Laboratorio I
a.a. 2025/2026

Array avanzati; destrutturazione; default; spread

Contenuti

- Usi avanzati degli array
  - Array come liste
  - Array come code
  - Array come pile
  - Array come alberi
- Assegnamento con destrutturazione
- Valori di default
- Operatore di *spread*

Usi avanzati degli array

Abbiamo visto che gli array in Javascript sono molto più flessibili rispetto al semplice accesso con indice.

- Array disomogenei [1, 2, “pippo”, 3.1415, {x: 10, y:12}, 3]
- Array sparsi [1, 2, *<empty>×4*, 7]
- Array con chiavi non-numeriche
  var a=[1, 2]; a.pippo=3
  a → [1, 2, pippo: 3] a.length → 2
- Gestione implicita della lunghezza
  var a=[1, 2, 3]; a.length → 3
  a.length=2; a → [1, 2]
  a.length=4; a → [1, 2, *<empty>×2*]

Nota: gli array JS **non** sono un blocco di celle di memoria consecutive (come in C)!

Invece, sono dizionari con chiavi numeriche.

Array come **tuple**

Gli array possono essere usati come **tuple**, fissando la loro lunghezza.

Per esempio, la tupla ⟨4, 1⟩ può essere rappresentata dall'array [4, 1].

Si tratta di un uso *convenzionale*: il programmatore promette di usare sempre array di lunghezza 2, e sempre con gli elementi "giusti".

Se ho una tupla t = [4, 1], posso fare:

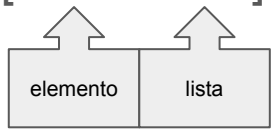
- Accedere agli elementi: t[0] → 4 t[1] → 1
- Assegnare l'intera tupla: `q = t;` q → [4, 1]
- Assegnare separatamente gli elementi: `[a, b] = t;` a → 4 b → 1
- Chiamare una funzione passando la tupla: `f(t)`
- Restituire più risultati da una funzione: `return t;`

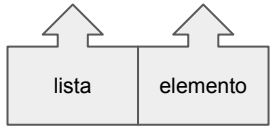
Array come **liste**

Gli array possono essere usati anche come **liste**, ovvero come **sequenze** di elementi di lunghezza variabile.

In questo stile, raramente si accede agli elementi tramite la loro posizione (*per indice*). Invece, si lavora in maniera **strutturale**:

- Una **lista vuota** è una lista: [ ]
- Un **elemento seguito da una lista** è una lista: [*testa resto*]
- In maniera analoga, ci si può concentrare sull'ultimo elemento: [*resto coda*]





Array come **liste**

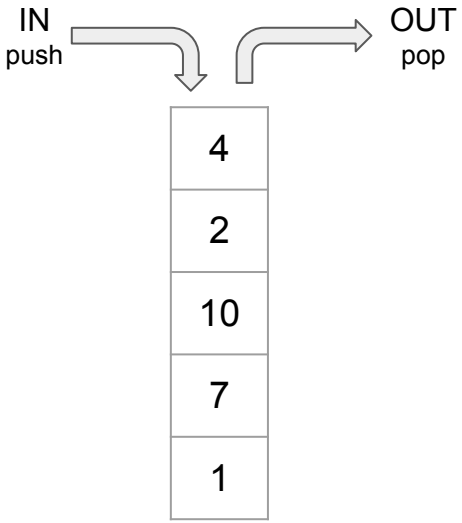
Molte funzioni predefinite sugli array li trattano come liste, e consentono di realizzare facilmente strutture dati come **code** e **pile** (che vedrete anche a P&A!)

