Esercitazione 6 - 02/11

Ultima esercitazione prima del compitino: **esercizi di riepilogo** su tutti gli argomenti visti finora.

Argomenti:

- tipi, operatori, casting, `++`, valori truthy e falsy
- condizioni: `if`, operatori logici
- cicli: `for`, `while`, `do ... while`, cicli annidati
- funzioni: parametri, `return`, variabili locali e globali
- funzioni ricorsive

Mappa degli argomenti

Argomento	Dove l'abbiamo visto	Esercizi di oggi
Tipi, operatori, casting, `++`, operatori logici	Esercitazioni 1, 2, 3	1
Tracciare un ciclo	Esercitazioni 3, 4	2
Condizioni e operatori logici	Esercitazione 2	3
`for`, `do ... while`, accumulatori	Esercitazioni 2, 3	4
`while` con sentinella	Esercitazione 2	5
Funzioni: errori tipici	Esercitazione 4	6
Variabili locali e globali, scope	Esercitazioni 4, 5	7
Scrivere funzioni	Esercitazioni 3, 4	8
Ricorsione	Esercitazione 5	9
Cicli annidati	Esercitazioni 3, 5	8

Indice degli esercizi

Esercizio	Argomento
Esercizio 1	Quiz a risposta multipla
Esercizio 2	Ciclo while: cosa stampa
Esercizio 3	Anno bisestile: scrivere il programma
Esercizio 4	Multipli di 3: completare il programma
Esercizio 5	Numeri fino a zero: positivi e multipli di 2
Esercizio 6	Funzioni: trovare gli errori
Esercizio 7	Variabili locali e globali: cosa stampa
Esercizio 8	Scrivere funzioni: massimo, tavola pitagorica, multipli
Esercizio 9	Ricorsione: scrivere una funzione

Esercizio 1

Dieci domande di ripasso. Rispondete **senza eseguire** il codice.

1. Quale di queste righe provoca un errore?

- a) `let x = 1; x = 2;`
- b) `const y = 1; y = 2;`
- c) `let z; z = 3;`
- d) `const w = 4; console.log(w);`

2. Quale di questi confronti vale false?

- a) `2 == "2"`
- b) `3 >= 3`
- c) `"a" < "b"`
- d) `2 === "2"`

3. "Chiedi un dato finché non è valido, con almeno una richiesta". Quale ciclo è il più adatto?

- a) for

- b) while
- c) do ... while
- d) Nessuno dei tre

4. Perché `if (x = 5)` è sempre un errore?

- a) Perché `=` assegna, non confronta
- b) Perché 5 non è un booleano
- c) Perché x dovrebbe essere const
- d) Perché manca l'else

5. Qual è la differenza tra parametro e argomento di una funzione?

- a) Il parametro è il valore restituito
- b) L'argomento sta nella definizione
- c) Parametro nella definizione, argomento nella chiamata
- d) Il parametro è sempre un numero

6. Dove è visibile una variabile dichiarata con let nel corpo di un for?

- a) In tutto il programma
- b) Solo dentro quel blocco
- c) Solo dopo la fine del ciclo
- d) Solo dentro le altre funzioni

7. Una funzione ricorsiva non termina mai. Cosa manca quasi certamente?

- a) Un parametro
- b) Il console.log
- c) Il return
- d) Il caso base

8. Quale condizione verifica che n sia pari e positivo?

- a) `n % 2 == 0 && n > 0`
- b) `n % 2 && n > 0`
- c) `n % 2 == 0 || n > 0`
- d) `n / 2 == 0 && n > 0`

9. while (i < 5) { console.log(i); } con i = 0, mai modificata nel corpo. Cosa succede?

- a) Stampa 0 1 2 3 4
- b) Stampa 0 una volta

- c) Errore di sintassi
- d) Stampa 0 all'infinito

10. Quale espressione dà il numero 42 a partire dalla stringa "42"?

- a) `Number("42")`
- b) `String(42)`
- c) `typeof "42"`
- d) `"42" + 0`

▼ **Soluzione**

1. b. `const` non permette di **riassegnare** la variabile: `y = 2` dà `TypeError`. Le altre righe sono valide.

2. d. `===` confronta anche il tipo: un numero e una stringa non sono mai strettamente uguali. `==` converte il tipo (`2 == "2"` è `true`), e le stringhe si confrontano in ordine alfabetico.

