

Laboratorio I
a.a. 2026/2027

Esercitazione: array, funzioni, oggetti e refactoring

Contenuti

- Code refactoring
  - Esempi ed esercizi
- Progettazione modulare
- Esercitazione sugli oggetti

Code refactoring

Riscrivere il codice senza cambiarne il comportamento, per renderlo migliore.

- **Leggibilità**: nomi che dicono cosa fa il codice
  - funzioni troppo piccole o dai nomi vaghi possono però peggiorare la lettura
- **Riuso**: scritto una volta, chiamato ovunque
  - verifica diretta dell'output
- **Modularità**: pezzi piccoli, ciascuno con un solo compito
- **Manutenzione**: un bug o una modifica si correggono in un punto solo

Regola pratica: *ogni volta che ti accorgi di copiare e incollare lo stesso pezzo di codice, quel pezzo starebbe meglio in una funzione da chiamare dove serve.*

Code refactoring (2)

In generale, per rendere modulare un codice esistente:

1. individua i blocchi ripetuti o con un nome naturale
2. estrai una funzione (parametri = ciò che cambia, return = ciò che serve fuori)
3. rilancia con gli stessi input e controlla che l'output sia identico
4. utilizza nomi chiari

Esempio: media dei voti *con funzioni*

(**Media dei voti**) Si scriva un programma che, dato un insieme di studenti con i rispettivi voti di 3 esami, calcoli e visualizzi la media aritmetica dei voti di ciascuno. L'utente fornirà il nome di ogni studente, seguito dai suoi 3 voti. L'acquisizione termina quando l'utente inserisce "stop" come nome di uno studente.

Esempio

Input: Luca, 28, 30, 27, Marta, 30, 29, 30, Marco, 18, 22, 25, stop

Output: Luca: 28.33

Marta: 29.67

Marco: 21.67

Media dei voti: cosa si può modularizzare?

```
type Studente = {nome:string, voti:number[]};
let studenti:Studente[] = [];
let nome:string = String(prompt("Nome (o 'stop'):"));
const N_ESAMI: number = 3

while (nome != "stop") {
  let esami:number[] = [];
  for (let i = 0; i < N_ESAMI; i++) {
    let voto:number = Number(prompt(`Voto ${nome}:`));
    esami.push(voto);
  }
  studenti.push({nome: nome, voti: esami});
  nome = String(prompt("Nome (o 'stop'):"));
}

for (let i = 0; i < studenti.length; i++) {
  let somma:number = 0;
  for (let j = 0; j < studenti[i].voti.length; j++) {
    somma += studenti[i].voti[j];
  }
  let media:number = somma/(studenti[i].voti.length);
  console.log(studenti[i].nome + ": " + media);
}
```

Media dei voti: cosa si può modularizzare?

possibli funzioni!

```
type Studente = {nome:string, voti:number[]};
let studenti:Studente[] = [];
let nome:string = String(prompt("Nome (o 'stop'):"));
const N_ESAMI: number = 3

while (nome != "stop") {
  let esami:number[] = [];
  for (let i = 0; i < N_ESAMI; i++) {
    let voto:number = Number(prompt(`Voto ${nome}:`));
    esami.push(voto);
  }
  studenti.push({nome: nome, voti: esami});
  nome = String(prompt("Nome (o 'stop'):"));
}

for (let i = 0; i < studenti.length; i++) {
  let somma:number = 0;
  for (let j = 0; j < studenti[i].voti.length; j++) {
    somma += studenti[i].voti[j];
  }
  let media:number = somma/(studenti[i].voti.length);
  console.log(studenti[i].nome + ": " + media);
}
```

