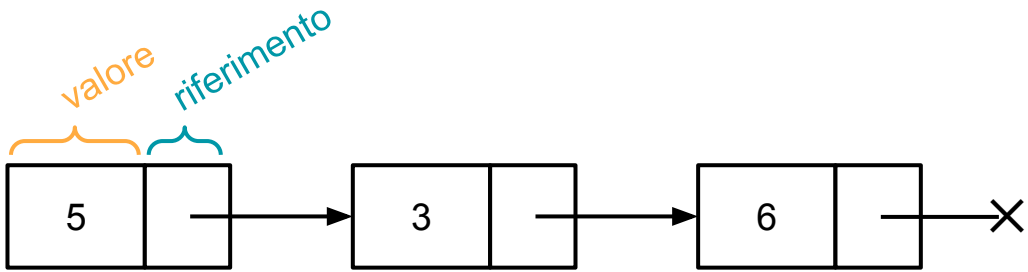
```
function compleanno(p) {
  p = {nome: p.nome, età: p.età+1}
}

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



Liste concatenate

Struttura dati ricorsiva che può essere:

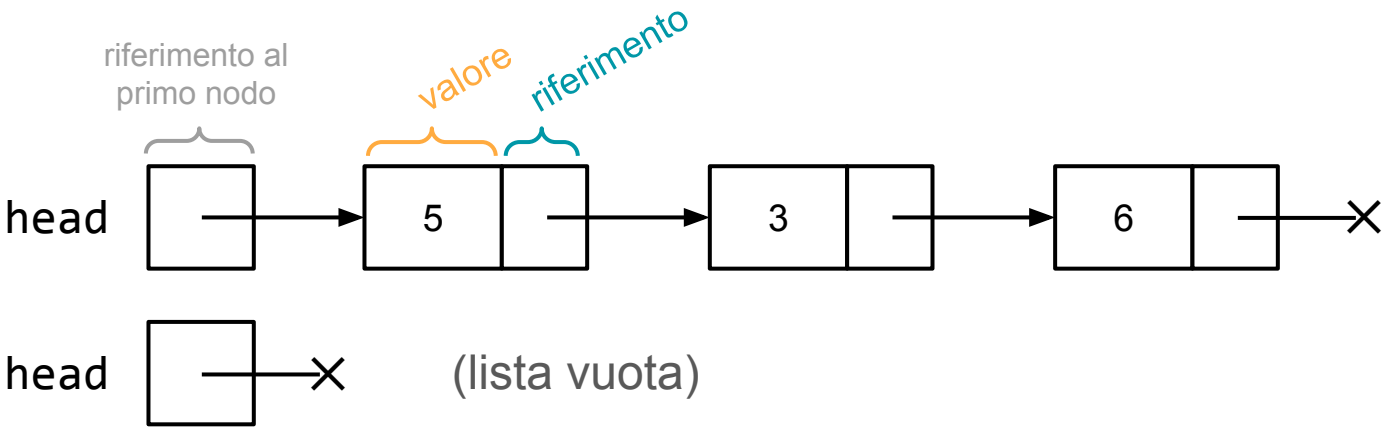
- **vuota**, oppure
- costituita da **un nodo** che contiene
  - un valore (di qualunque tipo)
  - un riferimento a una lista concatenata



Liste concatenate

Struttura dati ricorsiva che può essere:

- **vuota**, oppure
- costituita da **un nodo** che contiene
  - un valore (di qualunque tipo)
  - un riferimento a una lista concatenata



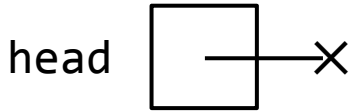
Perché studiare le liste concatenate

- Le liste rappresentano una sequenza di valori, proprio come gli array
  - Negli **array** l'ordine è dato dagli **indici**
  - Nelle **liste** l'ordine è dato dai **riferimenti** tra nodi
- Con gli array di JavaScript non vediamo cosa succede "dietro le quinte"
  - `push`, `pop`, `shift`, `unshift`, … spostano elementi e modificano la dimensione dell'array
- Implementare le liste concatenate:
  - Allena all'uso dei riferimenti e della ricorsione
  - Prepara a strutture dati più complesse come alberi e grafi

Liste in JavaScript

- Ogni nodo è rappresentato da un **oggetto** con due proprietà:
  - `val`: un valore (di qualunque tipo)
  - `next`: un riferimento al nodo successivo
- Manteniamo un riferimento al primo nodo (es. in una variabile `head`)

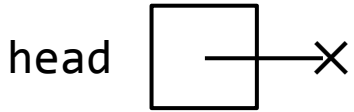
```
let head = null
```



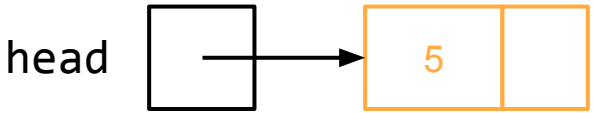
Liste in JavaScript

- Ogni nodo è rappresentato da un **oggetto** con due proprietà:
  - `val`: un valore (di qualunque tipo)
  - `next`: un riferimento al nodo successivo
- Manteniamo un riferimento al primo nodo (es. in una variabile `head`)