3. c. Il `do ... while` esegue il corpo **prima** di controllare la condizione: la richiesta avviene almeno una volta, poi si ripete finché il dato non è valido.

4. a. `x = 5` **assegna** `5` a `x` e vale `5` (truthy): la condizione è sempre vera. Per confrontare serve `===`.

5. c. In `function f(x) {...}` `x` è il **parametro**; in `f(4)` il valore `4` è l'**argomento**.

6. b. `let` ha visibilità di **blocco**: fuori dalle graffe del ciclo la variabile non esiste (`ReferenceError`).

7. d. Senza un **caso base** (o se le chiamate non si avvicinano ad esso) la funzione continua a richiamarsi finché lo stack si esaurisce.

8. a. Servono entrambe le cose, quindi `&&`. `n % 2` è `0` (falsy) proprio quando `n` è pari; `n / 2 == 0` vale solo per `0`.

9. d. `i` non cambia, quindi `i < 5` resta sempre vera: ciclo **infinito**.

10. a. `Number("42")` converte in numero. `"42" + 0` **concatena** e dà `"420"`; `typeof "42"` dà `"string"`.

Esercizio 2

Cosa stampa questo programma? Compilate una tabella con i valori di `a` e `b` dopo ogni giro per scoprirlo.

```
let a = 20;
let b = 1;
while (a > b) {
  a = a - b;
  b = b * 2;
  console.log(a, b);
}
console.log("fine", a, b);
```

A. Cosa stampa il programma?

B. Quante volte viene eseguito il corpo del ciclo?

▼ **Soluzione**

A. Tabella dei valori dopo ogni giro:

Giro	**`a`**	**`b`**	**`a > b` prima del giro**
(inizio)	20	1	
1	19	2	sì (20 > 1)
2	17	4	sì (19 > 2)
3	13	8	sì (17 > 4)
4	5	16	sì (13 > 8)

Dopo il giro 4, `a > b` è `5 > 16`: falso, il ciclo finisce. L'output è:

```
19 2
17 4
13 8
5 16
fine 5 16
```

B. 4 volte.

Esercizio 3

Un anno è **bisestile** se è divisibile per 4, **tranne** quelli divisibili per 100, a meno che non siano divisibili anche per 400. Ad esempio il 2024 e il 2000 sono bisestili, il 1900 e il 2023 no.

Scrivete un programma che legge un anno dall'input (controllando che sia un numero intero positivo) e stampa se l'anno inserito è bisestile o meno.

▼ **Soluzione**

```
let anno;
do {
  anno = Number(prompt("Inserire un anno: "));
} while (!Number.isInteger(anno) || anno <= 0);  // si ripete finché l'anno NON è valido

if ((anno % 4 == 0 && anno % 100 != 0) || anno % 400 == 0) {
  console.log(anno, "è bisestile");
} else {
  console.log(anno, "non è bisestile");
}
```

Controllate i casi: `2024`: bisestile (divisibile per 4, non per 100). `1900`: **non** bisestile (divisibile per 100, non per 400). `2000`: bisestile (divisibile per 400). `2023`: non bisestile (non divisibile per 4). Un anno come `-5` o `2.5` fa ripetere la domanda.

Come si legge la condizione del `do ... while`. Il `do ... while` si ripete finché la guardia è **vera**, quindi la guardia deve descrivere il caso **non** accettabile: l'anno va richiesto se **non è intero** oppure **non è positivo**. Se manca il pezzo `anno <= 0` il programma accetta `-2024` e lo dichiara bisestile, mentre l'esercizio chiedeva di rifiutarlo.

Ci sono due scritture equivalenti, e ciascuno sceglie quella che gli viene più naturale:

```
// 1. si scrivono direttamente i casi da rifiutare
while (!Number.isInteger(anno) || anno <= 0);

// 2. si nega la condizione che invece va bene (con le parentesi!)
while (!(Number.isInteger(anno) && anno > 0));
```

La seconda si ottiene dalla prima con le **leggi di De Morgan** (si nega ogni pezzo e si scambia `||` con `&&`). Attenzione al segno: l'anno va **richiesto di nuovo** quando è *minore* di zero, non quando è maggiore.