Refactoring: organizzare il codice con le funzioni

Soluzione attuale

```
type Studente = {nome:string, voti:number[]};
let studenti:Studente[] = [];
let nome:string = String(prompt("Nome (o 'stop'):"));
const N_ESAMI: number = 3

while (nome != "stop") {
  let esami:number[] = [];
  for (let i = 0; i < N_ESAMI; i++) {
    let voto:number = Number(prompt(`Voto ${nome}:`));
    esami.push(voto);
  }
  studenti.push({nome: nome, voti: esami});
  nome = String(prompt("Nome (o 'stop'):"));
}

for (let i = 0; i < studenti.length; i++) {
  let somma:number = 0;
  for (let j = 0; j < studenti[i].voti.length; j++) {
    somma += studenti[i].voti[j];
  }
  let media:number = somma/(studenti[i].voti.length);
  console.log(studenti[i].nome + ": " + media);
}
```

Con le funzioni

```
type Studente = {nome:string, voti:number[]};
const N_ESAMI: number = 3
function leggiStudente(nome:string): Studente {
  let esami:number[] = [];
  for (let i = 0; i < N_ESAMI; i++) {
    esami.push(Number(prompt(`Voto ${nome}:`)));
  }
  return {nome: nome, voti: esami};
}
function media(voti:number[]): number {
  let somma:number = 0;
  for (let v of voti) somma += v;
  return somma / voti.length;
}

let studenti:Studente[] = [];
let nome:string = String(prompt("Nome (o 'stop'):"));
while (nome != "stop") {
  studenti.push(leggiStudente(nome));
  nome = String(prompt("Nome (o 'stop'):"));
}
for (let s of studenti)
  console.log(s.nome + ": " + media(s.voti));
```

Modularizzare il codice con le funzioni: note

- **Si riduce lo sforzo di leggere:** si vede subito “cosa fa” il programma (ad esempio `leggiStudente(nome)`, `media(voti)`) e si scende nei dettagli solo quando serve.
- **Si corregge in un punto solo:** se ripeti il calcolo della media in tre posti e scopri un errore, devi ricordarti di correggerlo in tutti e tre. Con `media()` lo correggi una volta e vale ovunque.
- **Si possono provare i pezzi separatamente:** puoi chiamare media([28, 30, 27]) e vedere se ritorna 28.33; separare il calcolo dall'I/O è meglio di inserire tutto ogni volta.
- **Si lavora con meno cose in testa**: dentro `media()` ti interessano solo `voti` e `somma`, non `studenti`, `nome` o `esami`. Le variabili locali restano confinate, e ci sono meno occasioni di sbagliare nome o riutilizzare per errore variabili di un altro blocco.

- **Non si guadagna in velocità di esecuzione**: il costo di chiamare una funzione è trascurabile, e gli interpreti moderni (anche i motori JS) eliminano il ritardo da soli.

Progettazione modulare: dalla traccia ai pezzi

Esercizio: *…dato un insieme di **studenti** con i rispettivi **voti** di 3 esami, **calcoli** e **visualizzi** la **media** aritmetica dei voti di ciascuno. L'utente **fornirà** il **nome** di ogni studente, seguito dai suoi 3 voti. L'**acquisizione termina** quando l'utente inserisce "stop".*

1. **Leggi la traccia** e sottolinea i *sostantivi* (→ dati) e i *verbi* (→ azioni).
2. **Rappresenta i dati**: un tipo per ogni "cosa" della traccia, un array per gli insiemi, una costante per i numeri fissi.
3. **Dividi in funzioni**: una per azione, ciascuna con un solo compito. Scrivi solo la **firma** (cosa entra → cosa esce).
4. **Scrivi il programma principale** come sequenza di chiamate; poi implementa e **prova** una funzione alla volta.

Progettazione modulare: dallo scheletro al programma

Nella traccia	Nel codice
uno studente = nome + 3 voti	`type Studente = {nome: string, voti: number[]}`
un insieme di studenti	`Studente[]`
“3 esami”	`const N_ESAMI: number = 3`
acquisire nome / uno studente	`leggiNome(): string`, `leggiStudente(nome: string): Studente`
media aritmetica	`media(voti: number[]): number`
visualizzare	ciclo `for..of` + `console.log`
“finché stop”	`while` nel programma principale

- E se la traccia cambia? (es. si vuole anche lo studente con la media più alta)
  - si aggiunge una funzione, non si riscrive il resto.