Se A=[1, 2, 3, 4], allora:

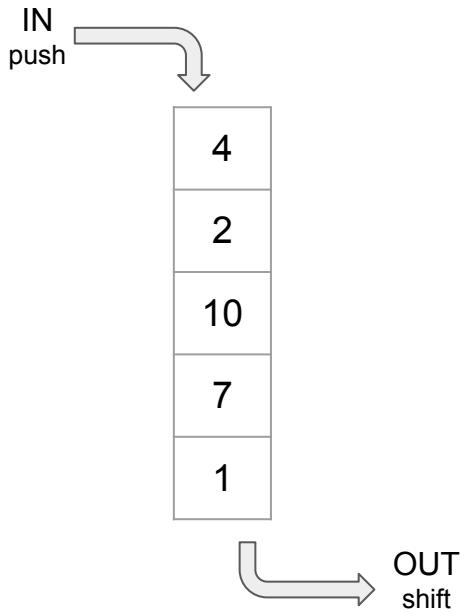
A.**push**(10) → 5	A → [1, 2, 3, 4, 10]	Aggiunge un elemento in coda a un array, restituisce la nuova lunghezza
A.**pop**() → 4	A → [1, 2, 3]	Estrae la coda dall'array e la restituisce
A.**shift**() → 1	A → [2, 3, 4]	Estrae la testa dall'array e la restituisce
A.**unshift**(10) → 5	A → [10, 1, 2, 3, 4]	Aggiunge un elemento in testa all'array, restituisce la nuova lunghezza

Pile e code

Pila - LIFO (Last In, First Out)



Coda - FIFO (First In, First Out)



Pile e code

Per realizzare una **pila**:

- A.push() per inserire sulla pila, A.pop() per estrarre dalla pila (cresce in coda)
- A.unshift() per inserire sulla pila, A.shift() per estrarre dalla pila (cresce in testa)

Per realizzare una **coda**:

- A.push() per aggiungere in coda, A.shift() per estrarre dalla testa
- A.unshift() per aggiungere in testa, A.pop() per estrarre dalla coda

Per convenzione, si tende a usare **push/pop per le pile**, **push/shift per le code**

Array come liste

Si può fare un *assegnamento destrutturante* con *spread* anche sulle liste:

- [testa, …resto] = [1, 2, 3, 4, 5] testa → 1 resto → [2, 3, 4, 5]
- [testa, …resto] = [1] testa → 1 resto → [ ]
- [testa, …resto] = [ ] testa → undefined resto → [ ]

Ciò consente di scrivere facilmente algoritmi ricorsivi sulle liste (qui, di numeri naturali):

```
function len(a) {
  let [t, ...r] = a
  return ????
}
```

Beh, c'è anche `a.length` :-)

```
function sum(a) {
  let [t, ...r] = a
  return ????
}
```

```
function max(a) {
  let [t, ...r] = a
  if (t)
    return ????
  else
    return 0
}
```

Array come liste

Si può fare un *assegnamento destrutturante* con *spread* anche sulle liste:

- [testa, …resto] = [1, 2, 3, 4, 5] testa → 1 resto → [2, 3, 4, 5]
- [testa, …resto] = [1] testa → 1 resto → [ ]
- [testa, …resto] = [ ] testa → undefined resto → [ ]

Ciò consente di scrivere facilmente algoritmi ricorsivi sulle liste (qui, di numeri naturali):

```
function len(a) {
  let [t, ...r] = a
  if (t==undefined)
    return 0
  return 1+len(r)
}
```

Beh, c'è anche `a.length` :-)

```
function sum(a) {
  let [t, ...r] = a
  if (t==undefined)
    return 0
  return t+sum(r)
}
```

```
function max(a) {
  let [t, ...r] = a
  if (t)
    return Math.max(t,max(r))
  else
    return 0
}
```

Array come liste

Alcuni metodi predefiniti consentono di scorrere un array (come lista) in maniera simile a quanto abbiamo visto negli esempi, ma più generale.

Se A è un array di elementi di tipo E, p:E→Bool un predicato, f:E→X una funzione

A.**every**(p)	Restituisce true se **tutti** gli elementi di A soddisfano il predicato p, false altrimenti
A.**some**(p)	Restituisce true se **almeno uno** degli elementi di A soddisfa il predicato p, false altrimenti
A.**find**(p)	Restituisce un elemento e di A tale che p(e) è true, o undefined se nessun e soddisfa p
A.**findIndex**(p)	Restituisce l'indice di un elemento e di A tale che p(e) è true, o -1 se nessun e soddisfa p
A.**includes**(e)	Restituisce true se A contiene un elemento uguale a e, false altrimenti
A.**forEach**(f)	Invoca f(e) per ogni elemento e dell'array A
A.**map**(f)	Restituisce un nuovo array A' = [$f(e_1)$, $f(e_2)$, … $f(e_n)$]
A.**filter**(p)	Restituisce un nuovo array A' contenente i soli elementi e di A che soddisfano p, in ordine

Array come liste

Sono a volte utili le funzioni di **riduzione**.