Un errore frequente è l'uso dell'operatore `%`: `anno % 4` **non** significa «anno per 4» né «anno diviso 4», ma il **resto** della divisione di `anno` per `4`. Ad esempio `27 % 4` vale `3`, perché `27 = 4 * 6 + 3`.

Esercizio 4

Il programma legge un intero positivo `n` e stampa la **somma dei multipli di 3** tra 1 e `n` (ad esempio per `n = 10`: `3 + 6 + 9 = 18`).

A. Completate gli spazi.

```
let n;
do {
  n = Number(prompt("Inserire un intero positivo: "));
} while (_____________________________);

let somma = ____;
for (let i = ____; i <= n; i = i + ____) {
  somma = somma + i;
}
console.log("Somma dei multipli di 3 fino a", n, ":", somma);
```

B. Cosa stampa il programma con `n = 10`? E con `n = 2`?

C. Cosa succede se l'utente inserisce `0`, poi `-4`, poi `2.5`, poi `9`?

▼ **Soluzione**

A.

```
let n;
do {
  n = Number(prompt("Inserire un intero positivo: "));
} while (!Number.isInteger(n) || n <= 0);       // si ripete finché n NON è valido

let somma = 0;
for (let i = 3; i <= n; i = i + 3) {            // si parte da 3 e si salta di 3 in 3
  somma = somma + i;
}
console.log("Somma dei multipli di 3 fino a", n, ":", somma);
```

B. Con `n = 10` il ciclo considera `i = 3, 6, 9` e stampa `Somma dei multipli di 3 fino a 10 : 18`. Con `n = 2` la condizione `3 <= 2` è subito falsa: il corpo non viene eseguito e la somma resta `0`.

C. `0`, `-4` e `2.5` non sono validi (`n <= 0` oppure non intero): il programma ripete la domanda. Con `9` esce dal `do ... while` e stampa `Somma dei multipli di 3 fino a 9 : 18` (`3 + 6 + 9`).

Perché `somma = 0` e perché il ciclo parte da 3. L'accumulatore deve partire da un valore che **non altera** il risultato: per una somma è `0`; con un valore diverso (per esempio `1`) la somma finale sarebbe quella dei multipli **più** quel valore.

Il `for` può essere completato in modi diversi, a seconda di come si legge il testo («multipli di 3 tra 1 e `n`»):

- **partire da `3` e saltare di `3` in `3`** (la soluzione sopra): ogni valore di `i` è già un multiplo di 3 e non serve nessun `if`;
- partire da `0` con passo `3`: funziona lo stesso, perché alla prima iterazione si somma `0`, che non cambia niente;
- partire da `1` con passo `1` (la lettura più "fedele" al testo): allora **non** si può più fare a meno di un `if (i % 3 == 0)` dentro il ciclo, che somma solo i multipli di 3.

La scelta dipende dallo **spazio** che l'esercizio lascia: se il codice proposto ha una sola riga nel corpo del `for` (`somma = somma + i;`) e nessun `if`, i valori di `i` devono essere **già** i multipli di 3, quindi il ciclo deve saltare di 3 in 3.

Esercizio 5

Scrivete un programma che legge numeri dall'utente **finché non viene inserito `0`**. Alla fine stampa:

- quanti numeri **positivi** e quanti **multipli di 2** sono stati inseriti.

Se il primo numero è `0`, il programma stampa `Nessun numero inserito`. Provatelo con `3, -2, 8, 0` e con `-5, -3, 0`.

▼ **Soluzione**

```
let n = Number(prompt("Inserire un numero (0 per terminare): "));

if (n == 0) {
  console.log("Nessun numero inserito");
} else {
  let positivi = 0;
  let pari = 0;
  while (n != 0) {
    if (n > 0) {
      positivi++;
    }
    if (n % 2 == 0) {                          // un multiplo di 2, anche se negativo
      pari++;
    }
    n = Number(prompt("Inserire un numero (0 per terminare): "));
  }
  console.log("Positivi:", positivi);
  console.log("Multipli di 2:", pari);
}
```

Con `3, −2, 8, 0`: positivi `2` (`3` e `8`), multipli di 2 `2` (`−2` e `8`). Con `−5, −3, 0`: positivi `0`, multipli di 2 `0`. Con `0` subito: `Nessun numero inserito`.

Le due condizioni sono in due `if` **separati**, non in un `if ... else`: un numero può essere positivo **e** pari (come `8`), e va contato in entrambi.