Esercitazione sugli oggetti

Esercizio 1: Array + funzioni

(**Perfetto**) Dato x ≥ 1, trovare il più piccolo numero perfetto p ≥ x e stamparlo con la lista dei suoi divisori (escluso p) separati da +.

Usa le funzioni

- `divisori(n: number): number[]`
- `somma(arr: number[]): number`
- `formattaDivisori(p: number): string`
- `primoPerfetto(x: number): number`

Esempio.

- **Input**:
  - 30
- **Output**:
  - 496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248

Esercizio 2: Oggetti + funzioni

(**Cassa automatica**) Letto il prezzo (in euro), leggere monete (in euro) finché il totale lo copre; poi stampare il resto con il minor numero di monete. Tagli possibili delle monete in euro: 2, 1, 0.50, 0.20, 0.10, 0.05, 0.01.

Usa le funzioni

- **`leggiPagamento(pCent: number): number`**
- **`calcolaResto(cent: number, tagli: number[]): Resto`**
- **`stampaResto(r: Resto): void`**

Hint:

- `type Resto = {[taglio: string]: number}`
- Usa chiavi come "0.50" perché l'ordine delle chiavi intere non segue l'inserimento

Esempio.

- **Input**:
  - Prezzo: 4.10€
  - Monete: 2€, 2€, 1 €
- **Output**:
  - 1 x 0.50€
  - 2 x 0.20€

Esercizio 3: Tipi + oggetti

(**Inventario**) Leggere (una per volta) coppie prodotto-quantità fino a "stop", sommando le quantità dei prodotti ripetuti; stampare l'inventario su una riga.

Usa le funzioni

- **`aggiungi(inv: Inventario, p: string, q: number): void`**
- **`formatta(inv: Inventario): string`**

Hint:

- `type Inventario = {[p: string]: number}`

Esempio.

- **Input**:
  - latte 2, uova 6, pane 7, latte 3, uova 6, stop
- **Output**:
  - (latte, 5) (uova, 12) (pane, 7)

Esercizio 4: Tipi + oggetti annidati

(**Analisi delle vendite**) Dato il negozio, restituire una tupla [incasso totale, prodotto più venduto]. A parità di pezzi, vince il nome alfabeticamente minore.

Usa le funzioni

- **`totalePezzi(p: Prodotto): number`**
- **`incasso(p: Prodotto): number`**
- **`analizzaVendite(negozio: Negozio): [number, string]`**

Hint:

- `type Prodotto = {prezzo: number, vendite: number[]}`
- `type Negozio  = {[nome: string]: Prodotto}`

Esempio.

- **Input**:
  - { quaderno: {prezzo: 3, vendite: [2, 4]},
  - penna: {prezzo: 2, vendite: [5, 3, 1]},
  - gomma: {prezzo: 1, vendite: [1, 2, 0]} }
- **Output**:
  - [39, "penna"]

Esercizio 5: Interface + array + oggetti

(**Registro studenti**) Data una lista di studenti, stampare la classifica per media decrescente (a parità: ordine alfabetico), con la media a 2 cifre decimali (default), e frequenze degli studenti per fascia di media in [0,18), [18,24), [24,28), [28,30].

Usa le funzioni

- `media(voti: number[], cifre: number): number`
- `precede(a: Studente, b: Studente): boolean`
- `classifica(s: Studente[]): string[]`
- `istogramma(s: Studente[]): Fasce`

Hint:

- `interface Studente { nome: string, voti: number[] }`
- `type Fasce = {[fascia: string]: number}`

Esempio.

- **Input**:
  - Anna [30,28,29], Bea [30,28,29], Zoe [30,29,30], Ugo [27,28,28], Eva [17,15,18]
- **Output**:
  - Classifica: Zoe: 29.67, Anna: 29, Bea: 29, Ugo: 27.67, Eva: 16.67
  - Fasce: [0,18): 1, [18,24): 0, [24,28): 1, [28,30]: 3