L'idea è semplice: si parte da un valore dato che costituisce il risultato iniziale, poi si scorre la lista, un elemento alla volta, e si applica una funzione data al risultato precedente e al nuovo elemento. Alla fine della lista, si restituisce il risultato.

A.**reduce**(f,z)	Fa una riduzione da sinistra a destra, partendo da z, e applicando f
A.**reduceRight**(f,z)	Fa una riduzione da destra a sinistra, partendo da z, e applicando f

Esempio

`A.reduce((r,e)=>r+e, 0)` — restituisce la somma di tutti gli elementi di A

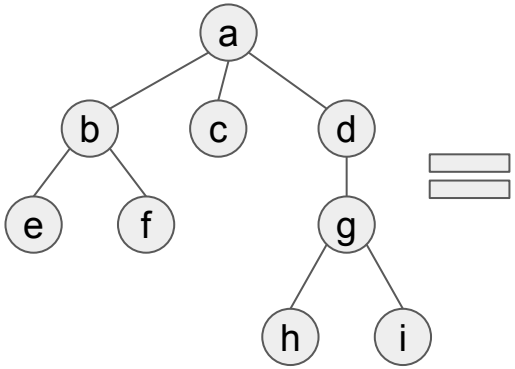
$f(e_n, \ldots f(e_1, f(e_0, z))\ldots)$ — $e_n + \ldots + e_1 + e_0 + 0$

Array come alberi

Notate come *tuple* e *oggetti* sono molto simili: però nelle tuple accedo ai membri **per posizione**, mentre negli oggetti accedo **per nome**.

Combinando le convenzioni per tuple e liste, possiamo rappresentare **alberi** (binari o k-ari) con gli array.

- Alberi binari: nodo = tripla [*valore*, *sx*, *dx*] (anche: { val: *v*, sx: *s*, dx: *d* })
- Alberi k-ari: nodo = coppia [*valore*, [*figli*]] (anche: { val: *v*, figli: *f* })




```
[ a, [  [ b, [ e, f ] ],
        [ c ],
        [ d, [ g, [ h, i ] ] ]
] ]
```

2-upla: [valore, [f_1, f_2, … f_n]]

Assegnamenti destrutturanti

Il concetto di *destrutturazione* è semplice: si prende un **tipo strutturato** (array, oggetto) e lo si “spacchetta” andando a guardare al suo interno.

In molti casi in cui normalmente è previsto un **nome di variabile** (tecnicamente: una *left-hand side* o lhs, ciò che può stare a sinistra di un =), JavaScript consente di usare la notazione di un **letterale** array o oggetto, in cui però al posto dei valori si indicano dei **nomi di variabili** (o meglio: dei lhs).

Notate che anche questa definizione è ricorsiva: si possono destrutturare array o oggetti che contengono altri array o oggetti come elementi, e così via ...

Assegnamenti destrutturanti

Alcuni esempi sugli array

Se A = [4, 7, 1]:

- [a, b] = A a → 4 b → 7
- [a, b, c, d] = A a → 4 b → 7 c → 1 d → undefined
- [a, b, c=3, d=8] = A a → 4 b → 7 c → 1 d → 8
- [, , c] = A c → 1
- [a, ...r] = A a → 4 r → [7, 1]
- [y, x] = [x, y] scambia i valori di x e y (senza variabili intermedie)
- [x, y] = f(“pippo”,3) ottiene due valori di ritorno distinti da una funzione

Assegnamenti destrutturanti

Alcuni esempi sugli oggetti

Se O = { n: “Pippo”, a: 35, c: true }:

- { a, c } = O a → 35 c → true
- { a, b } = O a → 35 b → undefined
- { a, b=2 } = O a → 35 b → 2
- { a, ...r } = O a → 35 r → { n: “Pippo”, c: true }
- { a: età, n: nome} = O età → 35 nome → “Pippo”
- { a: età, b: bimbi=0 } = O età → 35 bimbi → 0
- { x, y } = f(“pluto”, 3) estrae 2 fra molti valori di ritorno di una funzione, indicando per nome (x e y) quelli interessanti

Nota: in alcuni casi, il simbolo { potrebbe essere interpretato come l’inizio di un blocco. Per evitare confusione, si può scrivere l’assegnamento fra parentesi tonde.