```
let head = null
```



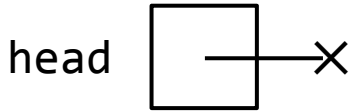

```
let head = { val: 5, next:  ...                    }
```



Liste in JavaScript

- Ogni nodo è rappresentato da un **oggetto** con due proprietà:
  - `val`: un valore (di qualunque tipo)
  - `next`: un riferimento al nodo successivo
- Manteniamo un riferimento al primo nodo (es. in una variabile `head`)

```
let head = null
```

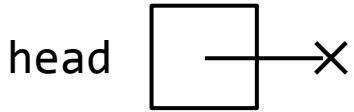



```
let head = { val: 5, next: { val: 3, next:  ...                    } }
```

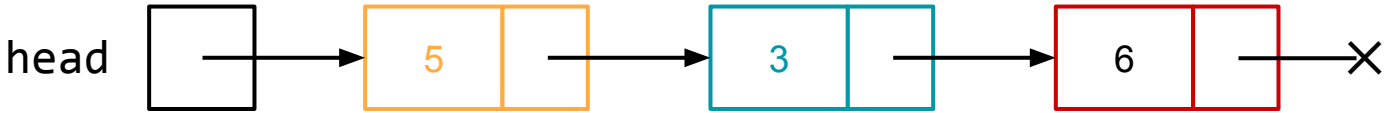
Liste in JavaScript

- Ogni nodo è rappresentato da un **oggetto** con due proprietà:
  - `val`: un valore (di qualunque tipo)
  - `next`: un riferimento al nodo successivo
- Manteniamo un riferimento al primo nodo (es. in una variabile `head`)

```
let head = null
```

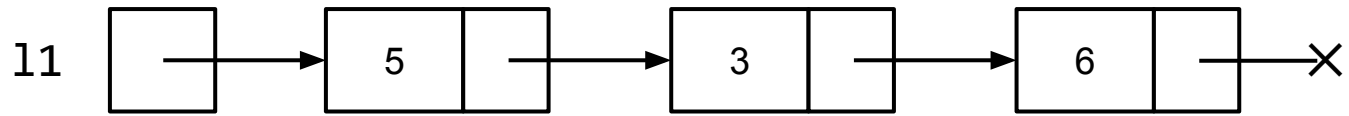



```
let head = { val: 5, next: { val: 3, next: { val: 6, next: null } } }
```



Liste e riferimenti

```
let l1 = { val: 5, next: { val: 3, next: { val: 6, next: null } } }
```

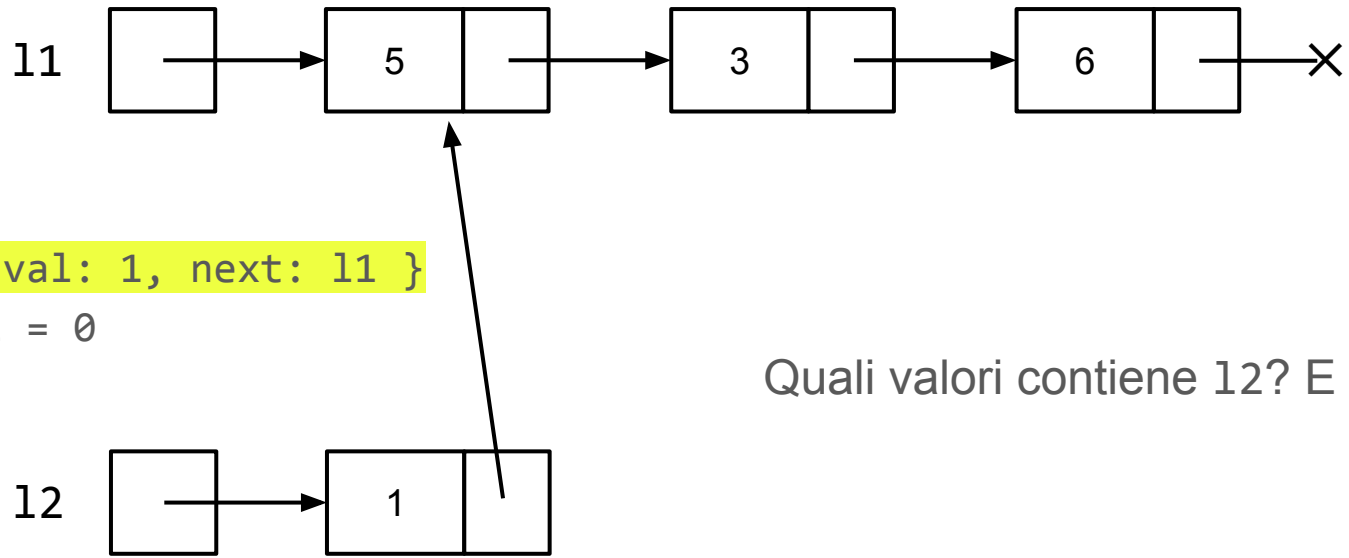