Perché ci sono due `prompt`. La lettura compare **due volte**: una **prima** del ciclo e una **alla fine del corpo**. Il `while (n != 0)` controlla l'ultimo numero letto: se è `0` non si entra nel ciclo (e il programma stampa `Nessun numero inserito`); se è diverso da `0` lo si elabora, e **poi** bisogna chiedere il numero successivo, che verrà controllato dal `while` al giro dopo. È lo schema classico della **sentinella** (il valore speciale, qui `0`, che chiude l'elenco): *leggi, finché non è la sentinella elabora e rileggi*.

Un'alternativa che evita l'`if` esterno è contare **quanti numeri** sono stati inseriti con un terzo contatore, e alla fine controllare se è maggiore di `0`. Con il `do ... while` lo stesso problema si può scrivere, ma bisogna fare attenzione al caso in cui il **primo** numero è già `0`: nel `do ... while` il corpo viene eseguito comunque una volta, e lo `0` verrebbe contato come numero valido (e come multiplo di 2).

Esercizio 6

Ciascuna di queste parti contiene un errore. Dite cosa stampa il codice e correggetelo.

A.

```
function area(base, altezza) {
  let a = base * altezza / 2;
}
let risultato = area(4, 5);
console.log("Area:", risultato);
```

B.

```
function doppio(x) {
  return x * 2;
  console.log("calcolo finito");
}
console.log(doppio(4));
```

C.

```
function media(a, b) {
  return (a + b) / 2;
}
console.log(media(4));
```

D. Il programma vuole incrementare `k` di 1.

```
function incrementa(n) {
  n = n + 1;
}
let k = 5;
incrementa(k);
console.log(k);
```

▼ Soluzione

A. Stampa `Area: undefined`: la funzione calcola `a` ma non la restituisce. Correzione: `return base * altezza / 2;`.

B. Stampa `8`, e `calcolo finito` **non** viene mai stampato: dopo il `return` la funzione termina, le istruzioni successive non vengono eseguite. Correzione: mettere il `console.log` **prima** del `return` (o toglierlo). Il `return` è una parola chiave che esiste **solo dentro le funzioni**: non si può usare in un blocco qualsiasi per «tirare fuori» un valore.

C. Stampa `NaN`: la funzione ha due parametri ma viene chiamata con un solo argomento; `b` vale `undefined` e `4 + undefined` è `NaN`. Correzione: chiamare `media(4, 6)` (servono tutti gli argomenti). JavaScript non segnala l'errore: a un parametro senza argomento dà il valore `undefined` (il valore di tutto ciò a cui non è stato assegnato niente), e poi cerca di andare avanti, ma un'operazione tra un numero e `undefined` non può dare un numero.

D. Stampa `5`: `n` è una **copia** del valore di `k`, modificarla non cambia `k`. Correzione: la funzione deve restituire il nuovo valore e il chiamante deve assegnarlo.

```
function incrementa(n) {
  return n + 1;
}
let k = 5;
k = incrementa(k);
console.log(k);                                  // 6
```

Esiste una seconda correzione, che però **non consigliamo**: scrivere una funzione `incrementaK()` senza parametri che lavora direttamente su `k`. Funziona perché, non trovando `k` tra le sue variabili locali, JavaScript la cerca tra quelle **globali** e trova quella del programma principale (vedi l'Esercizio 7). Ma la funzione diventa utilizzabile **solo** con quella variabile: la versione con il `return` si può usare con qualunque valore.

Esercizio 7

A. Cosa stampa questo programma? Tenete traccia del valore della variabile globale `x`.

```
let x = 10;

function f(x) {
  x = x + 1;
  return x * 2;
}

function g() {
  let y = x + 1;
  x = y;
  return y;
}

console.log(f(x));
console.log(x);
console.log(g());
console.log(x);
console.log(f(x));
```

B. Cosa succede eseguendo questo programma?

```
function conta() {
  let c = 0;
  c++;
}
conta();
console.log(c);
```

C. Cosa succede eseguendo questo programma?

```
for (let i = 0; i < 3; i++) {
  console.log(i);
}
console.log("fine:", i);
```

▼ Soluzione

A. Nella funzione `f`, `x` è il **parametro** (una variabile locale che nasconde la `x` globale: shadowing). Nella funzione `g` non c'è nessuna `x` locale, quindi `x` è la variabile **globale**.

```
22   f(10): il parametro x diventa 11, restituisce 22; la x globale resta 10
10   la x globale non è cambiata
11   g(): y = 10 + 1 = 11, x globale = 11, restituisce 11
11   la x globale ora vale 11
24   f(11): il parametro x diventa 12, restituisce 24
```

B. Stampa un errore: `ReferenceError: c is not defined`. La variabile `c` esiste solo **dentro** la funzione `conta`: fuori da essa non è visibile.