✔ let { a, c } = O

✘ { a, c } = O

✔ ({ a, c } = O)

Ambiguità: *comando* ::= *assegnamento* | ‘{‘ *blocco* ‘}’

Assegnamenti destrutturanti

La destrutturazione sugli oggetti consente anche di avere funzioni con molti parametri, non tutti obbligatori; il chiamante indica **per nome** anziché **per posizione** quelli che vuole indicare.

```
function disegna( x, y, { raggio=0, colore="nero", bordo=1, etichetta="" } )
{
    /* codice di disegno; usa x,y, raggio, colore, bordo, etichetta  */
}
```

E il chiamante può di volta in volta decidere quali caratteristiche vuole specificare

```
disegna(5,8, {colore="blu"})
disegna(5,8,{colore="rosso", bordo=2})
```

```
disegna(3, 2)
disegna(0,1,stile)
```

Valori di default

Nella definizione di una funzione possiamo dichiarare dei parametri opzionali, con valori di default:

```
function manipola(arr,filtro,aggiorna,n=2){
  let ris=arr

  for(let j=0;j<n;j++){
    ris=ris.map(aggiorna).filter(filtro)
  }

  return ris
}
```

```
let filtro = (x) => x % 2 != 0 || x > 4;
let aggiorna = (x) => x + 1;

manipola([1, 2, 3, 4], filtro, aggiorna)

manipola([1, 2, 3, 4], filtro, aggiorna,3)
```

Valori di default

Quindi i valori di default si possono usare:

- Nella dichiarazione di funzione, per dare un valore di default ai parametri formali
- Negli assegnamenti destrutturanti, per dare un valore di default agli elementi assenti (per numero negli array, per nome negli oggetti)

In generale: quando normalmente a una variabile sarebbe assegnato *undefined*, se nella dichiarazione c'è un valore di default, si assegna quel valore anziché *undefined*.

Lo spread

L’operatore di spread è denotato da tre punti consecutivi: **...**

Ha il significato generale di trasformare una *sequenza di elementi singoli* in un *singolo dato strutturato* e viceversa.

Alcuni esempi:

- f(...A) — chiama la funzione f, passando gli elementi di A come argomenti singoli
- p = { ...q, c: 3 } — p è una copia dell’oggetto q, con la chiave c che vale 3
- p = { c: 3, ...q } — p è una copia di q, con aggiunto c=3 se manca in q, altrimenti il valore di c in q
- B = [1, 2, ...A, 6] — B è un array contenente gli elementi di A, ma con 1 e 2 davanti e 6 in coda
- C = [...A, ...B] — C è la concatenazione degli array A e B
- R = { ...O1, ...O2 } — R è l’unione dei due oggetti O1 e O2; se hanno delle chiavi in comune, “vince” il valore di O2
- [testa, ...resto] = A — testa prende il primo elemento di A, resto tutti gli altri

Funzioni con numero variabile di parametri

Può essere necessario avere un numero variabile di parametri:

- Il parametro ‘rest’
  - `function f(p1,p2, ...args)  {}`
    - `args` è un array di parametri inclusi dopo i primi 2
- L’oggetto `arguments`
  - contiene un array con tutti i valori dei parametri, disponibile nello scope della funzione

Esercizi

Operazioni

Scrivere una funzione JavaScript `math` che prende come primo parametro un operatore (stringa) e poi un numero indefinito di operandi (numeri), e applica l'operazione agli operandi, in sequenza. L'operatore può essere uno tra: `'+', '-', '*', '/'`

Esempi:

```
math('+',4,2,7) -> 13

math('/', 8,2,2,2) -> 1
```

Filter in-place

Scrivere una funzione `fip(a,p)` che, dato un array qualunque `a`, e un predicato `p`, modifichi `a` in modo che tutti gli elementi che non soddisfano il predicato `p` siano rimossi da `a`. Si curi di non lasciare “posti vuoti” in `a`.