```
let l2 = { val: 1, next: l1 }
l2.next.val = 0
```

Quali valori contiene `l2`? E `l1`?

Liste e riferimenti

```
let l1 = { val: 5, next: { val: 3, next: { val: 6, next: null } } }
```

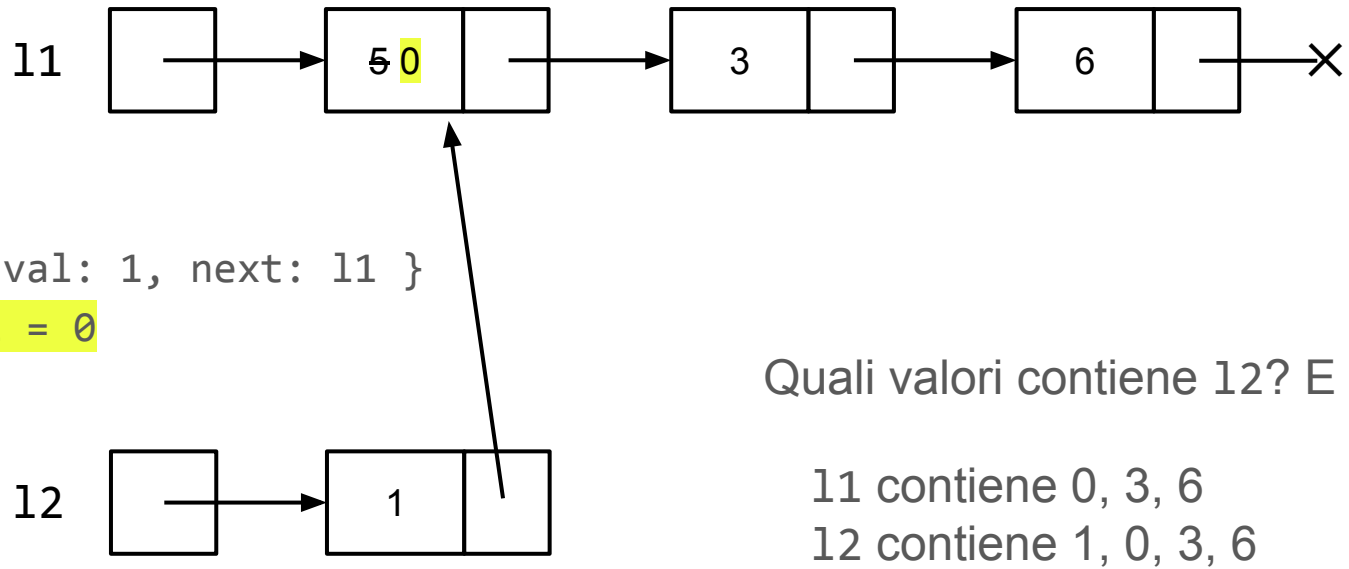



```
let l2 = { val: 1, next: l1 }
l2.next.val = 0
```

Quali valori contiene `l2`? E `l1`?

Liste e riferimenti

```
let l1 = { val: 5, next: { val: 3, next: { val: 6, next: null } } }
```



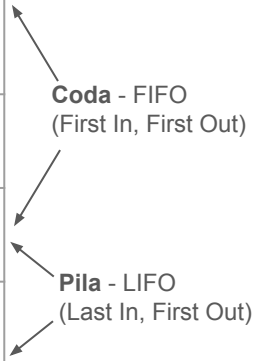

```
let l2 = { val: 1, next: l1 }
l2.next.val = 0
```

Quali valori contiene l2? E l1?

l1 contiene 0, 3, 6
l2 contiene 1, 0, 3, 6

Operazioni di base su liste

`listPrint(head)`	Stampa tutti i valori della lista in ordine
`listFind(head, value)`	Restituisce il primo nodo che contiene `value`, oppure `null` se non trovato
`listInsert(x, value)`	Inserisce un nuovo nodo con valore `value` dopo il nodo `x`
`listShift(head)`	Rimuove il nodo in testa e restituisce una coppia [testa aggiornata, valore rimosso]
`listUnshift(head)`	Inserisce un nuovo nodo in testa e restituisce la testa aggiornata
`listPush(head, value)`	Aggiunge un nodo in coda e restituisce la testa aggiornata
`listPop(head)`	Rimuove il nodo in coda e restituisce una coppia [testa aggiornata, valore rimosso]



Esercizi

Scrivere le seguenti funzioni ricorsive

1. `listLength(head)`: restituisce il numero di nodi nella lista
2. `listCopy(head)`: crea e restituisce una *nuova* lista identica a `head`
3. `listConcat(a, b)`: collega la lista `b` in fondo ad `a`, modificando i riferimenti di `a` e non creando nessun nuovo nodo
4. `listToArray(head)`: restituisce un array contenente tutti i valori della lista, nell'ordine in cui compaiono

Q & A