C. Stampa `0`, `1`, `2` e poi un errore: `ReferenceError: i is not defined`. La variabile `i` dichiarata con `let` nel `for` esiste solo dentro il ciclo.

Esercizio 8

A. Scrivete la funzione `massimo3(a, b, c)` che **restituisce** il più grande dei tre numeri. Provatela con `(3, 9, 4)` e `(-1, -5, -3)`.

B. Scrivete la funzione `tavola(n)` che **stampa** la tavola pitagorica `n × n`: una riga per ogni numero da 1 a `n`, con i prodotti separati da uno spazio. Usate **due cicli annidati**. Con `tavola(4)` stampa:

```
1 2 3 4
2 4 6 8
3 6 9 12
4 8 12 16
```

C. Scrivete un programma che chiede un intero positivo `N` (ripetendo la domanda se non è valido) e stampa **su un'unica riga** tutti i multipli di 2 o di 3 fino a `N`. Ad esempio con `N = 20`:

```
2 3 4 6 8 9 10 12 14 15 16 18 20
```

▼ **Soluzione**

```
// A.
function massimo3(a, b, c) {
  let massimo = a;
  if (b > massimo) {
    massimo = b;
  }
  if (c > massimo) {
    massimo = c;
  }
  return massimo;
}

console.log(massimo3(3, 9, 4));                 // 9
console.log(massimo3(-1, -5, -3));              // -1

// B.
function tavola(n) {
  for (let i = 1; i <= n; i++) {                // ciclo esterno: una riga per ogni giro
    let riga = "";                              // la riga riparte vuota ad ogni giro
esterno
    for (let j = 1; j <= n; j++) {              // ciclo interno: le colonne della riga
      riga = riga + (i * j) + " ";
    }
    console.log(riga);
  }
}

tavola(4);

// C.
let N;
do {
  N = Number(prompt("Inserire un intero positivo: "));
} while (!Number.isInteger(N) || N <= 0);

let riga = "";
for (let i = 1; i <= N; i++) {
  if (i % 2 == 0 || i % 3 == 0) {
    riga = riga + i + " ";
  }
}
console.log(riga);
```

Nella funzione `tavola` i due cicli partono da `1` (e non da `0`) perché `i` e `j` servono anche **come fattori**: se partissero da `0`, la prima riga e la prima colonna conterrebbero solo `0`, perché `0 * j` e `i * 0` valgono sempre `0`. I numeri non sono incolonnati bene quando passano da una a due cifre (con `tavola(10)`): per allinearli servirebbe una formattazione diversa, che qui non ci interessa; basta che la tavola sia **quadrata** (`n` righe, `n` colonne).

Nella funzione `tavola` la variabile `riga` è dichiarata **dentro** il ciclo esterno: ad ogni nuova riga riparte da `""`. Se fosse fuori, ogni riga conterrebbe anche quelle precedenti. La funzione non restituisce niente: il suo scopo è **stampare**. Ogni riga ha uno spazio finale.

Esercizio 9

Scrivete la seguente funzione **ricorsiva**. Prima individuate il **caso base** e il **passo ricorsivo**.

`potenza(base, esp)` che restituisce `base` elevato a `esp`, con `esp` intero maggiore o uguale a 0. Non usate `**` né `Math.pow`. Provatela con `potenza(2, 10)`, `potenza(3, 3)`, `potenza(5, 0)`.

▼ **Soluzione**

```
// caso base: esp == 0; passo: base * base^(esp-1)
function potenza(base, esp) {
  if (esp == 0) {
    return 1;
  }
  return base * potenza(base, esp - 1);
}

console.log(potenza(2, 10));                    // 1024
console.log(potenza(3, 3));                     // 27
console.log(potenza(5, 0));                     // 1
```

 Ludovica Pannitto - Università di Pisa - CC BY-NC-SA 4.0