Esempi

Se `a=[3,5,10,1,4]`, dopo la chiamata `fip(a,x=>x%2)` a dovrà essere `[3,5,1]`

Sommatoria

Si scriva una funzione `somma(n1, …, nk)` che, ricevuti come argomento un numero qualunque di numeri, restituisca la loro somma.

Esempi

`somma(5,7,2)` → `14`

`somma(3)` → `3`

`somma(10,5,-8,-7)` → `0`

isSorted

Scrivere una funzione `isSorted(a)` con `a` un array di numeri. La funzione restituisce `true` se l'array è ordinato in senso strettamente crescente, `false` altrimenti. Risolverlo senza usare cicli (comandi for, while, do/while). Provare a risolverlo con `.reduce()`.

Esempi:

```
isSorted([-21,-2,0,4,6,210]) -> true

isSorted([2,6,8,8,9,21]) -> false

isSorted([2,6,8,9,10,-42]) -> false
```

Deframmenta

Scrivere una funzione `deframmenta(a)`, con `a` array di numeri. La funzione restituisce una copia di `a` da cui sono state eliminate le occorrenze dei numeri non ripetute in sequenza (ovvero in posizioni contigue dell'array). Ad esempio dato l'array `a = [1,1,2,3,3,3,2,2,4]` la chiamata `deframmenta(a)` restituisce `[1,1,3,3,3,2,2]`, dove gli elementi in posizione `2` e `4` sono stati eliminati in quanto non ripetuti in sequenza.

Esempi

```
deframmenta([1,1,2,3,3,3,2,2,4]) → [1,1,3,3,3,2,2]
deframmenta([0,0,0,0,0,1,0,1,1]) → [0,0,0,0,0,1,1]
deframmenta([1,0]) → []
```

Una fabbrica di funzioni costanti

Si scriva una funzione `fabbrica()` che abbia il seguente comportamento.

Ogni volta che viene invocata, `fabbrica(k)` restituisce una funzione `f` tale che la chiamata `f()` restituisce (sempre) il valore `k`.

Esempio

```
var f=fabbrica(1)
var g=fabbrica(2)
var h=fabbrica(true)
f() → 1
g() → 2
h() → true
```

Applicazione parziale

Si scriva una funzione `partapply(bop,a)` che, data una funzione `bop` con due argomenti, e un valore `a`, restituisca una funzione che, se invocata con un argomento `b`, ritorni il valore di `bop(a,b)`. In altri termini:

```
∀ bop,a,b . partapply(bop,a)(b) ≡ bop(a,b)
```

Esempi

Se `r=partapply((x,y)=>x+y,1)` allora `r(5)` restituisce `6`.

Se `r=partapply((x,y)=>x+y,"d")` allora `r("esisto")` restituisce `"desisto"`.

Reverse

Si scriva una funzione `reverse(a)` che, dato un array `a`, restituisca un nuovo array contenente gli stessi elementi di `a`, ma in ordine inverso. La funzione reverse non deve modificare `a`.

Si provino diverse tecniche di soluzione:

- iterativa
  - usando dei cicli e opportuni indici
  - usando i metodi degli array
- ricorsiva
  - usando i metodi degli array
  - usando assegnamenti destrutturanti e l’operatore di spread

Funprop

Si scriva una funzione `funprop(f,p)` che, date due funzioni $f:\mathbb{N}\to\mathbb{R}$ e $p:\mathbb{R}\to\mathbb{B}$ (dove $\mathbb{B}$ rappresenta il dominio dei booleani), restituisca una funzione `s(a,b)` la quale, ricevuti due interi `a` e `b` con $a\leq b$, restituisca un array ordinato di tutti gli interi $k\in[a,b]$ in cui `f` soddisfa `p`. Se `p` non viene fornito, si considera sempre soddisfatto.

Esempi

```
funprop(n=>2*n,n=>n%2==0)(4,6) → [4, 5, 6]
funprop(n=>2*n,n=>n>10)(4,8) → [6,7,8]
funprop(n=>n,n=>n%2==1)(10,20) → [11,13,15,17,19]
funprop(n=>n*n)(4,6) → [4,5,6]
```

Q & A